Perspectives to Promote Modularity, Reusability, and Consistency in Multi-Language Systems

Hyacinth Chijioke Ali

A thesis submitted to McGill University in partial

fulfillment of the requirements of the degree of

Doctor of Philosophy

Department of Electrical & Computer Engineering
McGill University
Montréal, Québec, Canada

September 23, 2022

Abstract

In modern software systems, software modellers often use different modelling languages and multiple views to describe the characteristics of a complex system. This multi-language system allows modellers to express a specific system characteristic with the most appropriate modelling languages and notations. With the proliferation of independently developed and continually evolving modelling languages, it becomes more challenging to reuse or combine multiple languages in a multi-language modelling environment. Whenever models are collectively used to describe a system from different points of view, it is essential to keep the related models consistent. However, maintaining consistency across models is a difficult task, and in particular while supporting the independent evolution of the used modelling languages. Moreover, in a multi-view system, a modeller needs to navigate from one model element to another one in a related model in order to understand and modify the system under development. However, providing such navigation without dedicated support from the modelling environment is a daunting task.

In this doctoral thesis, we present a framework for the specification and development of *multi-language* systems based on *perspectives* to promote modularity in language reuse, inter-language consistency, combination of languages, and generic navigation of model artefacts. A *perspective* groups different languages for a modelling purpose, defines the role of each participating language, and specifies a generic navigation mechanism to traverse different related model elements. Furthermore, a *perspective* defines composite actions for building a consistent multi-model system and maintaining the links between different model elements.

The aim of this framework is to streamline the combination of multiple languages for a given purpose by providing a domain-specific language that a perspective designer can use to specify a perspective. The framework then employs a generative approach to generate tool support for the perspective, in particular to ensure appropriate language registration, model consistency (across language boundaries) when editing models, and generic navigation of model elements. The perspective designer only needs to focus on specifying relationships between different languages. Hence, a designer is freed from the full complexity as well as the error-prone implementation of consistency and navigation mechanisms.

We evaluate our approach with a perspective (*Fondue Requirement*) aimed at requirements elicitation and specification that combines five different languages. This *perspective* illustrates how a perspective designer can leverage our framework to register languages, specify perspectives, and then generate the implementation of the perspective. In addition, we analyse language actions for the five languages to demonstrate the benefits of perspective actions. To ensure the completeness and correctness of our approach, we further evaluate our framework with two notable multi-language modelling environments: the *User Requirements Notation* and the *Palladio Component Model*. Here, we focus on the relationships between different languages in each *perspective* and then show how our approach handles them.

Abrégé

Dans les systèmes logiciels modernes, les modélisateurs de logiciels utilisent souvent différents langages de modélisation et plusieurs vues pour décrire les caractéristiques d'un système complexe. Ce système multilingue permet aux modélisateurs d'exprimer une caractéristique spécifique du système avec le langage de modélisation le plus approprié. Les outils développés pour la modélisation et langages de modélisation eux-mêmes sont en constante évolution, et il devient par conséquent plus difficile de réutiliser ou de combiner plusieurs langues dans un environnement de modélisation multilingue. De plus, quand differents modèles sont utilisés en combinaison pour décrire un système, il est essentiel de garder les modèles cohérents. Malheureusement il est difficile de maintenir la cohérence entre les modèles tout en gérant l'évolution indépendante des langages de modélisation. De plus, le modélisateur doit pouvoir naviguer à partir des éléments d'un modèle vers d'autres éléments dans d'autres modèles qui y sont reliés pour bien comprendre et pour pouvoir modifier le système en cours de développement. Cependant, la navigation générique est difficile à réaliser sans support dédié de l'environnement de modélisation.

Dans cette thèse de doctorat, nous présentons un cadre pour la spécification et le développement de systèmes multilingues basés sur la notion de perspective. Une perspective favorise la modularité dans la réutilisation des langues de modélisation, la cohérence inter-langues, la combinaison de langues et la navigation générique de modèles. Une perspective regroupe différents langages pour aboutir à un but de modélisation précis. La perspective définit le rôle que chaque langue participante prends dans la perspective et spécifie un mécanisme de navigation générique qui permet l'utilisateur de traverser les liens de cohérance d'un modèle à un autre. Par ailleurs, une perspective définit des actions composites pour la construction un système multi-modèle cohérent et maintient automatiquement les liens entre les différents éléments de modèles qui sont logiquement reliés.

L'objectif de ce cadre de developpement est de faciliter la combinaison de plusieurs langues de modélisation dans un but précis et de permettre au concepteur de perspective de se concentrer uniquement sur la spécification des relations entre les différentes langues. Une approche générative assure alors un enregistrement automatique des langues utilisées, une gestion automatique de la cohérence des modèles et la navigation entre les éléments des modèles.

Nous évaluons notre approche en concevant une perspective (Fondue Requirements Perspective) visant à l'élicitation et la spécification des exigences. Cette perspective combine cinq langages différents et illustre ainsi comme un concepteur de perspective peut tirer profit de notre cadre pour enregistrer les langues, spécifier la perspective, et finalement générer l'implémentation de la perspective. De plus, nous analysons les actions pour les cinq langues afin de démontrer les avantages des actions générées pour la perspective. Afin d'assurer l'exhaustivité et l'exactitude de notre approche, nous évaluons notre cadre avec deux environnements de modélisation multilingues supplémentaires connus: la notation des besoins de l'utilisateur (User Requirements Notation) et le modèle de composants Palladio (Palladio Component Model). Ici, nous nous concentrons sur la relations entre les différentes langues dans chaque perspective, puis nous démontrons comment notre approche les gère.

Dedicated to the memory of my father, Ali Omeh, who played a crucial role in my career. You are gone, but your decision that I have to attend primary school has led to this Ph.D. You did not have anything to back up your decision, but your insistence completely changed my career.

Acknowledgements

It was a challenging and rewarding experience to see that this doctorate work culminates in a conclusion. I did not do it alone. Therefore, I would like to express my profound gratitude to all those who contributed directly or indirectly to the success of this thesis.

First, I would especially like to thank my primary advisor Gunter Mussbacher for his guidance, support, patience, and understanding throughout my doctoral research work. In addition, I salute his courage for accepting me as his student, even when he understands that I had a more specialized background in electrical engineering, with little or no specialized knowledge in software engineering. At last, I am happy that we managed to complete the task successfully with some remarkable achievements. Also, I express my warm gratitude to my co-supervisor Jörg Kienzle for his support, understanding, and guidance throughout this doctoral research. I truly appreciate my supervisors for giving me the privilege and the necessary support to work with the TouchCORE tool, which helped me gain a lot of practical software engineering skills.

I wish to express my sincere appreciation to the rest of my supervisory committee members, Daniel Varro and Sebastien Mosser, for their valuable feedback throughout this research work. Thank you also for accepting to review my thesis.

I also want to thank the Arbour Foundation, McGill Engineering Student Center, McGill

Scholarships & Aid, Gunter Mussbacher (advisor), Jörg Kienzle (advisor), and Christian Ejike Ali (brother) for their funding throughout my doctoral research work.

During these years, I was fortunate to be part of the Software Engineering Lab at McGill. I worked with several amazing research students who also contributed significantly to the success of this research work. I express my deepest gratitude to Keheliya Gallaba for his friendship and cooperation, especially at the beginning of my program. Also, I especially want to thank Matthias Schöttle who equally offered detailed guidance, especially, on how to navigate the TouchCORE tool. I would like to thank Aprajita, Marton Bur, Ruchika Kumar, Rijul Saini, Omar Alam, Yanis Hattab, Rohit Verma, and many more for their valuable feedback, friendship, and for welcoming me to their respective laboratories.

I would like to express my deep gratitude to my friends and family for their love, support, endurance, and understanding. I especially thank my elder brother Christian Ejike Ali, who oversees most of the funding, especially at the beginning of this program.

In addition, I would like to deeply appreciate my wife, who bore a lot of burden because of this program. During this program, my wife was alone in Nigeria taking care of our children for more than two years. I also thank my children for their patience, especially for my inability to be with them as required due to this program.

And finally, I want to thank my mother for her support, patience, and understanding throughout my education. She is a small businesswoman, but she often offers great financial support for my pursuit of education.

Contents

List of Figures

List of Tables

List of Acronyms

AI	artificial intelligence.
ALM	application lifecycle management.
API	application programming interface.
COLD	Concern-Oriented Language Development.
CORE	concern-oriented reuse.
DSL	domain-specific language.
EMF	eclipse modelling framework.
GPL	general-purpose language.
IDE	integrated development environment.
IoT	internet of things.
LEM	language element mapping.
MDA	model-driven architecture.
MDE	model-driven engineering.
MEM	model element mapping.
MLDE	multi-language development environment.
MOF	meta-object facility.
OCL	object constraint language.

OMG	object management group.
OSM	Orthographic Software Modelling.
PCM	palladio component model.
PML	perspectives for multi-language systems.
SLE	software language engineering.
SPL	software product line.
sum	single-underlying model.
SysML	system modelling language.
UML	unified modelling language.
URN	user requirements notation.
VCU	variation, customization, and usage interfaces.
VSUM	virtual single underlying model.

Chapter 1

Introduction

In this chapter, we summarise the context of this research work and then provide a brief motivation for the study with a review of the existing approaches' deficiencies. Furthermore, we present the research problems and then the methodology we follow to address the challenges. And finally, we outline the contributions as well as the publications based on this research and then present the structural content of the thesis.

1.1 Context of the Research Work

The development of some software systems seems easy and straightforward, especially when the system is small, for example, a basic software program to implement a simple calculator with addition, multiplication, division, and subtraction operators. This type of software system can be accomplished after a few days of basic training. On the other hand, it is a daunting task to write a complex program for a large system. Practically, it is almost impossible for software developers to write a bug-free program for large and complex systems. In principle, writing a 100,000-line program is much more difficult than 1000 times the effort of writing a 100-line program [1]. Hence, the complexity of a software system grows exponentially with the size of the system. Nonetheless, these challenges can be reduced if the program can be broken down into smaller sizes.

These problems of building software systems are significantly aggravating due to the increasing and varying demands of software users. In addition, the advent of data-driven software systems has particularly compounded the complexity of modern software systems. Consequently, software systems are now developed with various software technologies, such as Artificial Intelligence (AI), Machine Learning, and Internet of Things (IoT), to accommodate different dimensional requirements of the system. Software complexities can be categorised as follows:

- **Inherent Complexity**: This kind of complexity is related to the nature of the system. While some systems may be easy to understand even with a fairly large size, others may be more complicated due to the relationships within and across the system. Software engineers often follow a systematic approach to reason about such an inherently complex system.

- **Size Complexity**: These problems can be caused by the complex nature of the business domain or the efforts required to deal with different technologies to develop such a system. Software engineers usually address this kind of complexity by splitting the program into smaller sets of programs and then focusing on the essential parts of the system, instead of the entire system.

- **Uncertainty Complexity**: This kind of problem is mostly caused due to the business requirements or legal rules. The requirements of a program are often collected first before the design of the system and then the implementation. Although engineers often adopt a systematic approach to gather program requirements, these requirements are bound to change, and then the changes need to be propagated through all the stages of the software development life cycle to ensure that the expected behavior of the system is always maintained. This continuous modification of a system requirement implies that software systems are bound to evolve over time. On the other hand, uncertainty in assumptions about the world often complicates the software program. Software practitioners often make both rough assumptions and implicit assumptions, which generally do not take into account all corner cases.

In summary, software systems are often complex and will evolve over time. This continuous evolution of software systems needs to be systematically considered during the inception of the system.

Software engineering, similar to other engineering disciplines, aims to adopt systematic approaches to reduce software complexities, both during the development and operation stages. Software engineering fosters software development environments that promote the production of software systems that are reliable, usable, and maintainable [2]. During the development of a software system, a software engineer applies software engineering principles to elicit requirements, design, implement, test, deploy, and then maintain the software system. These different stages of a software development life cycle often require different expertise as well as different technologies, including software languages. This segregation of technologies within a software development environment is especially evidenced in complex systems, e.g., Auto Flight Control System, 5G Network, and Self-Adaptive Systems, to name a few.

Furthermore, most modern software systems are highly dependent on data to provide business insight. The integration of data and the use of several technologies during the software development raise several complexities that are difficult to deal with. Some of these challenges include maintaining consistent information at different levels as well as at different points of view of the software under development. Also, preserving traceability from the requirement level through to the deployment stage is a challenging endeavour. On the other hand, the size of systems often grows quickly, which presents a more challenging environment for working with such systems. These issues have been in existence for decades; however, the recent emergence of several modern technologies and the desire to combine them in a modular fashion further complicate the entire software development process.

To address these challenges, software engineering advocates modularity [3] during software development. The modularity approach decomposes a complex system into manageable and loosely coupled smaller systems so that software engineers can reason about and then work with a specific smaller module at a time. One of the state-of-the-art modular principles is the separation of concerns [4], which divides a complex system into a distinct section that addresses a particular concern. To promote the modularity paradigm in software engineering, several technologies, including modelling languages[1] and Model Based System Engineering, have emerged aiming to reduce the difficulties that are associated with the development of complex systems. Additionally, these technologies improve usability, speed, security, understanding, resilience, and time to market, to name a few, of software systems. This proliferation of software technologies, motivated by the separation of concerns, relies on the use of specialised technologies at different stages or view points of a software system to promote a more robust and reliable production of software systems. This modular approach has many benefits; however, it is challenging to

[1]e.g., Unified Modelling Language (UML), Systems Modelling Language (SysML), User Requirements Notation (URN), Palladio Component Model (PCM), and i* Framework for Goal-Oriented Modelling, to name a few.

combine different technologies while developing a software product. These challenges include consistency management between different software technologies; compatibility between the underlying framework of each technology; reuse of software artefacts; coordination between different software technologies; and generic navigation across different artefacts potentially from different technologies. These complexities have increased in the software engineering community and require a dedicated and systematic approach to address the complicated nature of software development processes. The modular approach, as well as the separation of concerns and other related approaches, such as *Software Product Lines* [5], is widely applied in Model-Driven Engineering (MDE) [6] to reduce complexities during software development and then promote production of more user-friendly and reliable systems.

Model-Driven Engineering advocates the use of software models, e.g., UML class diagram and URN goal model, to facilitate software development throughout the development life cycle. Also, *MDE* promotes a better understanding of the system, better documentation, code generation, better analyzability, communication between team members, and communication between stakeholders. A modelling language defines the conceptual relationship, representation, and semantics of models that conform to the language. To simplify the complexities of modern software systems, *MDE* employs separation of concerns by using multi-language modelling environment[2] to develop a specific system. This approach allows a more specialised language to be used for a dedicated part or level of the system under development. In addition, some models are so large that it becomes more difficult to work with them. Consequently, *MDE* introduced Multi-View Modelling [7] to address this challenge, i.e., the representation of a single model

[2]i.e., a modelling environment that supports the combination of two or more modelling languages during the development of a software system.

with several views to narrow the focus of a software developer. In all, these measures aim to facilitate the separation of concerns, which helps to deal with complex systems and, as well, promote a better understanding of the system.

1.2 Motivation

In model-driven engineering (MDE), modellers often use different modelling languages to *capture* the requirements; *describe* the characteristics; and then *prescribe* the structure, as well as the behaviour, of a system under development [8, 9]. This is also the case in Model-Based System Engineering, where modellers often combine different SysML diagrams (e.g., block definition diagram, internal block diagram, and requirement diagram) to model the system under development. At each level of abstraction or viewpoint of a complex system, specialised modelling languages are required to express the characteristics of the system in the most appropriate way. For example, a UML use case model is often used during requirement elicitation, while a UML state diagram is predominantly used during software design. When a general-purpose modelling language is inappropriate or difficult to use in a given context, domain-specific languages can be defined to handle the development concerns of the application domain.

In any case, when multiple languages, as well as, multiple views are used to describe a system, special care is required to ensure that the different views collectively and coherently describe the system. Sustaining the consistency between different models, especially when languages/models are added dynamically to the system, is a very hard and costly endeavour. This requires a systematic approach to combine different languages and then preserve the consistency between different models conforming to the languages. In addition, navigation support within and across models has received only limited attention. This is the case even though studies show the importance of good visualization and navigation mechanisms in both software usage and during development [10–12].

Existing approaches to preserve consistency in a multi-language modelling environment include those that rely on a single underlying model (SUM) [13–15] that captures all the conceptual relationships and consistency rules of a system based on a single metamodel. However, modifying this single metamodel to support new languages or update consistency rules may be rather complicated, because the modeller must simultaneously deal with the full combined complexity of the embedded languages. Other approaches retain separate metamodels for the languages and establish consistency conditions between language elements across language boundaries. However, these approaches react to inconsistencies after they occur, instead of preventing them. Vitruvius [9], for example, uses reactive model transformations to fix inconsistencies across model and language boundaries after one of the involved models is updated.

1.3 Statement of the Problem

To address the above concerns, we formulate the problem statement in terms of a research gap, which needs to be improved or new concepts introduced to promote the multi-language modelling environment.

- *Modular combination of languages to promote software evolution.*

  Software languages are often grouped to describe a software system. These languages are expected to collectively and coherently describe the system under development.

On the other hand, these languages often exist independently, and combining them requires a systematic approach.

To uphold a coordinated collaboration between the languages, some researchers favour integration of the metamodels of the languages as a single metamodel, which contain all the consistencies that are required to be maintained during modelling time. This approach addresses the challenge of consistency to a greater degree. However, this integration approach complicates both the maintenance and the evolution of the tool containing the languages. This is the case since a single change in any part of the single metamodel can potentially affect the whole system.

Alternatively, languages can be loosely coupled with a virtual model approach, which externally establishes the required consistencies between different languages. This approach keeps each language separate as a module in the system, i.e., modular combination. Unlike the tightly coupled integration approach, the modular approach allows each language to evolve independently, hence promoting modularity and evolution of the system. However, the modular combination of the languages is a non-trivial task because the software engineer has to face the consistency challenges, which need to be established external to the metamodel of each language.

- *Composite reuse of languages (i.e., reusing a language with its abstract syntax, concrete syntax, and semantics as a whole).*

  Software reuse is a powerful software engineering principle that allows a system under development to reuse existing software artefacts, instead of building them from scratch. This methodology accelerates the product-to-market process and promotes software quality.

Software languages are software, too. Hence, software languages can be reused just like other software artefacts. However, it may require more effort to reuse a software language because it comprises some building blocks, including abstract syntax, concrete syntax, and semantics. Hence, it is essential to reuse a language as a composite software artefact, which then provides the full potential of a language in the reused context. This composite reuse methodology is a non-trivial task that requires dedicated support from the software engineering community.

- *Consistency management between language artefacts.*

  Multi-language environment often comprises several model artefacts from different languages, which are collectively used to describe the system under development. In this situation, some model information are often scattered across software artefacts, which represent the same entity in the system. This scattered information that refers to the same entity is required to be consistent.

  However, this is often challenging in an environment that promotes the separation of concerns. When we keep the languages separate as well as their models, these related artefacts can evolve independently. Hence, managing the consistency is not an easy task, especially in a modular multi-language modelling environment.

- generic navigation mechanism across model elements potentially from different modelling languages.

  Navigation is an important mechanism in a modelling tool, but it often receives less attention from the software engineering community. Navigation support allows modeller to traverse related model elements within and across models, potentially from multiple languages. Some tools implement a navigation mechanism with a fixed set of languages.

However, with the proliferation of domain-specific modeling languages (DSMLs), it cannot be assumed that a fixed set of modelling languages is used to develop complex software systems. Rather, a flexible modelling environment needs to be provided that allows sets of languages to be integrated as the needs arise. Consequently, the corresponding set of models needs to be navigated in a generic way.

1.4 Methodology

To address the above problems, we adapted some *MDE* paradigms, which include metamodelling, template-based code generation, and proof-of-concept implementation. We provide a metamodel which allows perspective designers to combine independently-developed modelling languages in a software tool. To instantiate the metamodel, we implement two domain-specific languages which aim to simplify the composition of the reused languages (DSLs). In addition, the DSLs provide code generators that generate the implementation of the consistency management as well as the generic navigation mechanism across different models that conform to the reused modelling languages. As a proof-of-concept, we implement different perspectives with the TouchCORE [16] tool.

1.5 Thesis Contributions

Our work defines **P**erspectives for **M**ulti-**L**anguage Systems (PML), a framework for assembling multi-language systems based on existing, independent languages. We support a novel, proactive approach for preventing the occurrence of inconsistencies by monitoring and augmenting the language actions that a modeller uses to create and update a model. PML maintains consistency conditions including equivalency, equality, and multiplicity constraints across different model elements from different languages. *PML* promotes modular combination of languages and facilitates the consistency and reuse of an existing language across other languages and software systems. Additionally, *PML* provides a navigation mechanism to traverse different model elements potentially from different languages.

The main contributions of this doctoral research are as follows:

- ***PML* Framework:** The *PML* framework provides an architecture and approach for defining perspectives and then maintaining consistencies between different languages.
- ***PML* Metamodel:** We present a metamodel that combines independently existing modelling languages (with their language actions). The metamodel specifically targets languages defined with Ecore [17]. However, it can be applied to modelling languages based on other metamodelling environments with slight modifications.
- **Generic Templates:** We present generic templates that cover relationships between two or more metaclasses and dictate the sequence of actions that can prevent and, as well, maintain model consistencies. These templates handle cyclic dependencies and support complex language actions, i.e., actions that affect more than one language element, for which consistency needs to be ensured with one or more language elements in other models.
- ***PML* Domain Specific Languages:** We present two DSLs that assist a perspective designer to combine different languages, specify perspective actions, and encode the relationships between different language elements, including navigation

relationships. From the perspective definition expressed in the provided DSLs and the generic templates, we generate perspective actions that can then subsequently be used by the modeller to execute consistent model changes on the set of models governed by the perspective instead of the original language actions. This generative approach allows the perspective designer to focus on these key relationships and frees them from the error-prone implementation of perspective actions.

- ***PML* Generic Navigation Mechanism:** We present a generic navigation mechanism that allows a perspective designer to specify navigable links between different model elements in a perspective. This navigation mechanism handles navigation within model elements as well as elements across different models from different languages. Additionally, we use a generative approach to implement the navigation for each perspective; hence, perspective designers are freed from the manual implementation of each navigation in a perspective.

- **Comparison of *PML* Navigation with Notable UML Tools Navigation:** We analyse the navigation facilities of several popular modelling tools and then show how our navigation approach can support the discovered navigation facilities. On the other hand, no tool offers complete support for all navigation features provided by our navigation mechanism.

As proof-of-concept, we implement our approach in the TouchCORE tool [16], and illustrate our approach by combining five different modelling languages (class diagrams, use case diagrams, collaboration diagrams, use case maps, and a domain-specific modelling language) for the purpose of requirement elicitation. This case study shows (i) that all proposed perspective actions are needed to maintain consistency conditions in a multi-language modelling environment, (ii) that all templates are needed (one for update actions, three for delete actions, and twelve for create actions), and (iii) that only a small percentage of perspective actions involve complex language actions requiring a larger specification effort. Furthermore, we evaluate our framework with two notable multi-language modelling environments: *User Requirements Notation* and *Palladio Component Model*. Here, we focus on the relationships between different languages in each *perspective* and then show how the *PML* framework can be applied to specify those relationships and then preserve the consistencies during the modelling time.

1.6 Publications and Presentations

All the publications, as well as presentations related to this doctoral research, are listed below in order of publication or presentation.

The doctoral candidate was the main contributor to the publications 1 - 6 and 8. The candidate performed the experiments, prepared the manuscripts and then presented each article in a conference or workshop, where applicable. The other authors to the publications 1 - 6 and 8 are the advisors of the doctoral candidate, who supervised the entire process that led to the publications and edited the publications.

Additionally, the doctoral candidate contributed significantly to the publication 7 (tool demo paper). The candidate implemented most of the several multi-language modelling features present in the paper and then presented the demo paper during the 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion. Also, the candidate actively participated in the writing of the manuscript. Other authors contributed some features and the writing of the manuscript.

1. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Towards a Framework for Multi-Language Reuse. Presentation, 11th Workshop on Modelling in Software Engineering (MiSE 2019), Montreal, Canada, May 2019.

2. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Generic Navigation of Model-Based Development Artefacts. 11th Workshop on Modelling in Software Engineering (MiSE 2019), Montreal, Canada, May 2019. IEEE CS, 35-38. DOI: 10.1109/MiSE.2019.00013.

3. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Towards Modular Combination and Reuse of Languages with Perspectives. 1st International Workshop on View-Oriented Software Engineering (VoSE), Munich, Germany, September 2019. IEEE CS, 387-394. DOI: 10.1109/MODELS-C.2019.00060.

4. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Generic Graphical Navigation for Modelling Tools. 11th System Analysis and Modeling Conference (SAM 2019), Munich, Germany, September 2019. Fonseca i Casas, P., Sancho, M.R., and Sherratt, E. (Eds.), System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Springer, LNCS 11753:44-60. DOI: 10.1007/978-3-030-30690-8_3.

5. Ali, H., Mussbacher, G., and Kienzle, J. (2020) Action-Driven Consistency for Modular Multi-Language Systems with Perspectives. 12th System Analysis and Modeling Conference (SAM 2020), Montreal, Canada, October 2020. ACM, 95-104 DOI: 10.1145/3419804.3420270.

6. Ali, H.. (2020) Multi-Language Systems Based on Perspectives to Promote Modularity, Reusability, and Consistency. 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2020), Montreal, Canada, October 2020. ACM, article no. 29, 1-6. DOI: 10.1145/3417990.3419489

7. Schiedermeier, M., Li, B., Languay, R., Freitag, G., Wu, Q., Kienzle, J., Ali, H., Gauthier, I., and Mussbacher, G. (2021) Multi-Language Support in TouchCORE. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS 2021), pp. 625-629, DOI: 10.1109/MODELS-C53483.2021.00096

8. Ali, H., Mussbacher, G., and Kienzle, J. (2021) Perspectives to Promote Modularity, Reusability, and Consistency in Multi-Language Systems. Innovations in Systems and Software Engineering, Special issue on Software and Systems Reuse, DOI: 10.1007/s11334-021-00425-3

The following publications are not directly related to this thesis. However, the doctoral candidate actively participated in the projects as well as the publications during his Ph.D. program.

Publications 1 and 2 are the result of two different research workshops, and the doctoral candidate was among the participants. The candidate contributed to the design, implementation, and writing of the manuscript. Other authors were actively involved in the design, implementation, as well as writing of the manuscript. For the publications 3 and 4, the candidate performed the experiments, prepared the manuscripts, and then presented each article in a conference, where applicable. The second author to each of the publications is the candidate advisor, who supervised the entire process that led to the publications and edited the publications.

1. Combemale, B., Kienzle, J., Mussbacher, G., Ali, H., Amyot, D., Bagherzadeh, M., Batot, E., Bencomo, N., Benni, B., Bruel, J.-M., Cabot, J., Cheng, B.H.C., Collet, P., Engels, G., Heinrich, R., Jézéquel, J.-M., Koziolek, A., Mosser, S., Reussner, R.,

Sahraoui, H., Saini, R., Sallou, J., Stinckwich, S., Syriani, E., and Wimmer, M. (2020) A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems. IEEE Software, IEEE. DOI: 10.1109/MS.2020.2995125.

2. Mussbacher, G., Combemale, B., Kienzle, J., Abrahão, S., Ali, H., Bencomo, N., Bür, M., Burgueño, L., Engels, G., Jeanjean, P., Jézéquel, J.M., Kühn, T., Mosser, S., Sahraoui, H., Syriani, E., Varró, D., and Weyssow, M. (2020) Opportunities in Intelligent Modeling Assistance. Expert Voice, Software & Systems Modeling (SoSyM), Springer 19:1045-1053. DOI: 10.1007/s10270-020-00814-5

3. Ali, H., and Mussbacher, G. (2020) Layout Merging with Relative Positioning. 12th System Analysis and Modeling Conference (SAM 2020), Montreal, Canada, October 2020. ACM, 106-115. DOI: 10.1145/3419804.3420271. **(Best Paper Award)**

4. Ali, H., and Mussbacher, G. (2021) Layout merging with relative positioning in Concern-Oriented Reuse hierarchies. Information and Software Technology, Volume 143, ISSN 0950-5849, DOI: 10.1016/j.infsof.2021.106757

1.7 Thesis Outline

In the remainder of this thesis, we first provide the background of the research work in Chapter 2 and then motivate perspectives with some examples in Chapter 3. Furthermore, we explain the architecture of the *PML* framework in Chapter 4 and then present the *PML* navigation mechanisms in Chapter 5. Chapter 6 presents generic templates and the template workflow for the relationships between different language elements, which govern the automatic generation of the augmented language actions. In Chapter 7, we show how

PML handles advanced features of a perspective, which promote the reuse of a language that can have multiple models in a perspective, as well as conditional *LEMs*. In Chapter 8, we present the DSLs for languages, perspectives, and navigation facilities, while Chapter 9 validates our approach by building a complex *perspective* that combines class diagrams, use case diagrams, collaboration diagrams (environment model), a domain-specific model (operation model), and use case maps. Also, we illustrate how the *PML* framework can handle relationships in other notable multi-language modelling environments. We compare the *PML* framework to other contemporary multi-language systems in Chapter 10, while Chapter 11 summarizes our contributions and presents future work. Appendix A presents the details of all the generic templates, while we present the definition of our *PML* DSL in Appendix B. In Appendix C, we present sample models of our DSL, which details how we register languages in a software tool, and finally, Appendix D presents a complete perspective specification (with our DSL) for the Fondue Requirement perspective.

Chapter 2

Background

In this chapter, we present an overview of Model-Driven Engineering (MDE) and then the related fundamental concepts that are used in the subsequent chapters of this thesis. First, we explain what models are in the context of *MDE*. Second, we introduce software languages and then software language engineering, which stipulates the best engineering practices for building languages. Finally, we provide fundamental concepts about the generic navigation mechanism and its roles in MDE.

2.1 Building Software Challenges

It has never been easy to develop a complex software system. System requirements are often gathered at the beginning of a software production, which encompasses interactions between the domain experts and the software practitioners, e.g., IT administrators and software engineers. The elicitation of requirements can be challenging, especially in a very complex business domain, which requires software practitioners to succinctly and precisely represent business requirements in the context that can facilitate software development. The difficulties of understanding a very complex business domain are usually compounded with all the efforts to manage large software teams over multiple stages of a project that spans many months or years. Furthermore, incessant demands from clients as well as the time-to-market pressures are inherent to many of the modern software products, which also complicates the software development challenges. In addition, with the emergence of data-driven software systems, several systems are inherently complex and seriously demanding to develop, for example, Auto Flight Control System, 5G Network, and IoT, to name a few.

In addition to these complexities, which are related to the system or business domain, there are also great complexities of the software technologies on which the software systems are based. Most software companies rely heavily on complex infrastructure technologies, which have evolved over many years and consist of various software packages that are purchased from many vendors. Often, these packages are not maintained or lack proper documentation. Depending on how tightly coupled the technologies are in a software tool or a system, it may present a significant challenge when some of the associated packages for each technology cannot be maintained. This is especially the case when other technologies depend on the outdated package to support the life cycle of the system in question.

In addition, the requirements of a software system are often continuously modified to keep the system on par with the demands of clients. These inevitable changes in software requirements are extremely challenging to propagate to software systems that are developed with different technologies because modifying a part of a system can make it inconsistent with other technologies. These problems might be more complicated if the underlying technologies are combined together as a single software artefact, instead of separating them into software modules to promote independent evolution.

Similar to other engineering fields, the building of software applications requires a systematic approach to eliminate or reduce the associated challenges, which aim to help software engineers evolve their solutions in flexible ways while reusing existing software artefacts [18].

2.2 Model-Driven Engineering

Model-Driven Engineering is a software engineering methodology that advocates the use of models for all software development tasks, including documentation, requirement gathering, design, and implementation of the software system. *MDE* aims to foster greater flexibility in the development as well as maintenance of software solutions. A software model improves reasoning and understanding of the system as well as promotes communication between software developers and other stakeholders.

With the concept of model transformation [19, 20], software models are often used to automate several processes during software development. This automation paradigm can lead to several goals, including faster development, better software quality, and greater productivity [21]. There are various concepts and software tools to facilitate the automation and other benefits of models during software development.

However, the increasing challenges of software development outlined above require a more streamlined approach to foster modern software development. One of the prominent *MDE* approaches is Model-Driven Architecture (MDA) [18] which is defined as the realization of *MDE* principles around a set of Object Management Group (OMG) [22] standards, which include, but are not limited to, the Unified Modelling Language (UML), the Meta-Object Facility (MOF), and the Object Constraint Language (OCL). UML is a popular modelling language that can be used to create models of software systems. MOF is a modelling language that is used for the definition of other modelling languages. OCL is a precise textual language for expressing constraints that cannot be shown diagrammatically in UML models. In the following subsections, we explain the concept of software models, software languages, and the principles for building software languages.

2.2.1 Software Models

To streamline the development process of a software system, software engineers embrace many methods to address development challenges. One of the notable approaches is the use of models and modelling [23, 24]. Models provide abstractions of software systems, which allow software engineers to reason about these systems by ignoring extraneous details and focusing on the relevant aspects of the system. Software engineers rely on models to understand complex real-world software systems. Models provide essential benefits, including, but not limited to, predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics to its various stakeholders [23]. Models are often used as a base for the implementation of software systems, or derived from an existing system or developed system to help understand the system's behaviour.

Figure 2.1 shows the general architecture of a modelling environment, each model is based on a formalism (or language), which precisely defines the concrete syntax (or notation), the abstract syntax, and the semantics of the model. The concrete syntax specifies the view or notation of the model; the abstract syntax defines the structural relationships between the conceptual states of the model; and the semantics defines the meaning of the model. In the following sections, we provide more information about the formalism and then present some *MDE* principles for building the software languages.

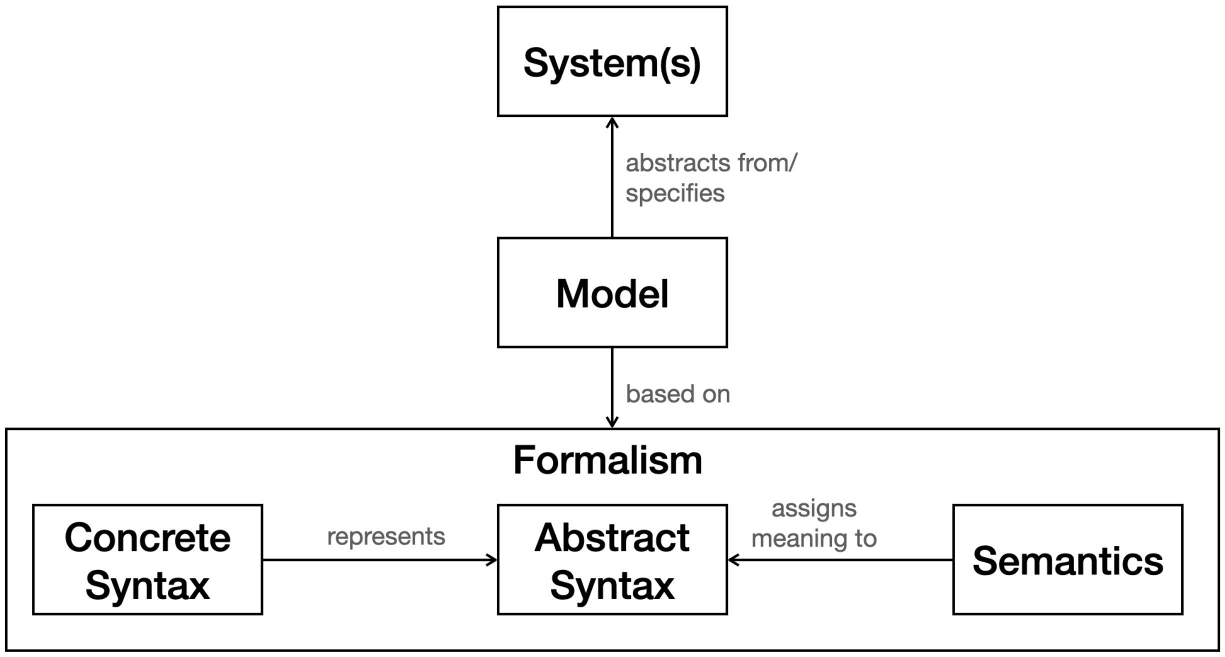


Figure 2.1: System, Model, and Formalism

2.2.2 Software Language

Software languages play essential roles in software engineering and computer science. There are several types of software languages, including modelling languages, programming languages, markup languages, and formal languages [25]. In general terms, there are two major categories of software languages: *General-Purpose Languages* (GPLs) and *Domain-Specific Languages* (DSLs) [26].

A GPL is a software language that targets much larger domains and hence provides a potentially suboptimal solution. Examples of prominent GPLs include popular programming languages, e.g., Java, C#, C, and modelling languages (e.g., UML class diagram). The expressiveness and notation of GPL are not tailored to a particular domain. Although a GPL can be used to develop any kind of software, domain experts may find it difficult to understand its concepts. This poses challenges for software developers, since it becomes more difficult to bridge the gap between problem space (domain experts) and the solution space (software developers). Therefore, the use of GPL limits the involvement of domain experts during the generation, specification, and implementation of a software system.

On the other hand, a DSL is a language whose expressiveness and notations are tailored to a particular domain of application [27,28]. It provides a more specific and efficient solution in a given domain. However, a DSL is not suitable for implementing any kind of software application across several domains. The main objective of a DSL is to bridge the gap between the problem space, i.e., where the domain specialists work, and the solution space, i.e., where the software developers work. Therefore, a DSL promotes the participation of domain experts during the life cycle of a software application. Hence, domain experts can create and manipulate models and other notations of a DSL [26]. Models developed by using a DSL are easier to understand and maintain [29] by stakeholders. Recently, DSLs have been adopted in a variety of domains including cyber-physical systems, computational sciences, and high-performance computing.

2.2.3 Software Language Engineering

The development of a software language is inherently a difficult endeavour [30, 31]. A software language is software, too [32]. Thus, it inherits all the complexities associated with the development of a software system, as well as its own specificities and implementation techniques [33, 34].

The building of a single DSL already faces many challenges, including maintainability, evolution, reuse, consistency, and user experience [30,35]. In a multi-language system, several DSLs are composed to prescribe and describe the characteristics of the system, which makes the aforementioned challenges even more difficult to deal with.

Since a DSL focuses on a small group of stakeholders, it is important that the benefits of a DSL offset the efforts required to build and maintain it. Therefore, it makes sense to apply software engineering techniques during the specification, implementation, use, and maintenance of a DSL. This notion led to the emergence of *Software Language Engineering* (SLE), which is defined as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [25].

Similar to GPL, constructing a DSL involves the creation of three main building blocks of a language: abstract syntax, concrete syntax, and semantics. In the following subsections, we briefly describe these constituents in the context of SLE.

Abstract Syntax:

The abstract syntax represents the key constituents of a language. It captures the concepts that exist in a language domain and establishes the structural relationship that exists between the concepts. There are two major ways of specifying the abstract syntax of a DSL: Context-Free Grammars (e.g., Backus-Naur Form [36]) and Metamodelling (e.g., Ecore and Meta-Object Facility (MOF) [37]). In this work, we focus on metamodelling where all the aspects of a language's abstract syntax are based on its metamodel. Models of a DSL must conform to the structural relationship captured in the abstract syntax (metamodel).

Concrete Syntax:

The concrete syntax defines the visual representation of a language. This constitutes a larger part of a user interface of a language. Concrete syntax can be represented textually, i.e., the use of characters to write programs that conform to a language, and graphically, i.e., the use of nodes and edges to establish models that conform to a language. In both cases, software developers use the visual representation of a language to create, modify, or delete instances of a language. Most of the programming languages (e.g., Java, C, Python) employ textual representation, while modelling languages (e.g., UML) are manipulated via the graphical representation. In either way, there are correspondences between the abstract syntax and concrete syntax elements.

Semantics:

The semantic of a language defines the meaning of its concepts. It promotes reasoning about the properties and runtime behaviour of the models or programs that conform to the language [30]. Furthermore, it facilitates the communication of one's understanding of a language to someone else. Here, we briefly present some categories of language semantics, which include the *operational*, *denotational*, *extensional*, and *translational* semantics:

- **Operational Semantics:** Operational semantics describes the operational behaviour of the concepts in a language (e.g., state transition system)
- **Denotational Semantic:** Denotational semantics associates mathematical concepts with the constructs of a language.
- **Extensional Semantics:** Extensional semantics expresses the semantics of a language based on the semantics of the super-language (e.g., UML Profiling)
- **Translational Semantics:** Translational semantics expresses the semantics of a language in another language with more precise semantics representation (e.g., Petri Nets mapping to Java)

2.3 Metamodelling

In this section, we focus on the definition of language abstract syntax in the form of a graphical model (metamodel). Metamodelling refers to the specification of the structural relationships between different types of element in a modelling language, i.e., the creation of a metamodel. Hence, a metamodel defines the structural rules as well as the relationships between different elements (metaclasses) of the modelling language. Then instance models are built on the bases of the modelling language's elements specified in the metamodel in order to abstract a concrete software system. Thus, metamodels serve as a blueprint that specifies elements and rules for creating and editing instance models.

To illustrate the relationships between a metamodel and its instance model, Figure 2.2 shows an excerpt of a UML Class Diagram metamodel and an instance of the class diagram elaborated based on the structural relationships and other rules of the metamodel. In addition, the figure defines some vital terms that are used in this thesis: *Language Element*, *Root Language Element*, *Nested Language Element*, and *Model Element*.

There are two main levels in this diagram: Model level and Metamodel level. At the metamodel level, we have a UML Class Diagram metamodel (excerpt), which comprises some metaclasses and their attributes. The metaclass `ClassDiagram` is the root metaclass - an instance of the root metaclass contains all the model elements, directly or indirectly, that are allowed to be defined by the metamodel. Each of the metaclasses, `Class`, `Attribute`, and `Operation`, is referred to as a language element. A language element can have zero or more attributes, which we define as a nested language element. For example, each of the name attributes in `Operation`, `Class`, and `Attribute` is a nested language element.

At the model level, we define a *Class Diagram* model (*Bank*), which captures very basic features of a bank and conforms to the specifications of the *Class Diagram* metamodel. The

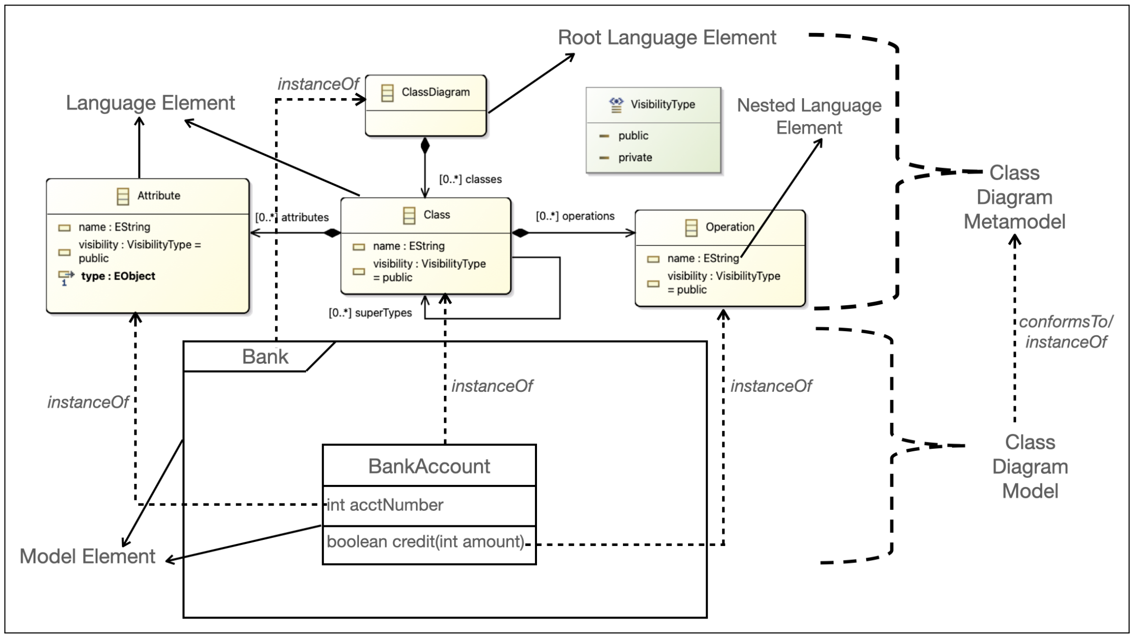


Figure 2.2: Class Diagram Metamodel (excerpt) and an Instance of the Metamodel

diagram elaborates a bank model, named *Bank*, that offers bank services to its clients. Each bank account should have an account number (*acctNumber*), which is an instance of the language element `Attribute` at the metamodel level. Furthermore, the bank allows their clients to deposit money into their account (*credit* operation), which is an instance of the `Operation` language element. The *Bank* element as well as the *BankAccount* element are called *Model Element*. Also, the *Bank* model element is an instance of the root language element (`ClassDiagram`), which we refer to as the owner or container of the instance model, and the *BankAccount* model element is an instance of the `Class` language element.

The metamodel is regarded as the type of class diagram model, and this relationship is not transitive. For example, an instance of the class diagram model, e.g., runtime user objects, is not an instance of the metamodel. Hence, the class diagram is the model of the runtime user objects, while the metamodel is the model of the class diagram. Figure 2.3

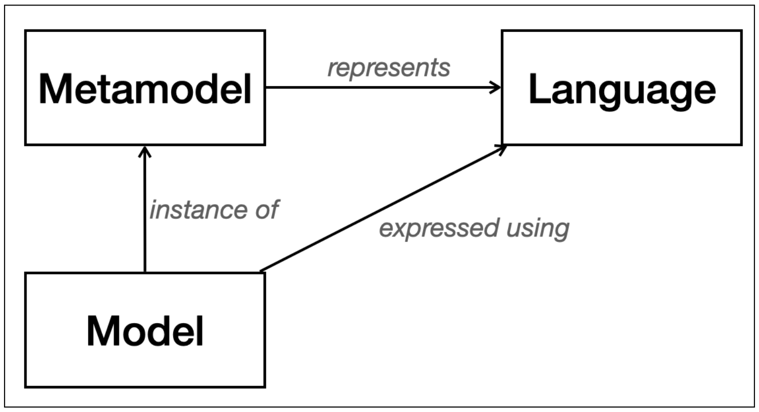


Figure 2.3: The Relationship Between Metamodel, Model, and Language

summarizes the relationship between a metamodel, a model, and a language. The metamodel represents the language as the formalism for which systems can be described or prescribed with the language in the form of a model [38].

2.4 Metamodelling Languages

A modelling language defines the abstract syntax that forms the rules that govern the creation of a model, which is based on the metamodel of the language. Similarly, a metamodelling language defines the formalism of a modelling language, i.e., a modelling language conforms to a metamodelling language. The model of a metamodelling language is referred to as metametamodel. In the following section, we briefly present Eclipse and then Eclipse Modelling Framework (EMF), which is a famous modelling framework for both definition and elaboration of instance models.

2.4.1 Eclipse

Eclipse is a large open source software, which is used for development and as a plug-in for software applications. Eclipse is designed to provide a highly integrated software tool platform, providing a forum for individuals and organisations that have the same interest to work on open source software [39]. Eclipse projects comprise generic frameworks, tools, and runtimes for building, deploying, and managing software across the software development processes. It is an Integrated Development Environment (IDE) for different programming languages which includes C, C++, Python, and Java. In this work, we use Eclipse and an Eclipse plug-in language workbench to develop two domain-specific languages and their code generators.

2.4.2 Eclipse Modelling Framework

Eclipse Modelling Framework (EMF) is a framework and code generation facility used by developers to accelerate software development in Model-Driven Engineering(MDE). EMF is an Eclipse plug-in and thus takes advantage of the resources provided by Eclipse. It is used to define the software domain model and then generate classes, code, and HTML page of a structured software data [40, 41]. Several software artefacts are required for a complete representation of a software model which includes Ecore class diagrams, software interface, and XML schema. In software modelling, developers would like to increase their productivity by limiting the manual production of all these artefacts. In this regard, EMF is a viable tool that can be used extensively to transform the software model from one form to another. For example, if a model is produced by using the specifications described in the XML schema, EMF provides tools and runtime supports that can produce a set of java interfaces that enable viewing and command-based editing for the model and a basic editor, along with

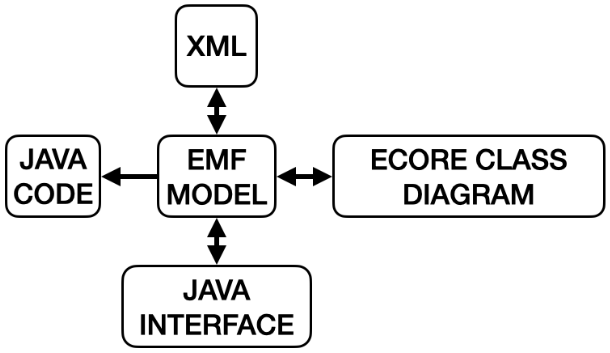


Figure 2.4: EMF Links Java Code, Java Interface, XML, and UML

the Ecore class diagram for the model. EMF also generates Java classes to implement the model. On the other hand, given a model in the Ecore class diagram, EMF can be leveraged to produce other software artefacts for the model. The EMF binding algorithm is shown in Figure 2.4. Thus, developers can design any model that matches their perspective or skills, and EMF will produce and link other models as well as the implementation code.

2.4.3 Ecore Metamodel

Ecore is an EMF core framework that is used to describe EMF models and runtime support for models [40]. The supports and description provided by the ecore model for EMF models include, but are not limited to, change notification, XMI serialisation, and an API for communicating with EMF objects. Ecore models provide detailed information on the packages, classes, and attributes of the domain model.

EMF modelling technique uses an ecore file which describes the metamodel of the EMF model. Ecore is a metamodel of itself, and thus it is a metametamodel [42]. Software applications that use the EMF model have .ecore extension files, which are represented in an XML schema.

An excerpt of the ecore metametamodel is shown in Figure 2.5, which captures some metaclasses that are essential for this work. Ecore is predominantly used to define the metamodel of modelling languages, including domain-specific modelling languages. In ecore, all elements are instances of the *EObject* model element. The `EPackage` allows language designers to retrieve an object representation of each language element, which can be used to implement the behavioural semantics of the language in question. Metaclasses of languages that conform to ecore metamodel are instances of the `EClassifier`, e.g., `Class`, `Attribute`, and `Operation` language elements in Class Diagram metamodel (see Figure 2.2) are instances of the `EClass`. The properties of metaclasses defined with ecore are called structural features (`EStructuralFeature`). Features that are typed using a primitive type or using an enumeration are called attributes (`EAttribute`), while features that are typed using a metaclass are called references (`EReference`). For example, the nested language element *name* in `Operation` metaclass (see Figure 2.2) conforms to the `EAttribute` metaclass. In this thesis, we use ecore to define the metamodel of the *PML* framework.

Figure 2.6 summarizes the *MDA* model hierarchy. The metametamodel is at the highest level (*M3*), which comprises models that define metalanguages, e.g., Ecore and MOF. Metametamodel conforms to itself. At *M2*, language is defined with metamodels that conform to the metametamodel at *M3*, e.g., UML class diagram language. These languages are then used at *M1* to prescribe or describe the real world system at *M1*, e.g., a bank UML class diagram model. And finally, at *M0*, we have runtime instances that conform to the models at *M1*. Levels *M3* to *M1* are referred to as the modelling world while *M0* is called the real world [43].

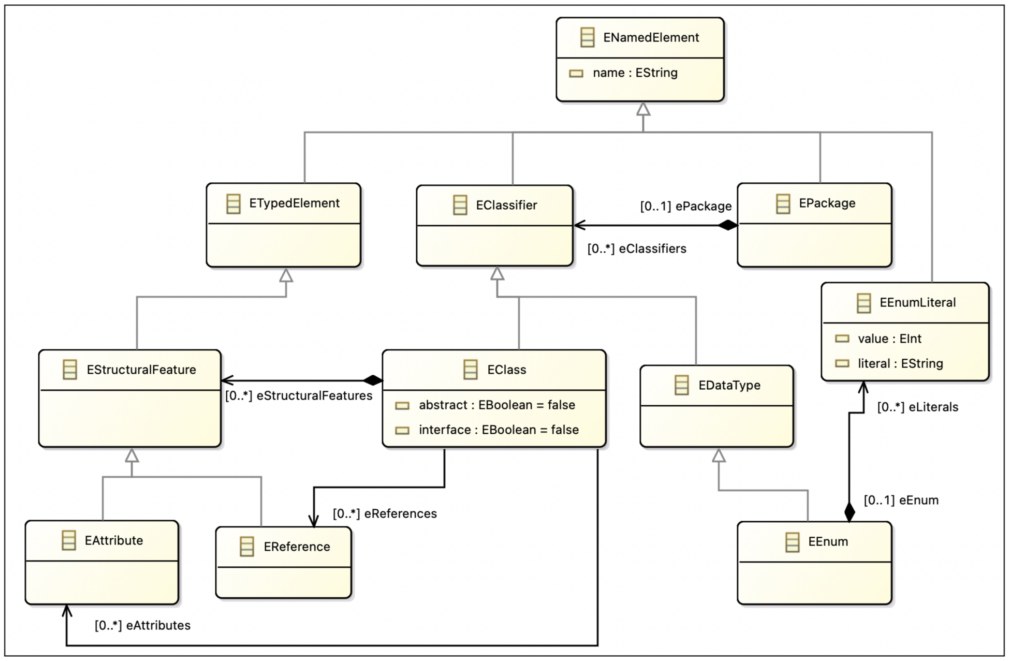


Figure 2.5: Ecore Metametamodel (excerpt)

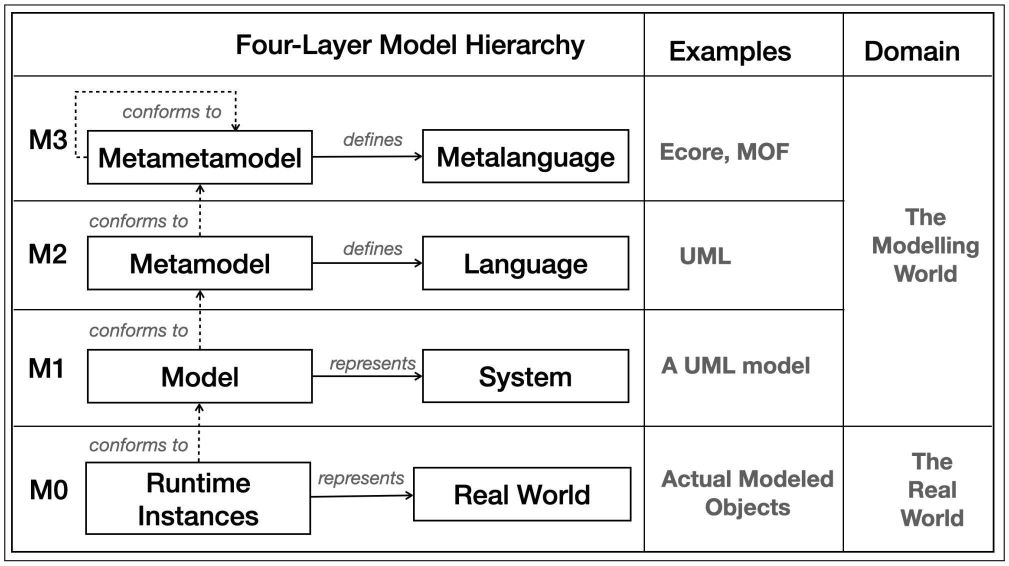


Figure 2.6: MDA Four-Layer Standard Modelling Stack

2.5 Multi-Language Modelling Environment (CORE)

This section presents a notable multi-language modelling paradigm in the software engineering community, Concern-Orientated Reuse (CORE) [44, 45]. A Multi-Language Modelling Environment supports the use of two or more modelling languages to perform all the required tasks to develop a software system. CORE applies the Separation of Concerns principle to facilitate software reuse in a multi-language modelling environment.

2.5.1 Concern-Oriented Reuse

Concern-Oriented Reuse (CORE) is a software reuse paradigm that promotes the use of a unit of modularization called *concern* as its main artefact during software development. A concern is a generic unit of reuse that groups related models (e.g., class diagrams, sequence diagrams, state machines) that cut across a software application [44]. These related artefacts, similar to what happens in Software Product Line development [5], deal with the commonalities and variabilities of the problem and solution space that the concern focuses on. Examples of concerns are *Authentication*, *Authorization*, and *Logging*, the realizations of which are often scattered and tangled in the models and implementation artefacts of a given system. During software reuse, different concern models are *woven/composed* to produce a specific artefact required for a given context or domain.

Each concern provides three interfaces to facilitate reuse: the variation, customization, and usage interfaces (VCU) [44].

Variation Interface:

The *variation interface* specifies the variabilities and commonalities in a family of software artefacts that a concern provides and the impact of each selection on system qualities. This helps a software designer to tailor a given concern to her specific needs via selection of different artifacts.

Customization Interface:

The *customization interface* allows adapting a chosen variation of the concern to a specific reuse context. For example, a concept *Authenticatable*, in an Authentication concern is adapted or customized as a *User* in a bank application.

Usage Interface:

The *usage interface* defines how a customized concern may eventually be used [45]. It specifies the design structure (structural language), behaviour (behavioural language), and intentions (intentional language) that the concern provides to the reusing application. For example, in a class diagram, the usage interface is the set of all public class properties, i.e., the attributes and the operations that are visible and accessible from the rest of the application.

The current mechanism in CORE combines different individual languages, e.g., class diagram, sequence diagram, state machine, by combining their metamodels. As a result, the combined languages have one metamodel to enforce consistency and coordination between different models that are instances of the individual languages. Although the reuse paradigm of CORE is very powerful, it currently requires a lot of work to add another language to the existing ones, because this requires modification of the one metamodel to include the new language. This has several drawbacks. First, modifications to the one metamodel could break backwards compatibility with already existing models. Second, updating the metamodel requires thinking about interactions and consistency constraints between the new language and *all* other languages, which is very tedious and error-prone work. This limitation (among others) motivates this PhD research work. Hence, *PML* aims to fill the gap by providing a robust mechanism for adding independent languages to CORE. Furthermore, *PML* aims to facilitate coordination and consistency management between different models that are instances of different languages.

2.5.2 CORE Metamodel (*excerpt*)

The basic structure of a concern is shown in the excerpt of the CORE metamodel in Figure 2.7. A concern (`COREConcern`) groups related models (`COREModel`) together, with at least one model by default. A `COREModel` is an abstract class that is defined concretely through subclassing. Several models including class diagram, sequence diagram, and state machine can be corified, i.e., making them to behave like concern models (`COREModel`). A concern is composed of one feature model (*COREFeatureModel*), which is a subclass of the `COREModel`. A feature model groups the commonality and variations of a family of software products such as authentication, logging, and authorization. Each software artefact in a family is represented as a *COREFeature*. A feature is realized by one or more instances of the `COREModel`; conversely, an instance of a `COREModel` can realize one or more features. The classes with a pink background in Figure 2.7 depict the structure through which concern models are reused.

A concern can reuse another concern as indicated by `COREReuse`. A `COREConcern` contains a set of `COREReuse` (*reuses*) and each `COREReuse` references another concern (*reusedConcern*). The reference from `COREConcern`, through `COREReuse`, to `COREConcern`

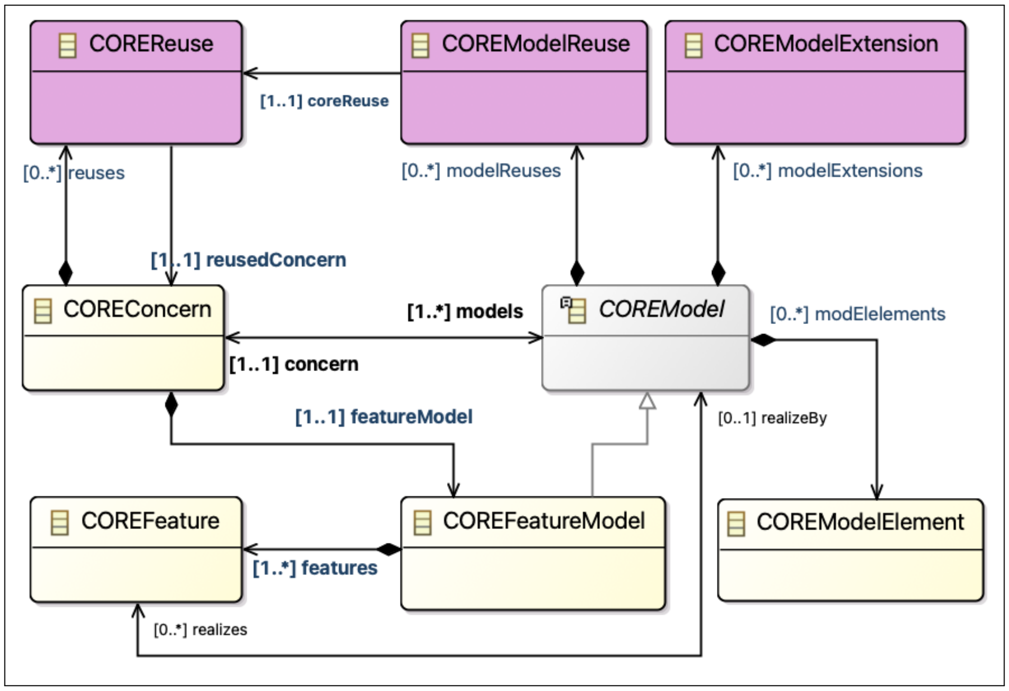


Figure 2.7: Basic Structure of a Concern

represents a cross-boundary relationship between two concerns, the *reusing* and the *reused* concerns. In addition, a concern model can extend another concern model via the metaclass, `COREModelExtension`.

With this mechanism, CORE models, which could be implemented via subclassing of `COREModel`, e.g., class diagram, will have the essential features of CORE including the variation, customization, and usage interfaces (not shown in the metamodel for simplicity) and can easily be reused or reuse other models. This CORE structure has some limitations that also motivate this doctoral research.

- **Single Metamodel (P1)**: A metamodel, which represents all languages supported in `CORE`, is a subclass of `COREModel`. The aim of using a single metamodel, i.e., single underlying model, is to facilitate consistency as well as coordination between instances of the languages. Although the subclassing approach is required to *corify* the languages, it is more challenging to evolve the metamodel because the developer has to face all the complexities of the tightly coupled single metamodel. Also, adding and removing of languages, as well as maintaining the consistencies, will be extremely complicated, as these processes have to be implemented manually. Moreover, modifications to the one metamodel could break backwards compatibility with already existing models.

- **Language Models Contain the *CORE* Reuses (P2)**: *CORE* provides the framework to facilitate the reuse of software artefacts. However, instances of such reuses are contained in the instances of the supported languages; hence, complicating the idea of separation of concerns. Since *CORE* provides the reuse mechanism, the instances of the reuses should be maintained in the *COREConcern*, instead of language models such as class diagrams and state machines.
- **Tightly Coupled *CORE* Metamodel and Language Metamodel (P3)**: The language metamodel is tightly coupled with the *CORE* metamodel by subclassing the `COREModel` metaclass. One of the biggest drawbacks of this is that in order to codify a language, one has to modify its metamodel (so that it subclasses COREModel, for example). Metamodel modifications are undesirable, because all existing models and tools become incompatible.

To promote *modularity* as well as separation of concerns, the current way of dealing with languages in *CORE* had to be improved to support a modular multi-language modelling environment, which led to the creation of the *PML* framework. Although these CORE challenges motivate our research work, our approach can be applied in other multi-language modelling environments that are based on metamodels.

2.5.3 TouchCORE

TouchCORE is a multitouch-enabled software tool used for software design modelling [16]. Its main objectives include, but are not limited to, developing flexible and reusable software design models [46]. It supports the three interfaces of concerns: variation, customization, and usage interfaces. TouchCORE also provides the engine for combining different software models – so-called *weaving* [46, 47] – during the reuse process. All proof-of-concept implementations for the research work presented in this thesis are carried out within the TouchCORE tool.

2.6 Summary

This chapter presents the background concepts that are used for the rest of this thesis. First, we introduce Model-Driven Engineering (MDE), a software engineering methodology that advocates the use of models for all software development tasks. A software model improves reasoning and understanding of the system under development. Furthermore, it facilitates communication between software developers and other stakeholders. Also, software models are used to automate several processes during software development, including the automatic generation of source code and other implementations.

To express models of a system, software engineers create languages, which can be either general-purpose languages (GPL) or domain-specific languages (DSL). A GPL is a software language that targets much larger domains and hence provides a potentially suboptimal way of expressing a specific problem at hand. On the other hand, a DSL is a language whose expressiveness and notations are tailored to a particular domain of application. It provides a more specific and efficient solution in a given domain. A language plays an important role in software engineering because it outlines the infrastructure for creating models.

The creation of models as well as building of software languages is a difficult task. The task becomes even more difficult in multi-language environments when the underlying technologies or languages are combined together as within a single software artefact or metamodel. In this case, it is more challenging to evolve the metamodel of the combined languages because the software engineer has to face all the complexities of a tightly coupled single metamodel. We present a notable multi-language modelling environment (Concern-Oriented Reuse), which supports different tightly coupled modelling languages. We present the challenges of this architecture and then explain how it motivates this research work.

The next chapter provides further motivation for this research work with some real-life perspective examples. We demonstrate some of the key features of a perspective, including perspective actions and mappings between different language elements, as well as between model elements.

Chapter 3

Motivation

In this section, we first motivate *perspectives* by using a single language and then with multiple languages, each with a simple example. We demonstrate the features of *perspective* including *perspective actions* and potential *mappings* (i.e., links) between different language elements. We target software languages that are defined using a metamodel-based approach. Our assumption is that each modelling language (i.e., its metamodel and its language actions) exists independently, without prior links or constraint conditions with other languages.

Definition 3.0.1 (Perspective). A *perspective* combines different languages for a modelling purpose; defines the role of each participating language as well as composite actions for building a consistent multi-model system and maintaining the links between different model elements.

Definition 3.0.2 (Software Language). A software language is a software system that provides an infrastructure for building other software systems, which conforms to the language in question. A software language comprises three main building blocks:

abstract syntax (e.g., metamodel), concrete syntax, and semantics.

Definition 3.0.3 (**Language Element**)**.** A language element is a concept that participates in the definition of the abstract syntax of a software language. In a metamodel-based abstract syntax, metaclasses and attributes of metaclasses are examples of language elements.

Definition 3.0.4 (**Language Action**)**.** A language action is a function in the API of the language that provides editing steps for constructing models that are instances of the concerned language metamodel.

In each *perspective*, the metamodels of different languages are combined with mappings between the language elements to produce a modular underlying model where a module represents a participating language.

Example 3.0.1. *ITU goal and scenario models and UML class diagrams may be combined to describe a system by establishing mappings between actors in goal models and classes, as well as steps in scenario models and operations.*

This modular combination aims to reduce the complexity associated with a single underlying model during software evolution. It also means that the language actions of each metamodel have to be manipulated to provide the most appropriate actions for building and then maintaining the relationships between different language elements as specified by the mappings in the *perspective*. For example, if a language action creates a new actor in a goal model, it now also needs to create a class in a class diagram given the mapping between these two language elements. A *perspective* can redefine a language action or simply re-expose or hide it. Furthermore, it can add new actions that cannot be achieved with existing language actions. Collectively, these actions are referred to as *perspective actions*, which comprise the individual actions of each language, e.g., create class, add operation, or modify parameter, and the new actions, e.g., map x to y, where x and y refer to different language elements.

Definition 3.0.5 (Perspective Action). A perspective action re-exposes or augments an existing language action to enforce consistency rules defined in the perspective mappings.

All perspective actions are used to construct a model instance conforming to the languages in the *perspective*. However, a perspective action proactively prevents an inconsistency, instead of fixing a broken consistency. Hence, *PML* fosters proactive actions, i.e., actions that prevent inconsistencies from occurring.

3.1 Single-Language Perspective

A software language is used to capture the characteristics of a software system. Software systems exhibit different features that include, but are not limited to, structural, behavioural, and intentional characteristics. These characteristics often spread across different levels of abstraction; e.g., a class diagram language can be used to specify the domain model of a system during the requirement stage. A class diagram can also be used to describe the complete system structure at the design stage. Moreover, a class diagram may be used to define the metamodel of a language based on MOF. For each of these uses, different language features of class diagrams are applicable. Hence, the level of the abstraction determines the required construction semantics and view of a language.

Definition 3.1.1 (**Construction Semantics**)**.** Construction semantics define the editing steps for building and maintaining models that conform to the concerned language.

In this section, we illustrate different use cases of a class diagram language, each use case representing a single language *perspective*. Each *perspective* that reuses a class diagram for a modelling purpose, e.g., domain modelling, design modelling, or metamodelling purpose, augments the construction semantics and views of the class diagram to reflect the corresponding purpose of the language in the perspective.

Domain Model Perspective

A domain model is a class diagram, which describes a system's structural conceptual relationships. It uses classes, attributes, associations, compositions, and generalisation to describe the entities of the system. An example is shown in Figure 3.1, which depicts the structural relationships that exist for a simple bank account.

The model view depicts classes and their attributes, as well as relationships between pairs of classes. Right clicking on a class reveals the supported language actions in a domain model, e.g., create attribute or delete class, as shown in Figure 3.1. Since a domain model is a class diagram, these actions are *re-exposed* from the class diagram language. Similarly, a modeller can specify a relationship between a pair of classes by drawing a line from one to the other and selecting the desired relationship. The corresponding language actions are also *re-exposed* from the class diagram language. Operations are not shown, because a domain

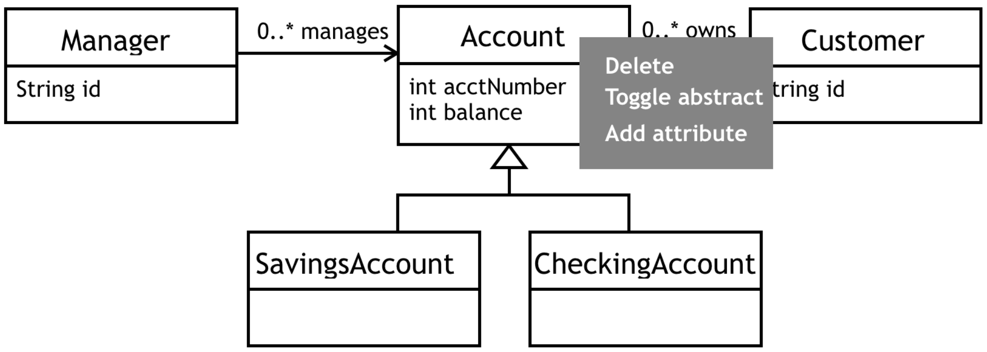


Figure 3.1: Bank Domain Model with Supported Actions

model does not support operations. Hence, the language actions associated with operations such as create/delete operation and update operation parameter are *hidden* in the domain model perspective.

Design Model Perspective

Another single-language perspective is a class diagram that is used to describe a complete system structure. In this case, a design model is composed of all design classes including their attributes, relationships, and operations.

The design model perspective supports a view that provides the language actions that are required for the specification and maintenance of a design model. Unlike the domain model perspective, the design model perspective allows a modeller to add or delete operations and other related design model actions, as illustrated in Figure 3.2. In this case, all language actions of the class diagram language are supported and hence *re-exposed* by the perspective.

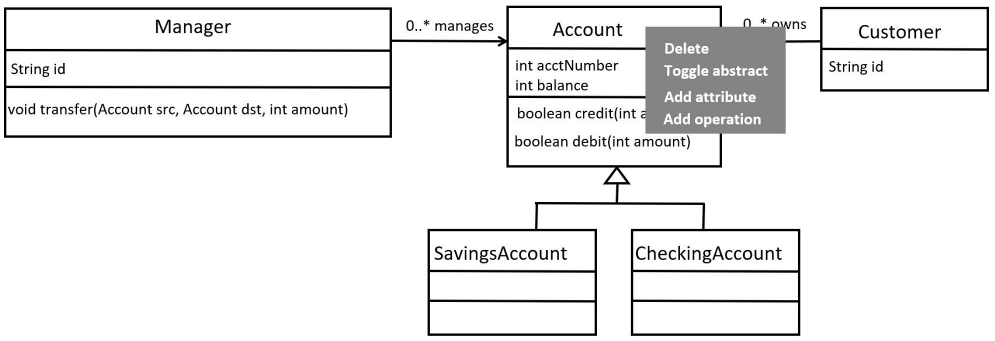


Figure 3.2: Bank Design Model with Supported Actions

3.1.1 Metamodelling Perspective

A perspective can also exist at the metamodelling level. For example, the Meta Object Facility (MOF) is a metamodelling language which is used to define modelling languages. MOF is a single-language perspective, which provides key concepts, such as types (e.g., classes and enumerations), attributes, operations, generalisation, and associations, for the definition of languages. These concepts also exist in the class diagram language. However, MOF restricts some concepts from class diagrams, which include association classes, interfaces, n-ary associations, and dependencies. Thus, the metamodelling perspective derives the MOF language from the class diagram language by *hiding* the language actions that are associated with the restricted concepts and *re-exposing* all others.

3.2 Multi-Language Perspective

In this section, we illustrate perspectives that use more than one language. In a multi-language perspective [48], two or more languages are combined for a given purpose. To support the co-evolution of models conforming to different languages, several mappings, consistency constraints, and perspective actions are specified between different language elements. A mapping captures a relationship between two language elements, each from a separate language. Furthermore, the mappings are supported with consistency constraints to ensure that the values of the mapped elements are always consistent, especially at modelling-time.

Furthermore, perspective actions are defined that are used by the modeller instead of the original language actions whenever changes to the models are made. In addition to the *re-expose* and *hidden* actions used for single-language perspectives, existing language actions now may have to be *redefined* to suit the aims of the perspective. A perspective action may prevent an inconsistency from occurring or may be triggered to fix broken consistencies which may arise from an update or other activities that change a value of either mapped language element.

Consider that a modeller is working on a software design for a bank application. First, the modeller decides to use a class diagram language to capture the design structure of the system, as shown in Figure 3.2. Also, the modeller defines the operational behaviour of the system with a sequence diagram language. For example, Figure 3.3 shows the behaviour of the *debit* operation defined in the class diagram model with a sequence diagram. Since these two languages are used to elaborate a single system, it is essential that these models are consistent with each other. The consistency requirements of the bank system are:

- **C1:** Each *sequence diagram model* must describe the behaviour of a corresponding *operation* in the class diagram model.
- **C2:** Each *lifeline* in the sequence diagram model must be typed by a *class* in the class diagram model.
- **C3:** The behaviour of each *operation* in the class diagram model can be described with

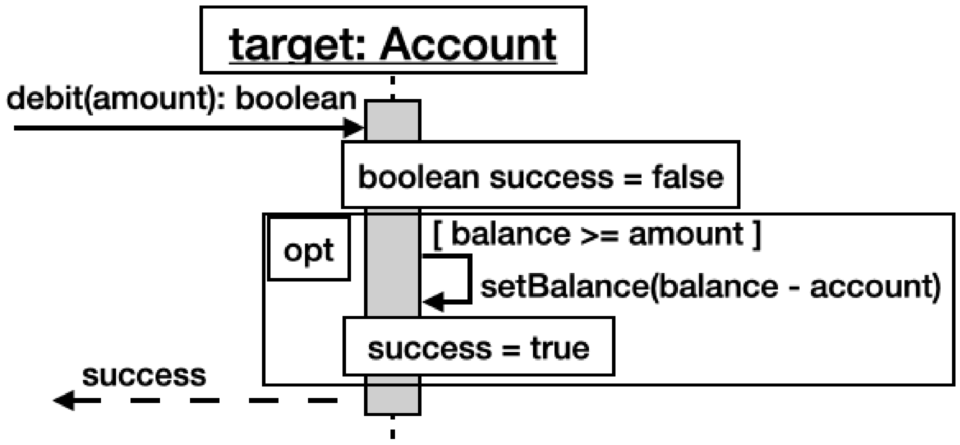


Figure 3.3: Debit Operation Sequence Diagram

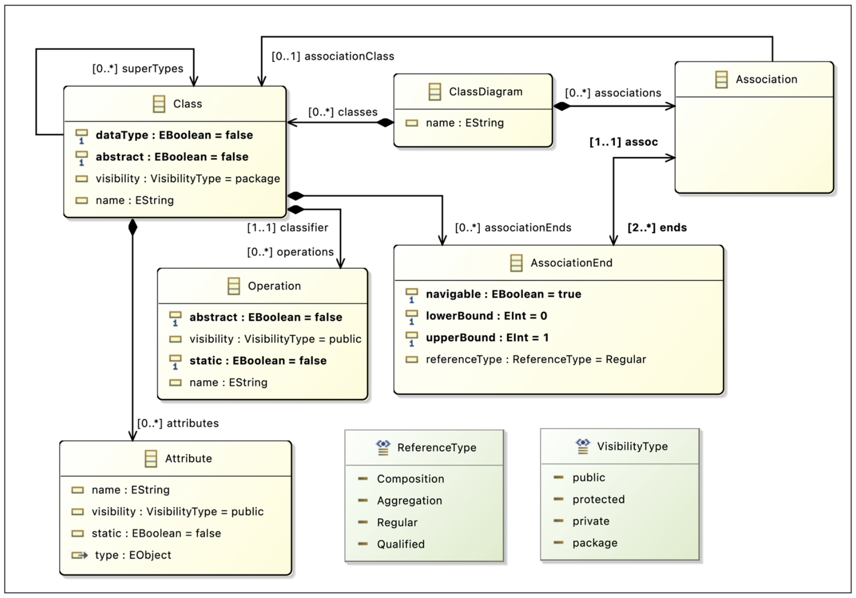
a corresponding *sequence diagram model*.

- **C4:** Each *class* in the class diagram can be a type of several *lifelines* in the sequence diagram model.

Ensuring consistency is challenging especially when the modeller has to deal with it manually. In *PML*, we provide a *perspective* that groups together different modelling languages for a specific modelling purpose, and then ensures that the desired consistencies are automatically maintained across the models built with the languages.

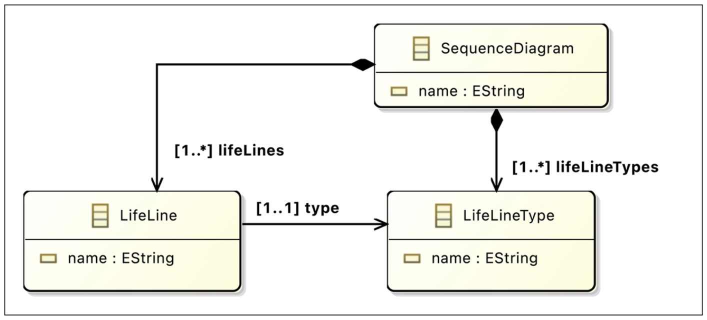
Class Diagram and Sequence Diagram Perspective

A typical example of a multi-language perspective that would be useful for our bank design example is the *Software Design Perspective*, i.e., a *perspective* which combines a class diagram language and a sequence diagram language, assuming that both languages have been defined independently. Figures 3.4 and 3.5 show the metamodels of the class diagram language and



```
public void createClass(ClassDiagram owner, String name, boolean dataType);

public void updateClassName(Class class, EAttribute nameFeature, String newValue);

public void deleteClass(Class class);
...
```

Figure 3.4: Class Diagram Metamodel (excerpt) and Some Language Actions



```
public void createLifeLine(SequenceDiagram owner, String name);

public void updateLifeLineName(LifeLine lifeLine, EAttribute nameFeature, String newValue) ;

public void deleteLifeLine(LifeLine lifeLine);
...
```

Figure 3.5: Sequence Diagram Metamodel (excerpt) and Some Language Actions

the sequence diagram language, as well as their language actions, respectively[1]. Hence, for both languages to collaborate as a perspective, the mappings and consistency conditions between *class* and *lifeline* as well as *operation* and *sequence diagram* need to be established, as outlined above.

Furthermore, the perspective provides actions to support the collaboration of the two languages. Some of the original language actions are *re-exposed*, e.g., *create attribute* in a class diagram and *create statement* in a sequence diagram. Such actions only affect model elements of one language, and do not influence the model elements of the other language. On the other hand, an action that affects a model element with a consistency mapping may influence other linked language elements, and hence has to be *redefined*.

Example 3.2.1. *Actions that need to be redefined for our perspective would include* create lifeline*,* create message *in a sequence diagram, and* create operation *in a class diagram).*

In an independent sequence diagram, i.e., a sequence diagram which is not linked with any other language, creating a lifeline or message requires only the basic actions of a sequence diagram. However, in a *Software Design Perspective*, a lifeline that is not linked to a class constitutes a violation of the mapping between the two model elements and its associated constraint condition. Proactively, this violation can be avoided by redefining the *create lifeline* action to include mapping the new lifeline to an existing class or creating a new class and linking it with the lifeline. Thus, the *redefine* action ensures one of the consistency objectives of the perspective, i.e., a lifeline must represent an object of an existing class in

[1]Although the official UML diagram language has one metamodel for all the UML diagrams, we keep the metamodel of each UML diagram separate to promote modularity as well as the evolution of multi-language software systems. Hence, we treat each UML diagram as a separate modelling language

the associated class diagram (C2). Another example of a *redefine* action is updating the name of an operation in a class diagram. In this case, the action also updates the name in the mapped sequence diagram (C1).

A multi-language perspective may also offer new actions not supported by any existing language action, e.g., an action that creates a mapping between two model elements. A modeller may want to change the class linked to an existing lifeline. In this case, the *mapping* action allows the modeller to update the mapping information of the lifeline, i.e., by replacing the linked class with another class from the class diagram.

These examples of single-language and multi-language perspectives demonstrate how our proposed PML framework aims to manipulate existing languages to provide language designers or modellers with relevant views and language actions.

3.3 Roles in a Perspective

In this section, we introduce other categories of perspectives: *perspective* with single-role and *perspective* with multi-role.

3.3.1 Perspective with Single-Role

A software language can be used at different levels of abstraction to *specify* or *describe* the characteristics of a software system. A class diagram language, for example, can be used to specify the domain model of a system during the requirement stage and also to describe the complete system structure at the design stage, and as well, to create metamodels (in the form of MOF) at metamodelling level. For each of these uses (i.e., class diagram language roles), different language features of class diagrams are applicable. A perspective with single-role reuses exactly one language at a particular level of abstraction, where the reused language plays a specific role.

Example 3.3.1. *Perspectives with single-role include* domain model perspective*,* design model perspective*, and* metamodelling perspective *(see Section 3.1).*

Every perspective with single-role is a single-language perspective, but the reverse is not the case.

3.3.2 Perspective with Multi-Role

In a perspective with multi-role, two or more language roles are combined for a given purpose. These roles can emanate from a single language, i.e., the language plays multiple roles in the perspective. Alternatively, different languages can play different roles in a perspective with multi-role. In addition to the *re-expose* and *hidden* actions used for perspectives with single-role, existing language actions now may have to be *redefined* to suit the aims of the perspective, i.e., enforce relationships between language roles in perspectives. To support the co-evolution of models based on different roles, several mappings, consistency constraints, and perspective actions are specified between language elements.

Every multi-language perspective is a perspective with multi-role, but the reverse is not the case. A typical example of a multi-role perspective is the combination of a class diagram language (structural role) and a sequence diagram language (behaviour role). Hence, for both languages to collaborate as a perspective, the necessary relationships between language elements, e.g., a mapping between a Class metaclass and a Lifeline metaclass, and constraint conditions need to be established.

Another example of a multi-role perspective is domain and design model perspective, i.e., a *perspective* that reuses a class diagram (a single-language perspective) for both domain modelling (domain role) as well as design modelling (design role). This type of perspective often implements equivalency constraints between elements across models from different language roles. For example, a class in a domain model is required to be mapped with at least one implementation class in the design model; whereas each design class needs to be mapped with exactly one domain class. This equivalency constraint does not enforce consistent information, e.g., that the names of the domain model class and mapped design model class are the same. However, the equivalency constraint ensures that these mappings exist, which helps to ascertain when the domain model is fully implemented in the form of a design model.

3.4 Summary

This section motivates perspective with some (and simple) perspectives. We show how a perspective reuses a single language (class diagram language) for a domain modelling, design modelling, and metamodelling purposes. A single-language perspective can *re-expose*, as well as *hide* some language actions to support only the required language actions based on the role of the language. Also, we present a multi-language perspective that combines a class diagram and sequence diagram languages for requirement elicitation purpose. In addition to *re-expose* and *hidden* actions, a multi-language perspective *redefines* language actions to suit the aims of the perspective. Finally, we introduce other features of a perspective, which include perspectives with single-role and perspectives with multi-role.

Following the motivation of *perspectives* by using a single language and multiple languages, the next chapter presents the general overview of the *PML* framework. We discuss the *PML* architecture, workflow, and then its metamodel.

Chapter 4

PML Framework

This chapter presents the general overview of the **P**erspectives for **M**ulti-**L**anguage Systems (PML) framework. First, we discuss the principal modelling levels in the *PML* framework. Second, we present the *PML* workflow and then its metamodel. We further illustrate how we apply the *PML* framework to address some of the multi-language challenges outlined in Chapter 2.

4.1 Levels in the *PML* Framework

To *specify*, *prescribe*, *describe*, and *develop* software systems using several languages, the *PML* approach incorporates three levels, the *language level*, the *perspective level*, and the *model level*, as shown in Figure 4.1.

4.1.1 Language Level

The *language level* contains the collection of different modelling languages (i.e., L_1, L_2, L_3) that are being used in a software system. At this level, the framework focuses on two main

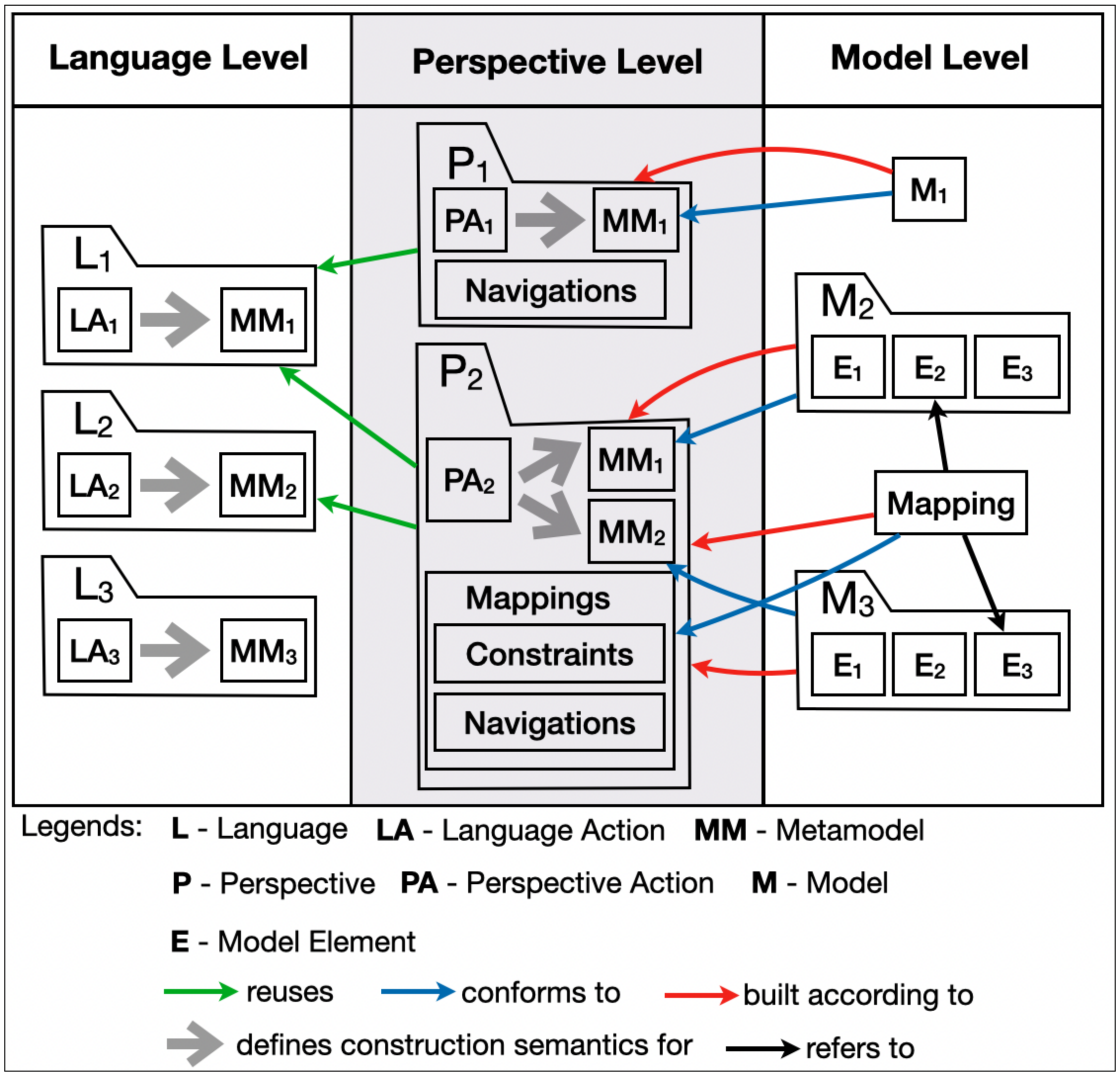
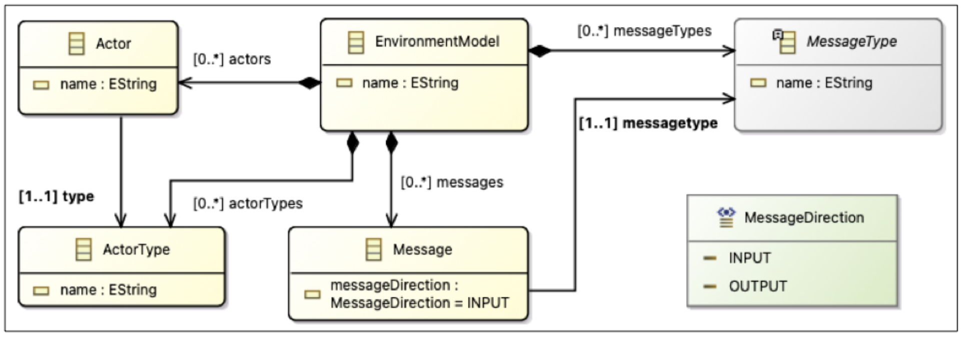
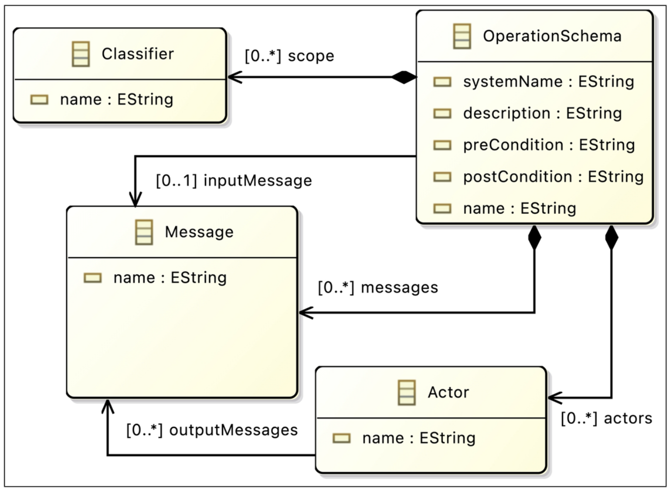


Figure 4.1: Generic Architecture of *PML*



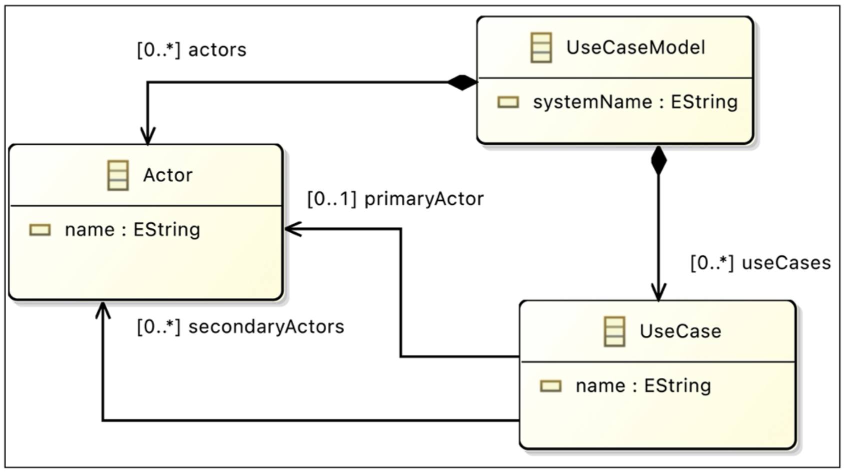
```
public void createActor(EnvironmentModel owner, String name);

public void updateActorName(Actor actor, EAttribute nameFeature, String newValue);

public void deleteActor(Actor actor) ;
...
```

Figure 4.2: Environment Model Metamodel (excerpt) and Some Language Actions



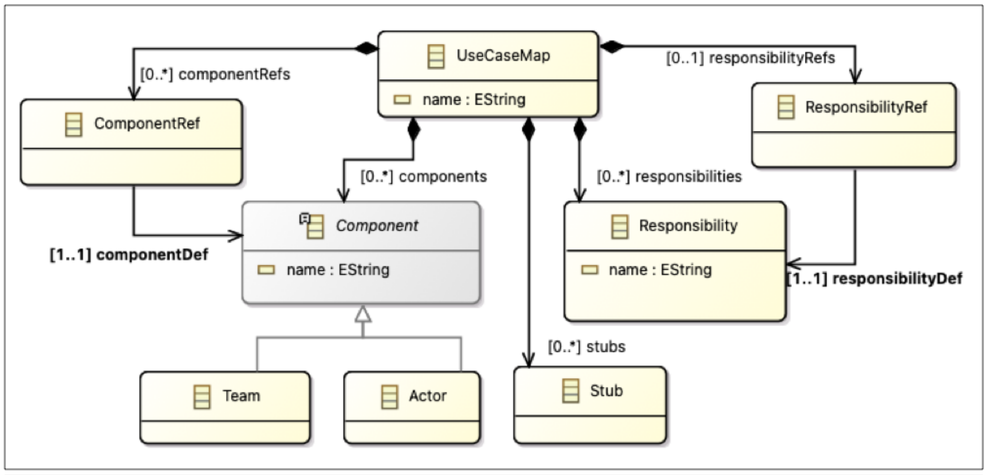
```
public void createMessage(OperationSchema owner, String name);

public void updateMessageName(Message message, EAttribute nameFeature, String newValue)

public void deleteMessage(Message message);
...
```

Figure 4.3: Operation Model Metamodel (excerpt) and Some Language Actions



```
public void createActor(UseCaseModel owner, String name);
public void updateActorName(Actor actor, EAttribute nameFeature,
    String newValue);
public void deleteActor(Actor actor);
...
```

Figure 4.4: Use Case Model Metamodel (excerpt) and Some Language Actions



```
public void createResponsibility(UseCaseMap owner, String name);
public void updateResponsibilityName(Responsibility responsibility,
    EAttribute nameFeature, String newValue);
public void deleteResponsibility(Responsibility responsibility) ;
 ...
```

Figure 4.5: Use Case Map Metamodel (excerpt) and Some Language Actions

building blocks of a modelling language: the abstract syntax or *metamodel* (e.g., MM_1 for L_1) and the *language actions* (e.g., LA_1 for L_1).

The language actions define the construction semantics for the metamodel. These actions encapsulate complete editing steps that are used by the modeller when elaborating a model using the language. Internally, the actions can create, read, update, and delete instances of the language metaclasses. Language actions are at a higher abstraction level than CRUD operations, and one language action may in fact perform several CRUD operations.

Example 4.1.1. *In a class diagram language, creating an instance of the metaclass operation and then adding the instance to the list of operations of a class is a single language action (*create operation*) shown in Figure 3.4.*

Hence, the language actions constitute the *API* for building models with the language.

As shown in Figure 4.1, there is no direct link between any of the languages used in *PML*, i.e., the languages are independent. Figures 3.4 to 3.5 and Figures 4.2 to 4.5 show a collection of six different independent language metamodel excerpts. For each language, a subset of the offered language actions are also listed. Most of the examples in the rest of the thesis are based on these languages.

4.1.2 Perspective Level

The *perspective level* defines *perspectives*, which reuse and group one or several *languages* for a specific modelling purpose. In a perspective, a language plays one or more *roles*, which collectively address the purpose of the perspective. A perspective also holds the conceptual relationships between language elements of the involved languages, which contains the consistency conditions as well as the navigation mechanisms that need to be established and then maintained between the languages to foster their collaboration. Consistency conditions discussed in this work are equivalency, equality, and multiplicity constraints. The equivalency, equality, and multiplicity constraints are explained in more detail below.

Definition 4.1.1 (Consistency). Consistency establishes a correspondence between a pair of model elements based on equivalency, equality, or multiplicity constraints.

Definition 4.1.2 (Equivalency Constraint). An equivalency constraint dictates an existence of a correspondence between a pair of model elements that refer to the same entity or concept in the software under development.

Definition 4.1.3 (Equality Constraint). An equality constraint dictates an existence of a synchronized value between a pair of attributes, each from a pair of corresponding model elements.

Definition 4.1.4 (Multiplicity Constraint). A multiplicity constraint dictates the number of allowable instances of a language element that can correspond to an instance of a related language element.

A single-language perspective [48] (P_1 in Figure 4.1) reuses exactly one language and then defines perspective actions that reuse the actions of the reused language. Such a perspective allows a language to be tailored to a specific purpose.

Example 4.1.2. *P_1 could reuse a class diagram language (L_1 with MM_1) to allow a modeller to build domain models.*

In this case, the modeller does not need to be able to create operations. To this aim, a *domain modelling perspective* can exclude the *create operation* language action from the actions available to the modeller.

In addition, a single-language perspective provides a navigation mechanism that allows a modeller to quickly view the structural relationships of the elements in a model. This navigation mechanism is handy, especially when the modeller is working with a large number of elements.

Example 4.1.3. *In a class diagram model, a modeller can use the navigation feature to list all the classes that are contained in the model and then navigate to his desired class.*

Locating such a class would be more challenging, without the navigation, because the modeller would need to scan the whole model, which might be a daunting task, especially with a very large model.

On the other hand, in a multi-language perspective [48], two or more languages are combined for a given purpose. Hence, a multi-language perspective (P_2 in Figure 4.1) defines not only perspective actions, but also relationships between language elements, which include the consistency conditions and the navigation mechanisms across language boundaries.

Example 4.1.4. *P_2 could be a multi-language perspective that combines a class diagram language (L_1 with MM_1) and sequence diagram language (L_2 with MM_2), the purpose of which is to build design models specifying both design structure and design behaviour.*

In this case, a multiplicity constraint that should be enforced by the perspective could be that a public operation in the class diagram may be mapped to a sequence diagram that specifies the operation's behaviour. To this aim, a *Language Element Mapping* (LEM) with an *Optional* (0..1) cardinality would be specified in the perspective that keeps track of which operation is associated with which sequence diagram. Similarly, an inter-language navigation mapping can be specified in the perspective that establishes a link between mapped operation and sequence diagram.

Similar to language actions, the *perspective actions* define the construction semantics for models elaborated using the perspective. These perspective actions can:

1. *re-expose language actions*, i.e., offer (reuse) language actions as perspective actions without any modification,

2. *redefine language actions*, i.e., augment existing language actions to suit the purpose of the perspective, or

3. *define new perspective actions* that perform tasks not specific to any of the involved languages, e.g., actions that establish links between model elements from different models.

4.1.3 Model Level

The *model level* shows models conforming to different languages and built according to a perspective.

Example 4.1.5. *In a multi-language perspective that combines a class diagram language and sequence diagram language (see Figures 3.2 and 3.3), Figure 3.2 (M_2 in Figure 4.1) is a model built according to the perspective (P_2) and an instance of the class diagram language metamodel (Figure 3.4, i.e., MM_1). Similarly, Figure 3.3 (M_3) is a model built according to the perspective (P_2) and an instance of the sequence diagram language metamodel (Figure 3.5, i.e., MM_2).*

Another important feature at this level are the mappings (i.e., links) which establish correspondences between different model elements (i.e., *model element mappings* (MEM)). Figure 4.1 shows a mapping between two model elements, E_2 from M_2, and E_3 from M_3 (e.g., a mapping between a class and a lifeline type, see Figure 3.2 and Figure 3.3). Creating a mapping is a new perspective action, because this edit operation is not directly linked to any action of the involved languages. Each mapping is typed by a *LEM* at the perspective level, which specifies the consistency conditions, such as multiplicity constraints, to ensure the consistent co-evolution of the mapped model elements.

4.2 PML Workflow

The basic workflow of the *PML* process is shown in Figure 4.6. At the language level, the *language designer* defines software languages, and focuses on their building blocks which include the abstract syntax (encoded with a metamodel), the concrete syntax, and semantics, which are encoded in language actions (at least the construction semantics).

At the perspective level, a perspective designer is responsible for the definition of a perspective including the selection of languages as well as the specification of the language element mappings (*LEMs*) and the perspective actions. To create a perspective, a perspective designer, first, decides on the set of model types and languages that are needed for a modelling purpose. Considering the *Software Design Perspective*, the model types are the class diagram design model (e.g., Figure 3.2) and sequence diagram behavioural model (e.g., Figure 3.3). Also, the set of languages are the class diagram language (Figure 3.4) and the sequence diagram language (Figure 3.5). Then, the designer defines a perspective to combine the different languages of those model types for a given purpose, in our case software design. This definition includes the *LEMs* as shown in Figure 4.6 (for example, a language element mapping between the `Operation` metaclass in the class diagram metamodel and the `SequenceDiagram` metaclass in the sequence diagram language, which ensures that each instance of the `SequenceDiagram` metaclass must be mapped to a corresponding instance of the `Operation` metaclass). Furthermore, the perspective designer specifies which language actions are to be re-exposed or redefined as perspective actions. Then, the designer generates the implementation of the perspective including the specified *LEMs* as well as the perspective actions.

At the modelling level, a *modeller* can use a perspective to create sets of models conforming to one or multiple software languages within the purview of a perspective (see

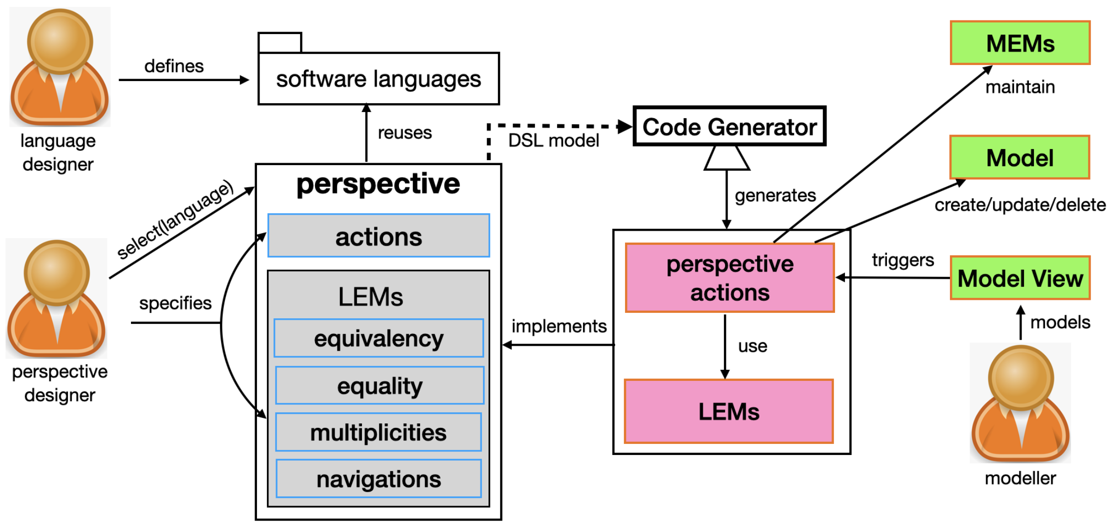


Figure 4.6: Definition and Execution of a Perspective

Figure 4.6). A modeller interacts with the model view, which triggers a perspective action, which in turn calls the respective language action(s) to create, update, or delete the desired model element(s) while ensuring model consistency. To maintain the consistencies between different models, the *perspective actions* use the consistency conditions encoded in the *LEMs* to ensure that the models are always consistent. *Perspective actions* may create or delete *MEMs* (i.e., maintain *MEMs*) while ensuring that the perspective models are consistent.

Example 4.2.1. *In a* Software Design Perspective*, a* perspective action *may replace an existing* MEM *between a lifeline type and a class with a new* MEM *between the lifeline type and a different class.*

4.3 PML Metamodel

This section presents details of the *PML* metamodel, which provides the structure to combine different languages and then specify consistency conditions between different language elements, potentially across language boundaries. First, we discuss how to combine different languages as well as their mappings. Second, we present the perspective actions and explain how they maintain consistency across different language elements. Third and finally, we discuss the advanced features of perspective actions, including *action effects* and *derived parameter*.

4.3.1 Perspective, Language, and Mappings

To combine different languages for a modelling purpose, a perspective reuses different languages, with each language accomplishing a specific role. Hence, the `Perspective` metaclass contains a set of *languages* (`LanguageMap`) which establishes correspondences, each between a language role (i.e., *key* in the `LanguageMap` metaclass) and the corresponding `Language`. A `Language` can either be an existing `Perspective` or an `ExternalLanguage`; hence, a perspective can reuse other perspectives as well as existing languages. In this doctoral research work, we focus on how a perspective reuses/combines existing languages (e.g., class diagram, use cases, and sequence diagram languages), while the reuse of a perspective in another perspective needs to be addressed in future work. A perspective may have a *default* language role which gives the corresponding model a higher priority during navigation, i.e., in a multi-language perspective system [48, 49], a model corresponding to the default language role is presented first. However, the user can navigate to other models from the default model.

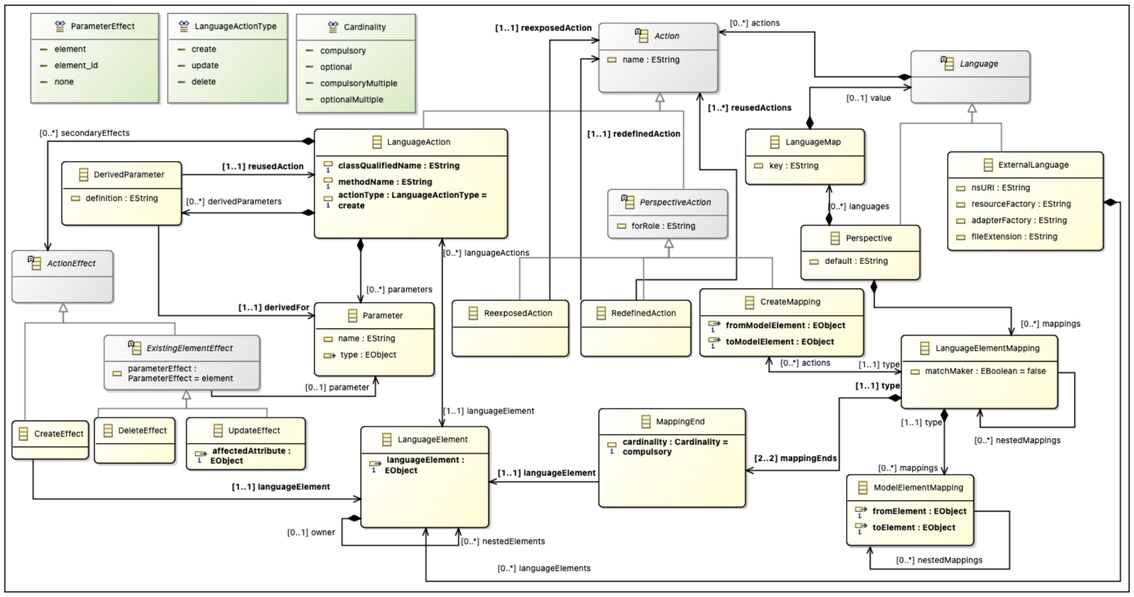


Figure 4.7: *PML* Metamodel (excluding navigation mechanism)

As explained in Section 4.1, each language has its own standalone metamodel. In *PML*, an external language is represented by the metaclass `ExternalLanguage`, which captures the details of the language (*nsURI*, *resourceFactory*, *adapterFactory*, and *fileExtension*) which are used to register the language in the software system. Since the prototype implementation of our work focuses on EMF defined modelling languages, these language information details are pertinent to EMF languages. *nsURI* represents the unique name space identifier for the language metamodel package; *resourceFactory* designates the factory which is associated with serialization as well as deserialization of the language models; *adapterFactory* refers to the factory which provides the needed interfaces and notifications support for the model views; and the *fileExtension* represents the file extension of each language model.

In order to be able to create language element mappings, the *PML* framework must be made aware of the metaclasses in the language whose instances can be mapped. Therefore each language (`ExternalLanguage`) contains a set of language elements

(`LanguageElement`), that each references the corresponding actual language element (e.g., metaclass) in the language metamodel. Furthermore, each `Language` is composed of actions (`Action`), which is either a representation of an external language action (`LanguageAction`) or a perspective action (`PerspectiveAction`). The external language action captures the construction semantics defined for the corresponding language elements (see Figures 3.4 to 4.5 and Figures 4.2 to 4.5 for examples). To *re-expose* or *redefine* an external language action, the perspective needs to know the qualified name (*qualifiedName*) as well as the main name (*methodName*) of the corresponding language action. Also, the `LanguageAction` encodes the language action type, i.e., *create*, *update*, or *delete*, and as well, contains a set of parameters which are required to call the language action.

4.3.2 Perspective Actions

The perspective action (`PerspectiveAction`) manipulates the construction semantics of each language (i.e., language actions) to enforce consistency conditions between different models of the participating languages. The `PerspectiveAction` class represents a set of new actions (i.e., actions that act across language boundaries) and can propagate existing language actions, either modified or re-exposed. The `PerspectiveAction` class has three subclasses: `CreateMapping`, `ReexposedAction`, and `RedefinedAction`.

`CreateMapping` instantiates a `ModelElementMapping` (i.e., *MEM*) between two model elements (*fromElement* and *toElement*) and is typed by a particular `LanguageElementMapping` (i.e., *LEM*). Hence, `CreateMapping` is independent of any `LanguageAction`, because it does not affect the model of the respective languages. `ReexposedAction`, on the other hand, simply exposes a language action that is supported by the perspective without any change.

`RedefinedAction`, however, augments the construction semantic of an action (*redefinedAction*) to comply with the specifications of the perspective. Furthermore, `RedefinedAction` reuses other language actions (*reusedActions*) to effect the desired consistency conditions being specified in the concerned *LEM*. For example, a *perspective* designer may specify that creating a class in a class diagram model optionally requires the creation of an actor in a use case model and the establishment of a *MEM* between the two model elements. The language action (*create class*) is then being redefined by the perspective to ensure that the multiplicity constraints as well as the equivalency constraint between the class and the actor are maintained, while reusing the *create actor* language action.

To foster collaboration between different language elements, e.g., between `Operation` in a class diagram language and `SequenceDiagram` in a sequence diagram language, a `Perspective` groups a set of language element mappings (`LanguageElementMapping`), which comprises a pair of mapping-ends (`MappingEnd`), i.e., *from* and *to* mapping-ends. Each `MappingEnd` defines the multiplicity, i,e., `Cardinality`, of the corresponding language element. We cover four categories of the cardinality: *Compulsory* (1..1), *Optional* (0..1), *Optional-Multiple* (0..*), and *Compulsory-Multiple* (1..*). Furthermore, each `MappingEnd` refers to the `LanguageElement` which references the actual language element in the language metamodel. Hence, in each *language element mapping*, each respective mapping-end specifies the minimum and maximum allowable instances of the language element which can be mapped.

Consider a *language element mapping* between a `Class` metaclass (cardinality of 1) from a class diagram language and a `LifeLineType` (cardinality of 0..*) from a sequence diagram language. This implies that a single instance of the `LifeLineType` cannot be mapped to more than one instance of the `Class` metaclass. This multiplicity constraint also dictates that a `LifeLineType` instance cannot exist without being mapped with a `Class` instance. Two model elements (*fromElement* and *toElement*; e.g., a `Class` instance and a `LifeLineType` instance) are required to create a *MEM* based on a given instance of the `LanguageElementMapping` (i.e., the *MEM type*).

To maintain consistencies of properties between two mapped model elements, we introduce the *nested mapping*, i.e., synchronized mapping, (`LanguageElementMapping`) which synchronizes a pair of language element attributes, each from a different language element; e.g., the *name* attribute in a `Class` metaclass from a class diagram language and the *name* attribute in an `Actor` metaclass from a use case model language. If such a synchronized mapping between the *name* attributes is nested within the mapping between the `Class` and `Actor`, then our framework ensures an equality constraint (i.e., the values of the respective attributes are always kept in sync). The flag, *matchMaker*, designates the properties of language elements that are used to automatically determine model elements that should be matched for mappings.

Example 4.3.1. *When the* matchMaker *of the nested mapping between* name *attributes is set, then this means that the class name can be used to determine a corresponding actor, i.e., an actor with the same name, that should be mapped to the class.*

In this work, we focus on synchronization of related attributes, instead of model element matching that requires complex constraints. This attribute synchronization approach allows perspective designers to capture constraints with our DSLs and then automatically generate the implementation of the *perspective*. This generative approach frees a perspective designer from the manual implementation, which is tedious and error-prone. In addition, we are yet to observe a need for such complex constraints across languages in a multi-language systems.

While *nestedMappings* are used to enforce equality constraints between properties, the parent mapping of nesting mappings enforces equivalency constraints between mapped model elements (e.g., a `Class` and an `Actor` are equivalent, because they refer to the same entity). On the other hand, the *cardinality* in each `MappingEnd` ensures the multiplicity constraints between mapped model elements, i.e., to how many model elements a given model element can or must be mapped. The multiplicity, equivalency, and equality constraints, i.e., the consistency conditions, are taken into account in the perspective actions to proactively prevent inconsistencies between mapped model elements in different models.

4.3.3 Action Effects

A simple language action involves only one language element with one or more *language element mappings*. For instance, in a *class diagram, use case diagram, sequence diagram* perspective, the *redefined* perspective action, *create class*, which is based on a *LEM* between the `Class` metaclass and `Actor` metaclass, creates a class in the class diagram model for a corresponding actor in the use case diagram using existing language actions. Furthermore, the actor is mapped to the class. The `Class` metaclass may also have another *LEM* with another metaclass, e.g., a `LifeLineType` in a sequence diagram, which is handled the same way by the *redefined* perspective action, *create class*.

A complex language action, on the other hand, affects more than one language element with one or more *language element mappings*, and, hence, requires a more systematic approach to enforce the consistency conditions between different instances of the language elements.

Definition 4.3.1 (Language Action Effect). Language action effects refer to the change in a state of a model element resulting from the execution of a language action.

Consider the *redefined* perspective action to create a sequence diagram and two *LEMs* (one between the `Operation` metaclass and `SequenceDiagram` metaclass (LEM_OSD) and another between the `Class` metaclass and `LifeLineType` metaclass (LEM_CLLT)) as shown in Figure 4.8. If the language action to create a sequence diagram also creates a lifeline and lifeline type automatically, the corresponding *redefined* language action (*create sequence diagram*) affects both LEM_OSD and LEM_CLLT. However, these *LEMs* are associated with different language elements. We group the effects of a language action into two categories: (1) the *primary* effect, e.g., creating the actual sequence diagram; (2) the *secondary* effects, e.g., creating the lifeline type and then the lifeline. The *relevant* secondary effect, however, is only the creation of the lifeline type in this case, because a *LEM* is defined for `LifeLineType` and a change is required for the mapped element (the `Class` in this case). The metamodel needs to capture only the relevant secondary effects of a language action, which we will refer to simply as secondary effects from now on.

A *relevant* secondary effect is based on existing *LEMs* in a given perspective. Hence, a given language action can have a *primary* effect, a *secondary* effect, or no effect in a perspective. If the *create lifeline type* action is directly called from the model view, then the effect of creating the lifeline type is the primary effect. Similarly, if there is no *LEM* that concerns the `LifeLineType` metaclass in a perspective, then the act of creating a lifeline type has no effect in the perspective. Hence, each perspective defines the effects of the language actions it augments. On the other hand, the language designer is mainly responsible for defining languages, which conform to our assumptions, i.e., languages that have metamodels

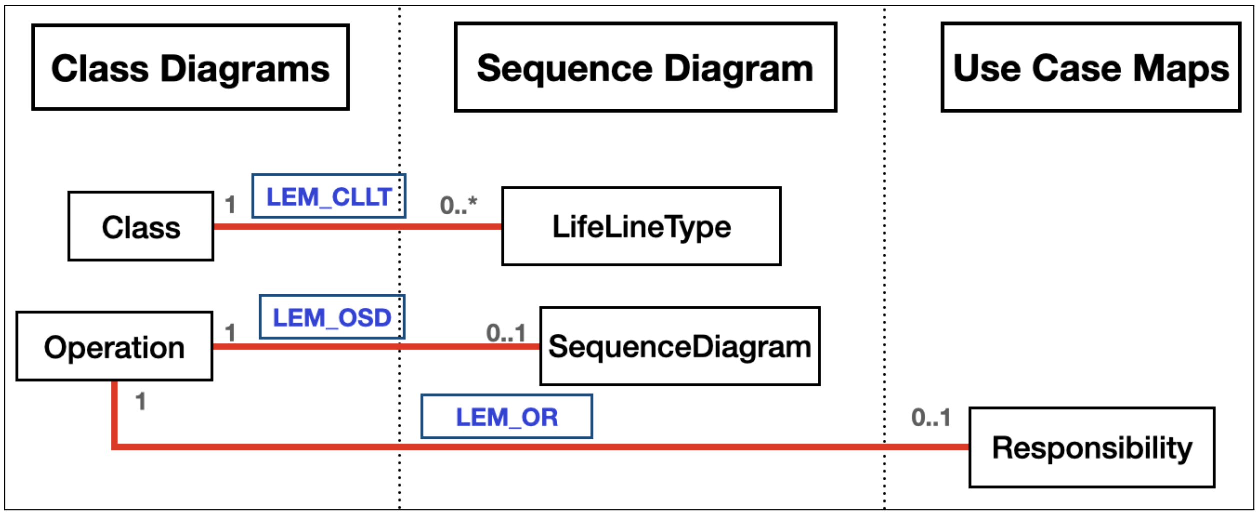


Figure 4.8: Examples of LEMs

and language actions. How the languages are used in each perspective is handled by the perspective designer.

To effectively propagate both the *primary* and *secondary* effects of each *redefined* language action (e.g., create sequence diagram), the `LanguageAction` metaclass encodes the *primary* effect, while the `ActionEffect` represents the *secondary* effects. The primary effect of each `LanguageAction` is inferred from the referenced *languageElement* and the attribute *actionType*. The propagation of the *primary* effect typically depends on the *LEM(s)* between the referenced *languageElement* of the `LanguageAction` in question and other language elements across language boundaries.

To handle the *secondary* effects of the *redefined* language action, each `ActionEffect` captures the necessary details required to propagate those changes across other models in the *multi-language* system. The `CreateEffect` encodes the details required to propagate changes due to creating new model element(s) as the *secondary* effect, e.g., the effect of creating a lifeline type in a *create sequence diagram* action. Hence, the `CreateEffect` references the type (`LanguageElement`) which is used to retrieve the new model element(s). For both `DeleteEffect` and `UpdateEffect` (i.e., `ExistingElementEffect`), the element to delete or update, respectively, should be directly or indirectly contained in the parameters of the *redefined* language action (see the reference from `ExistingElementEffect` to `Parameter`). An `ExistingElementEffect` is currently constrained to being directly or indirectly contained in a parameter, because we have not observed the need to support more elaborate schemes to identify the affected element of such secondary effects. When the *parameterEffect* is *element*, the *parameter* is the affected model element. Otherwise, the *parameter* is an identifier that can be used to retrieve the actual affected model element. Furthermore, each `UpdateEffect` references the affected attribute (i.e., the updated attribute of the model element) since a given model element can have more than one synchronized attribute. In general, these details are used to generate the perspective actions (with the help of a generic template) which enforce the consistency conditions between the concerned model elements.

The `LanguageAction` metaclass (see Figure 4.7) is a representation of an externally defined language action. *PML* uses the details of this representation, such as the *classQualifiedName* and *methodName* attributes, to generate the implementation of each perspective action that references the `LanguageAction` in question. Generally, the *PML* metamodel targets languages defined with Ecore. However, the metamodel can be applied to modelling languages based on other metamodelling environments with slight modification (e.g., by changing referenced `EObject` to `Object` for Java-based modelling languages).

4.3.4 Derived Parameter

A *redefined* perspective action reuses one or more language actions which are used to propagate the primary effects of the language action in question. Considering the *Software Design Perspective*, the *redefined* perspective action *createOperation* may be required to create a sequence diagram after creating an operation. Consequently, creating a sequence diagram may require to create a corresponding lifeline with a lifeline type according to the specifications of the perspective. In this case, the *redefined* perspective action *createOperation* reuses the *createSequenceDiagram* and *createLifeLineType* language actions. To propagate these effects of creating an operation, i.e., the creation of the sequence diagram (primary effect) as well as the lifeline type (secondary effect), the *redefined* perspective action needs to derive the parameters for each of the reused language actions from the original language action (i.e., the *redefined* language action). This parameter derivation is based on the fact that the parameters of the original action are known when it is called to create the model element, e.g., *operation*.

To support the parameter derivation in the *PML* framework, each instance of the `LanguageAction` metaclass is composed of a set of instances of the `DerivedParameter` metaclass. The `DerivedParameter` concept defines how to map a single parameter from the *redefined* language action parameter to the corresponding *reused* language action parameter, as shown in Figure 4.7. When the parameter of the *reused* language action cannot be derived from the *redefined* language action, then the *definition* of the `DerivedParameter` can assign a literal value (or default value) to the *reused* language action parameter in question. Also, the *definiton* can ask a user to provide the needed parameter value as the case may be.

4.3.5 Integration of the *PML* Metamodel with the *CORE* Metamodel

This section summarizes how we integrate the *PML* metamodel with the *CORE* metamodel and then explains how this integration addresses the problems of the old way of dealing with languages in *CORE* outlined in Section 2.5.2. Since *CORE* metamodel, which is also the metamodel of the TouchCORE tool, is used for the proof-of-concept implementation of this doctoral research work, we briefly explain how we integrate *PML* with the *CORE* architecture. Figure 4.9 shows the integration of the *PML* metamodel with the *CORE* metamodel. The changes are represented in green font colour for easier identification, and the newly introduced metaclasses are shown in orange.

Compared to Figure 2.7, we modify the name of the metaclass `COREModel` to `COREArtefact` and now it is contained in `COREConcern`, instead of being the root class of the language model. This change of name, as well as the containment association, reflects the idea that the concept (`COREArtefact`) can now be any software artefact supported by CORE, including *CORE* models and other related artefacts, e.g., *perspectives* and implementations of external languages.

In addition, we introduce a reference variable in `COREModelELement`, which contains an EObject reference to the model element in the corresponding language model. Some model elements in a model play special roles in a *COREConcern*, e.g., model elements that participate in the *customization interface* or *usage interface* (see Section 2.5.1). This reference approach decouples the actual language model elements from the *CORE* model, thanks to EMF, which supports cross references between different EMF models.

To support these changes in the *CORE* metamodel, we then adapt the *PML* metamodel (metaclasses in orange background) to fit into the *CORE* metamodel. Now, the `Language`

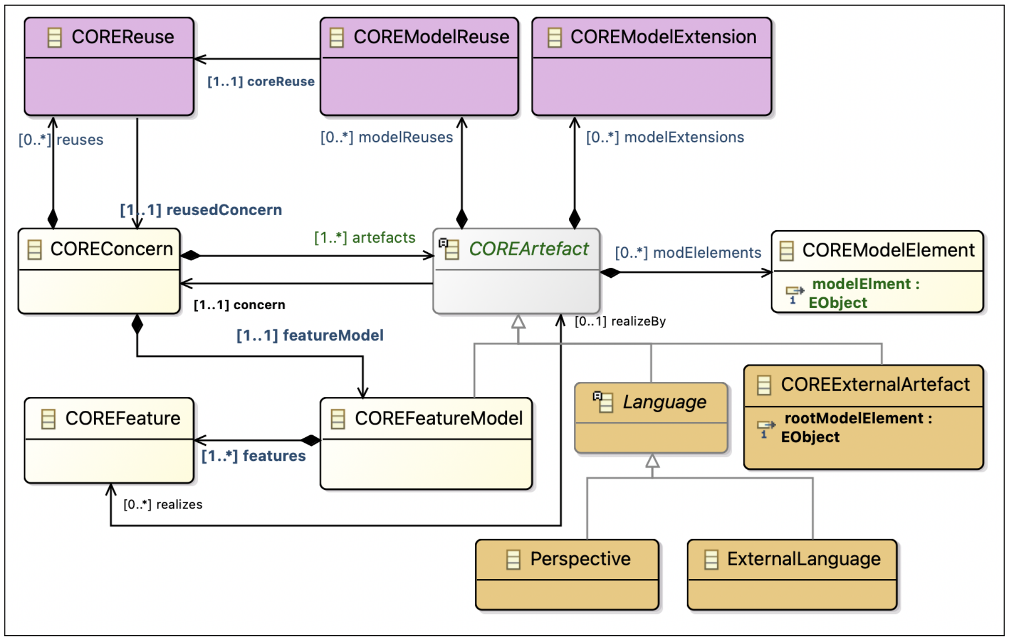


Figure 4.9: Adaptation of *PML* to *CORE*

metaclass (abstract), which is either a *perspective* or an *external language* (see Section 4.7) is a subclass of `COREArtefact`. This integration allows *CORE* to register *perspectives* and external languages as software artefacts. However, the implementation of *perspectives* and the language definitions do not depend on *CORE*. To manipulate the language models in *CORE*, we introduce `COREExternalArtefact`, which represents an external model within the concern, and which contains a reference to the root model element of an external model expressed in one of the supported CORE languages.

This current status of the *CORE* metamodel promotes separation of concerns because the *CORE* metamodel and the metamodels of the languages that can be used to elaborate realization models for a concern are completely independent. The *PML* framework handles coordination, as well as consistency, between models expressed with these languages. In addition, integrating *PML* with *CORE* addresses some of the problems with the old way of dealing with languages in *CORE* (see Section 2.5.2) as follows:

- **Independent Language Metamodels - Addresses P1**: Keeping each language metamodel separate from other language metamodels promotes the independent evolution of the languages. *PML* manages their relationships that would have been expressed in the single metamodel.

- ***CORE* Contains Model Reuses and Extensions - Addresses P2**: The `COREConcern` now contains both `COREModelReuse` and `COREModelExtension` since the adaptation of the *PML* framework with *CORE* decouples languages from the *CORE* infrastructure. However, the powerful *CORE* reuse methodology (`i.e.,` `COREModelReuse` and `COREModelExtension`) is maintained with the support of *PML*, which provides an architecture that allows *CORE* to reference external language model elements, e.g., to designate customization interfaces for models, or to specify mappings for the weaver.

- **Decouples *CORE* Metamodel from Language Metamodels - Addresses P3**: As shown in Figure 4.9, modifying the *CORE* metmaodel does not affect the external language definitions, including the language metamodel and language actions. Hence, both *CORE* and the external languages can evolve independently, which improves the modularity of *CORE* and the external languages.

4.4 Summary

In this chapter, we present the general overview of the *PML* framework. First, we discuss three main levels in the *PML* framework, which include language level, perspective level, and model level. The language level contains a collection of different languages that are used in a software system. The perspective level defines the perspectives, which reuse and group one or several languages for a given modelling purpose. At the perspective level, a perspective designer specifies the language element mappings, which encompasses the consistency constraints and the navigation mechanism to ensure that the concerned models can evolve consistently while supporting the modeller to interact and navigate the model elements. At the model level, a modeller can elaborate models that conform to the reused languages and are built according to the perspective. In addition, the modeller can create or delete *MEMs*, which are typed by the respective *LEMs* at the perspective level.

Furthermore, we show the workflow of the *PML* process. There are three main roles in the *PML* workflow process: language designer, perspective designer, and modeller roles. A language designer defines software languages and focuses on their building blocks, which include the abstract syntax, the concrete syntax, and the semantics. The perspective designer is responsible for the definition of a perspective, including the selection of languages as well as the specification of the language element mappings (LEMs) and the perspective actions. A modeller uses the perspective to create sets of models conforming to one or multiple software languages within the purview of a perspective.

In addition, we present the *PML* metamodel, which describes all the structural features of the framework and how it can be used to specify perspectives. And finally, we show how we adapt the *PML* metamodel with the *CORE* metamodel, and then explain how the integration addresses some of the *CORE* specific problems as well as other multi-language modelling challenges.

This chapter is based on the following publications:

1. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Towards a Framework for

Multi-Language Reuse. Presentation, 11th Workshop on Modelling in Software Engineering (MiSE 2019), Montreal, Canada, May 2019.

2. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Towards Modular Combination and Reuse of Languages with Perspectives. 1st International Workshop on View-Oriented Software Engineering (VoSE), Munich, Germany, September 2019. IEEE CS, 387-394. DOI: 10.1109/MODELS-C.2019.00060.

3. Ali, H., Mussbacher, G., and Kienzle, J. (2020) Action-Driven Consistency for Modular Multi-Language Systems with Perspectives. 12th System Analysis and Modeling Conference (SAM 2020), Montreal, Canada, October 2020. ACM, 95-104 DOI: 10.1145/3419804.3420270. (Acceptance rate: 62

4. Ali, H.. (2020) Multi-Language Systems Based on Perspectives to Promote Modularity, Reusability, and Consistency. 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2020), Montreal, Canada, October 2020. ACM, article no. 29, 1-6. DOI: 10.1145/3417990.3419489

5. Schiedermeier, M., Li, B., Languay, R., Freitag, G., Wu, Q., Kienzle, J., Ali, H., Gauthier, I., and Mussbacher, G. (2021) Multi-Language Support in TouchCORE. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS 2021), pp. 625-629, DOI: 10.1109/MODELS-C53483.2021.00096

6. Ali, H., Mussbacher, G., and Kienzle, J. (2021) Perspectives to Promote Modularity, Reusability, and Consistency in Multi-Language Systems. Innovations in Systems and Software Engineering, Special issue on Software and Systems Reuse, DOI: 10.1007/s11334-021-00425-3

In the next chapter, we present the *PML* generic navigation mechanism, which allows modellers to traverse their model elements both within and across model elements, potentially across language boundaries.

Chapter 5

PML Generic Language Navigation

As part of the *PML* framework, in this chapter, we present the *PML* generic navigation mechanism that aims to support software modellers in navigating their models, potentially across language boundaries. Modern model-based software engineering almost always requires the use of models expressed with different languages to capture the many different characteristics of complex systems. This set of models needs to be navigated to understand the system under development.

This navigation problem manifests particularly in the case of *perspectives*, where a perspective designer often combines different languages for a modelling purpose. In that case it is important that the modeller can navigate through the set of models in the context of the purpose of the perspective to fully understand the whole system.

Most of the existing software tools have navigation facilities that are tailored to the existing languages in the respective tools [50, 51]. However, the addition (or removal) of languages to such a tool, including domain-specific languages, requires a manual update of the existing navigation infrastructure to support the new languages. Instead of manually updating the navigation facilities to support a new language in a tool or a new perspective with a tailored need for navigable links, which is tedious and error-prone, we provide a generic navigation mechanism. This generic navigation mechanism can be tailored to the needs of a perspective. With this approach, a perspective designer can encode the specification of navigable links during the specification of a perspective, and our framework then generates the implementation of the navigation mechanism.

In this section, we first motivate our generic navigation facility with the help of four examples representing typical and common navigation situations. For each example, the relevant features of the generic navigation bar are elaborated. We further demonstrate the filtering of language elements using our generic navigation mechanism and then present the navigation metamodel.

5.1 Single-Model Navigation

The first situation concerns the navigation of a single model as in the case of a single-language perspective, i.e., intra-model (and hence also intra-language) navigation. A complex model may consist of many model elements, and it is hence desirable to have a concise and easy-to-use way to find important model elements. In this section, we illustrate intra-model navigation using a class diagram. In this case, e.g., a modeler may want to browse through all classes in the model, find operations, or navigate to superclasses. Figure 5.1 depicts a class diagram of a bank system and our navigation bar that makes this navigation possible.

Clicking the drop-down arrow under *BankClassDiagram* in the navigation bar pops up the *Classes* of the model, listed under the tab *Classes*. Clicking on a class reveals the operations and superclasses of the class in the navigation bar. In this example, we navigate from the class diagram to the class, `PensionAccount`, and then to its superclass, `Account`. Once a class is selected, the background of the class is highlighted in yellow in the model and centred on the screen, if needed, for easier identification.

To realize this navigation in our generic navigation bar, several *navigation mappings* have to be specified on the class diagram metamodel shown in Figure 5.2. The first navigation occurs from a class diagram to its classes (`Classifier`), the second from classes to operations (`Operation`), and the third from classes to its superclasses (`Classifier`). Consequently, the first required navigation mapping has the `ClassDiagram` metaclass as its source and the `classes` reference as its target. The second navigation mapping has the `Classifier` as its source and the `operations` reference as its target, while the third also has the `Classifier` as its source and the `superTypes` reference as its target. A reference is used as the target instead of a metaclass, because one metaclass may have several relationships with another metaclass.

These three navigation mappings each consist of one *hop*. However, it may also be necessary to skip intermediate elements and, e.g., define a navigation that goes from a class diagram directly to all the operations defined in the diagram without listing all the classes first. This requires the definition of multiple hops, e.g., the first hop is from the `ClassDiagram` to the `classes` reference and the second continues on with the `operations` reference. Therefore, a navigation mapping within a language has one *from* element (source) and one or several ordered *hops* (targets).

The navigation mapping from `Classifier` to `superTypes` is different compared to the other mappings, because it is useful to not only show the direct superclass of a class, but instead the complete hierarchy of superclasses. Therefore, the *closure* flag is set for this mapping, i.e., the modeller desires to recursively navigate or view a relationship in the model.

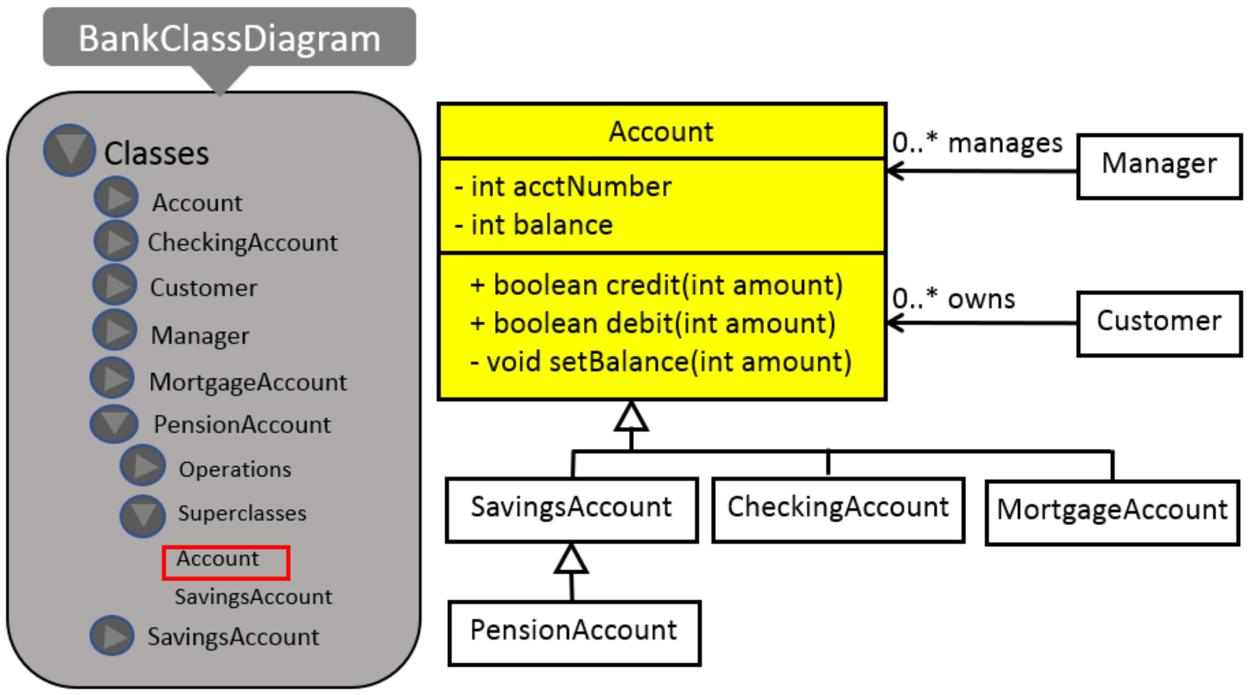


Figure 5.1: Bank Class Diagram

Table 5.1 summarizes the intra-model (and hence by default intra-language) mappings for a class diagram, where the value in the name column corresponds to the name used to name the tab in the navigation bar.

	from	hop	name	closure
1	ClassDiagram	classes	Classes	false
2	Classifier	superTypes	Superclasses	true
3	Classifier	operations	Operations	false

Table 5.1: Class Diagram Intra-language Mappings

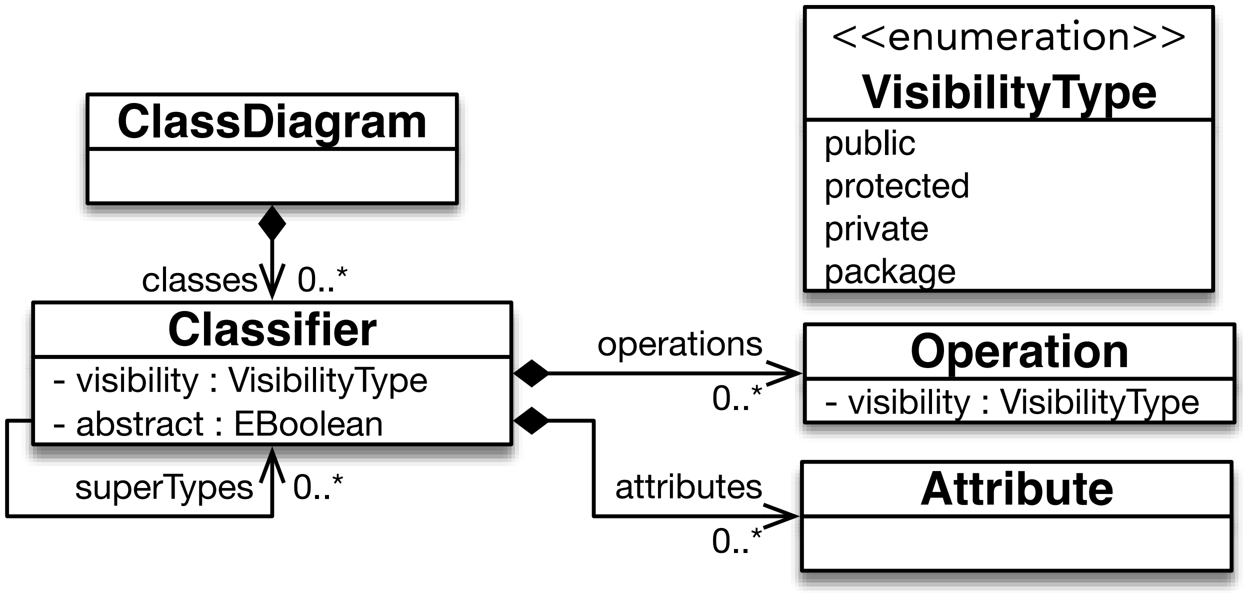


Figure 5.2: Class Diagram Metamodel (excerpt)

5.2 Multi-Model Navigation

The second situation concerns the navigation of multiple models as is the case in multi-view modeling, i.e., inter-model navigation. The navigation may involve models of the same type, i.e., intra-language navigation, or models from different languages, i.e., inter-language navigation.

Example 5.2.1. *An example of inter-model, intra-language navigation is a sequence diagram that defines the behaviour of an operation, which sends messages to invoke other operations. In this case, one may want to navigate from the invocation message in the first sequence diagram to another sequence diagram showing the detailed behaviour of the invoked operation.*

This navigation can be handled the same way as single model navigation, with the *from* element being the message and one *hop* to its sequence diagram reference. In this case, though, the reference points to a model element in a different model.

Example 5.2.2. *An example for inter-model, inter-language navigation is a class diagram and sequence diagram navigation, where one may want to navigate from an operation declaration to a sequence diagram defining the behaviour of the operation as shown in Figures 5.3 and 5.4.*

In this situation, the two languages – the class diagram language and the sequence diagram language – are used together in a specific way for a modelling purpose, i.e., class diagram and sequence diagram *perspective.*

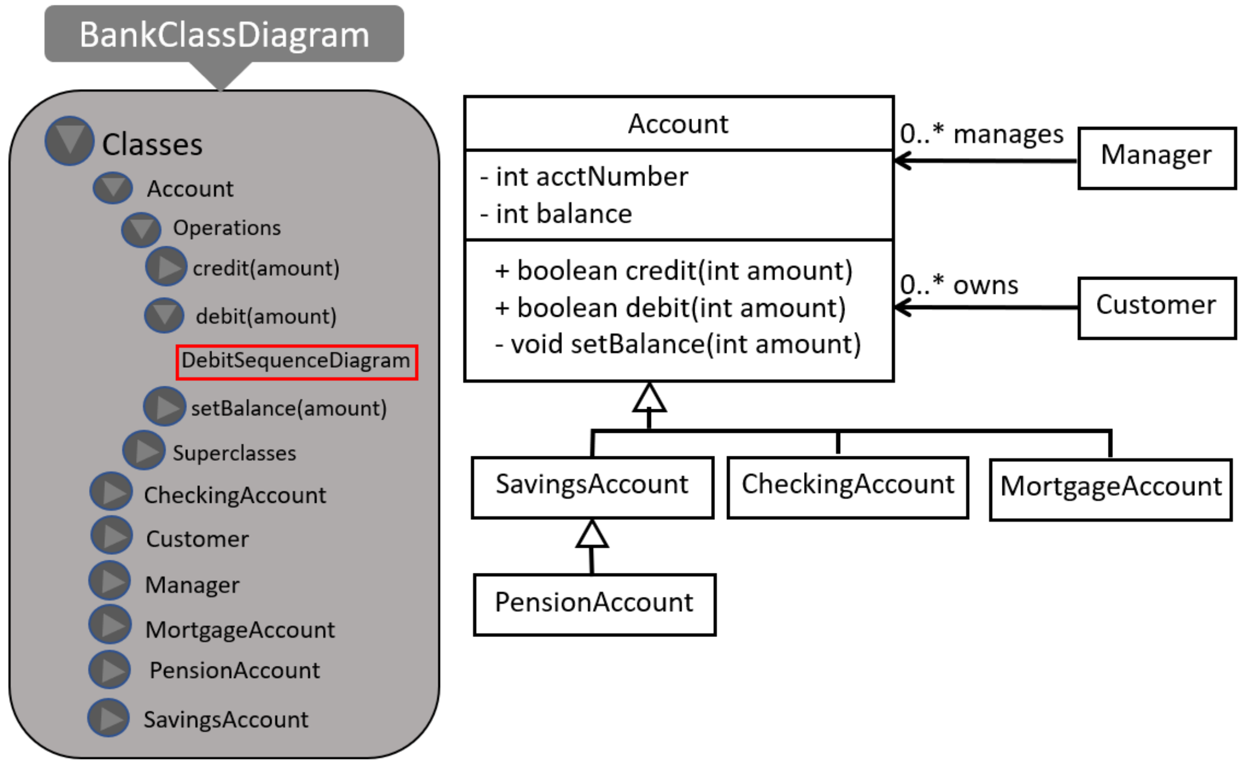


Figure 5.3: Operation to Sequence Diagram

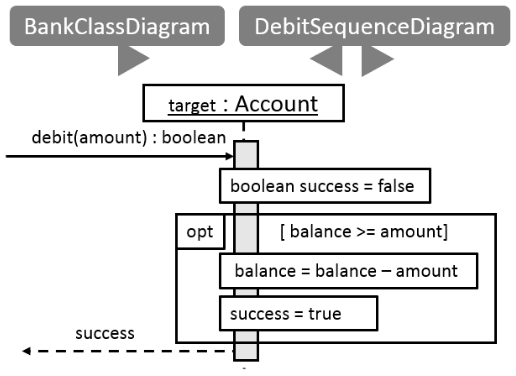


Figure 5.4: Sequence Diagram of Debit Operation

In the navigation bar, this connection is visualized also as the "right" arrow, which opens a drop-down list similar to intra-model navigation. When *debit(amount)* under the *Operations* tab is clicked, a list of other linked models pops up as shown in Figure 5.3. Upon clicking the *DebitSequenceDiagram* tab, as highlighted in the figure with a red box, the linked sequence diagram is opened as shown in Figure 5.4. Because this navigation involves a different type of model, the navigation bar is extended to display the class diagram model name as well as the sequence diagram model name to the right.

Navigating back to the class diagram can then simply be achieved by directly clicking on the class diagram name in the navigation bar. Furthermore, the sequence diagram has a "left" arrow which also opens a drop-down box to navigate any incoming inter-model navigation mappings in the opposite direction.

Example 5.2.3. *A workflow model may establish a mapping from one of its steps to the same sequence diagram. The "left" arrow then allows navigating from the sequence diagram to the class diagram or the workflow model.*

We also need to take into account that it is always possible to directly open any model using a file browser to view it. Even if the above sequence diagram is opened directly with a file browser (and not through navigation starting with a class diagram), the navigation bar should still show that the sequence diagram depicts behaviour that is best understood in the context of the class diagram or workflow model. However, which model should be shown to the left of the sequence diagram name in the navigation bar? To determine this, one incoming inter-model navigation mapping may be designated as the default one by setting its *default* tag.

The main difference to the intra-language navigation mappings is the fact that there exists no prior link between the metamodel of the class diagram language and the metamodel of the sequence diagram language (assuming that these two metamodels have been developed independently). Consequently, an inter-language mapping involves a *from* model element as is the case for intra-language mappings and a *to* model element instead of reference hops.

5.3 Software Product Line Navigation

The third situation is encountered during Software Product Line (SPL) development, which groups related model artifacts with commonalities and variabilities for a given family of products [5]. In SPL, a *feature* designates a user-relevant functionality or system quality that can be present or not in a product. A feature diagram describes the relationships among features, i.e., the set of feature configurations that produce valid products.

Figure 5.5 depicts a metamodel for feature diagrams. A `FeatureDiagram` basically has a list of `Features` with a parent/children relationship among them. Some of these features may be optional, while others are mandatory. A `requires` relationship exists when the selection of a particular feature demands the selection of another feature, while the `excludes`

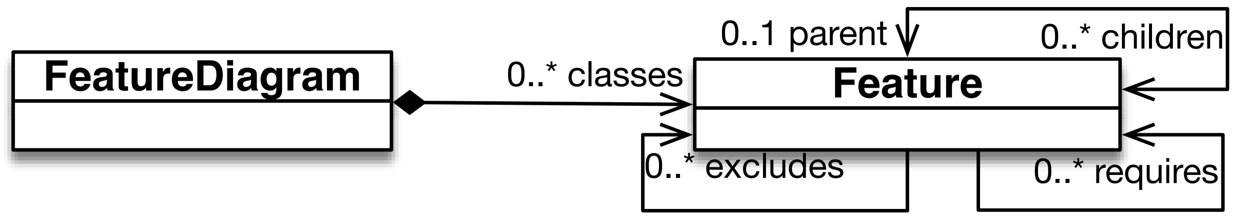


Figure 5.5: Feature Diagram Metamodel (excerpt)

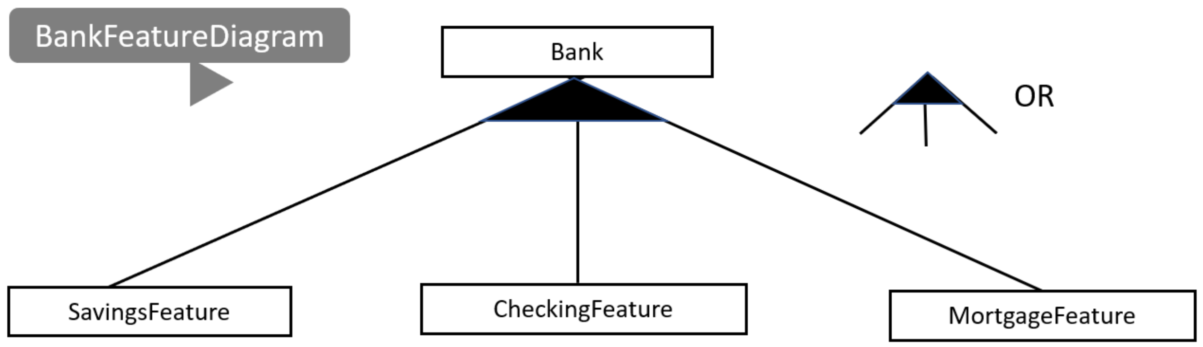


Figure 5.6: Feature Diagram of a Bank System

relationship ensures that two features are not simultaneously present in a given product.

Figure 5.6 shows a small example feature model for the bank system. It depicts features that involve different kinds of bank accounts. The features *SavingsFeature*, *CheckingFeature*, and *MortgageFeature* are in an OR relationship, meaning that at least one of them must be selected in order to create a valid configuration.

In model-driven SPLs, the structural and behavioural properties of features are described with models linked to these features. In additive variability, each feature is realized by one or several models, and to derive a product the realization models corresponding to the chosen features are composed with each other. In negative variability, a so-called 150% model describes the system with all features enabled. Each feature is linked to model elements related to the feature, and to derive a product the model elements that are not linked to any chosen features are removed from the model.

While negative variability requires a highlighting feature similar to what is shown in Figure 5.1, positive variability requires navigation among models. To illustrate feature-based navigation in SPLs, we split the bank account class diagram from Figure 5.1 into four smaller class diagrams to realize the account features of the feature diagram in Figure 5.6. Following the principle of positive variability, the class diagrams can then be composed (i.e., merged) to produce a bank model with the desired features.

Clicking on the "right" arrow under *BankFeatureDiagram* first shows the features (similar to classes in a class diagram) and then the models realizing a feature (similar to the sequence diagrams of operations). Selecting a feature highlights the feature in the feature diagram, while selecting a model of a feature takes the modeler to the model associated with this feature as illustrated in Figure 5.7.

Figure 5.7 shows the class diagram that contains the common structure used by *all* bank account features. At this time, though, the developer is currently working on the class diagram in the context of the *CheckingFeature*. This focus is depicted in the navigation bar by displaying the name of the *CheckingFeature* in the navigation bar instead of the *BankFeatureDiagram*. The "right" arrow under the CheckingFeature allows navigating to the models associated with the feature, i.e., the shared *AccountsClassDiagram* and the *CheckingClassDiagram* (which shows the generalization of the `CheckingAccount` class). The "left" arrow under the *AccountClassDiagram* shows, when clicked, a drop-down list with all other features that also use this class diagram.

Example 5.3.1. *When the "SavingsFeature" is clicked, the name "CheckingFeature" in the navigation bar is changed to "SavingsFeature", i.e., a context switch, and clicking*

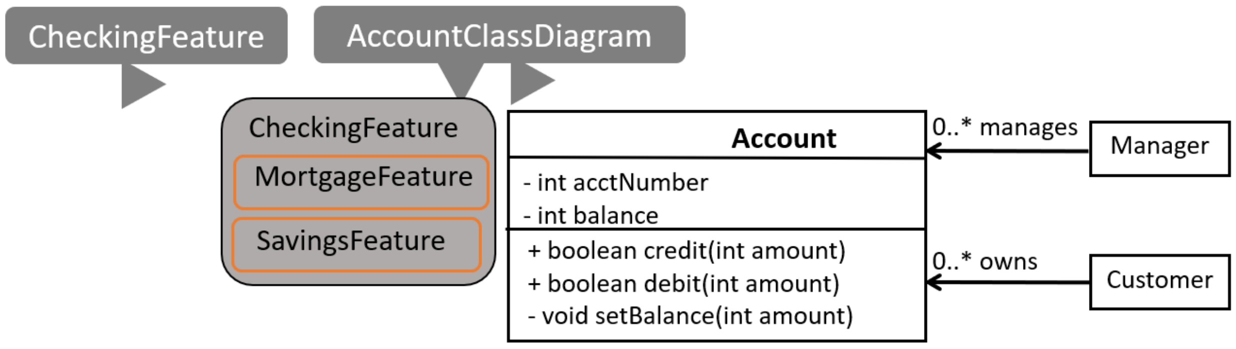


Figure 5.7: Account Class Diagram in CheckingFeature

the arrow under the "SavingsFeature" shows the models associated with it as shown in Figure 5.8.

In terms of required navigation mappings, feature-based navigation does not introduce any new kind of mapping. The mappings between a feature diagram and its features are intra-model mappings already discussed in Section 5.1. The mappings from features to class diagrams are inter-model, inter-language mappings already discussed in Section 5.2. However, since a feature is treated differently than other model element in terms of how it is displayed in the navigation bar, a `fromIsNavigationKey` flag needs to be set in its navigation mapping. This flag ensures that the corresponding feature of a model is displayed in the navigation bar.

Example 5.3.2. *When a modeller navigates from a class diagram to a class, to an operation, and then to a sequence diagram that defines the behaviour of the operation. At this juncture, the* from *element is the operation and the* to *element is the sequence diagram. However, the navigation bar shows the class diagram and sequence diagram, because the modeller switched from the class diagram to the sequence diagram through a sequence of navigation links (see Figure 5.4). Considering the navigation* from

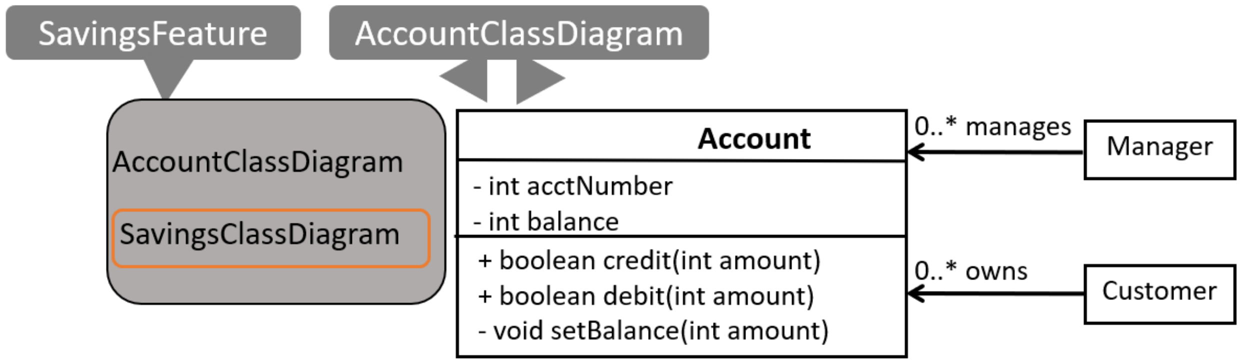


Figure 5.8: Account Class Diagram in SavingsFeature

operation to *sequence diagram,* fromIsNavigationKey *is not set, because the operation is not required to be shown in the navigation bar (i.e., the operation is not the navigation key). On the other hand, navigating from a feature diagram to a feature, and then to a model linked with the feature, as shown in Figure 5.7, displays the feature and the model in the navigation bar because the feature is the navigation key. Hence, the flag* fromIsNavigationKey *is set in this case to realize this context switch when the* from *element of a navigation link is required to be displayed in the navigation bar when a modeller traverses the link.*

5.4 Navigation of Reusable Artifacts

The fourth and final situation discussed here concerns the use of reusable artifacts during software development. As an example, consider the sequence diagram for `debit(amount)` in Figure 5.9 and assume that a reusable artifact for authentication exists with a sequence diagram as shown in Figure 5.10. When the `debit(amount)` sequence diagram reuses the `Authentication` sequence diagram, the body of the reusing sequence diagram replaces the box labeled with * in the reused sequence diagram. Consequently, the authentication check

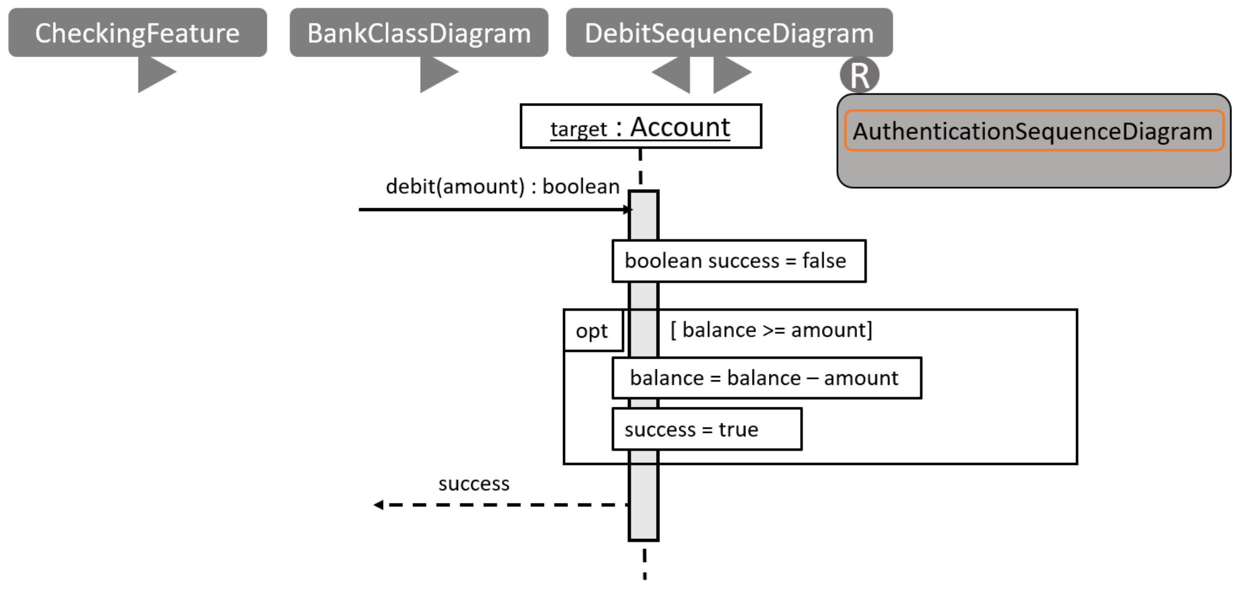


Figure 5.9: Reuse of Authentication

is performed before the body of the reusing sequence diagram. To specify this reuse, a composition specification needs to be provided that links the `debit(amount)` sequence diagram with the *Authentication* sequence diagram as defined in the metamodel for reuse specifications (see Figure 5.11). The reuse metamodel captures the links between reused elements and reusing elements with a mapping between `reused` and `reusing` elements, respectively. In our example, a mapping is established between the instance of the `SequenceDiagram` metaclass representing the `debit` sequence diagram to the `SequenceDiagram` metaclass instance representing the `authentication` sequence diagram. Once such a mapping in the reuse specification is established, it should be possible to navigate this composition link from a sequence diagram to another sequence diagram in a different reusable artifact with the help of the navigation bar.

To support this navigation, an "R" is displayed under the *DebitSequenceDiagram* in Figure 5.9. Clicking on it shows all reuses of this model (or individual model elements of the model).

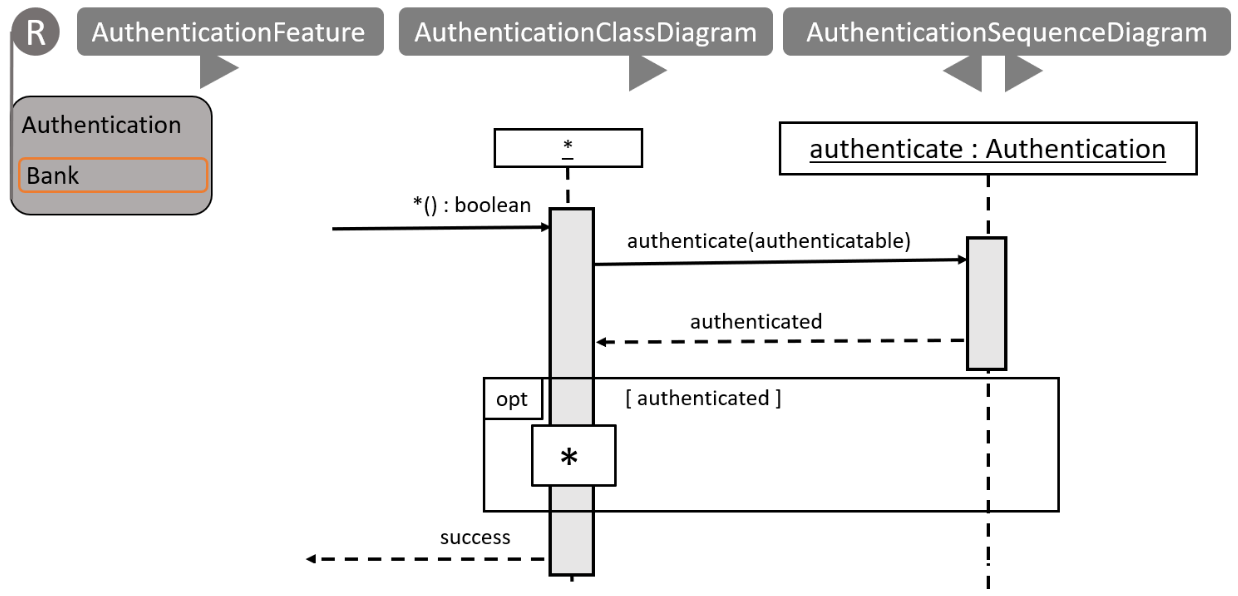

R
AuthenticationFeature
AuthenticationClassDiagram
AuthenticationSequenceDiagram
Authentication
Bank
*
authenticate : Authentication
*() : boolean
authenticate(authenticatable)
authenticated
opt
[authenticated]
*
success

Figure 5.10: Authentication Reuse Hierarchy

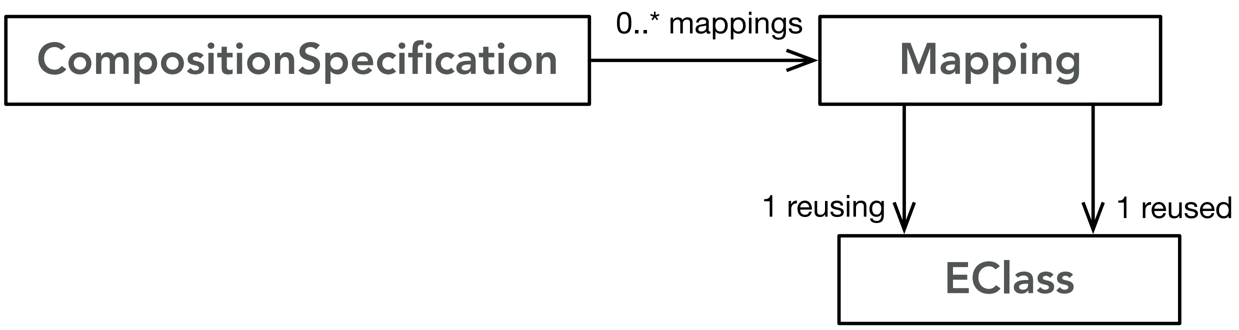


Figure 5.11: Reuse Metamodel (excerpt)

Once a reuse is selected, the modeler is taken to the reusable artifact. This involves a context switch, which results in the navigation bar showing the reused sequence diagram with its default parent (i.e., its class diagram) and the default parent of the class diagram (i.e., its feature). As shown in Figure 5.10, an "R" at the left of the navigation bar indicates the reuse hierarchy that is currently explored (e.g., the reusable artifact *Authentication* and the *Bank* that is reusing it). Clicking on an element in the reuse hierarchy results in direct navigation to that level.

In terms of navigation mappings, an intra-language mapping needs to be established (e.g., from the reusing sequence diagram to the reused sequence diagram). This navigation mapping requires a `from` element. This can be a model (or model element) (e.g., a sequence diagram in our example). Furthermore, two hops are required, which are references. The first hop is identified by the `reusing` reference and the second hop is identified by the `reused` reference. Note, however, that the `reusing` reference needs to be traversed in the reverse direction, because the reference is at the side of the source element of the hop (i.e., the reusing sequence diagram). Since reuse links are treated differently than other navigation links (due to the required context switch from the reusing artifact to the reused artifact), the `reuse` flag needs to be set for this navigation mapping.

5.5 Filtering of Model Elements

A complex model diagram may have a large number of model elements, which may be overwhelming to show in the navigation bar. To streamline navigation, there is in this case a need for filtering of model elements to allow modelers to focus on specific elements or groups of elements at a given time. In this section, we demonstrate how our generic navigation approach can handle filtering of model elements using the running example. A modeler may want to find all classes in a system and show only the *public* operations of each class. We demonstrate this mechanism with the class diagram shown in Figure 5.12, which depicts a bank system where the `Account` class has two public methods and one private method.

Similar to intra-language navigation in Section 5.1, clicking the drop-down arrow under *BankClassDiagram* in the navigation bar pops up the *Classes* of the model. Clicking on a class reveals the operations and superclasses of the class in the navigation bar. In this example, we navigate from the class diagram to the class, `Account`, and then only to its public operations, `credit(amount)` and `debit(amount)`.

To realize this filtering mechanism in our generic navigation bar, a *filtering condition* has to be encoded for the class diagram metamodel shown in Figure 5.2. We filter based on an attribute value of the relevant model element. For example, the filtering condition could be *abstract classes*, *public classes*, *protected operations*, or *private operations* to name a few. In Figure 5.12, the result based on filtering of *public operations* is shown. To achieve this, the filtering condition specifies the attribute of the metaclass that the filter should consider (i.e., the `visibility` attribute of the `Class` metaclass), the comparison value (i.e., the enumeration literal `public`), and a comparison operator (i.e., `EqualTo`). Table 5.2 demonstrates the filtering with two example conditions, each specified in a row.

To allow a modeler to dynamically configure which navigation mappings and associated filters the navigation bar uses to populate its content, it is possible to activate and deactivate navigation mappings at runtime through preference settings.

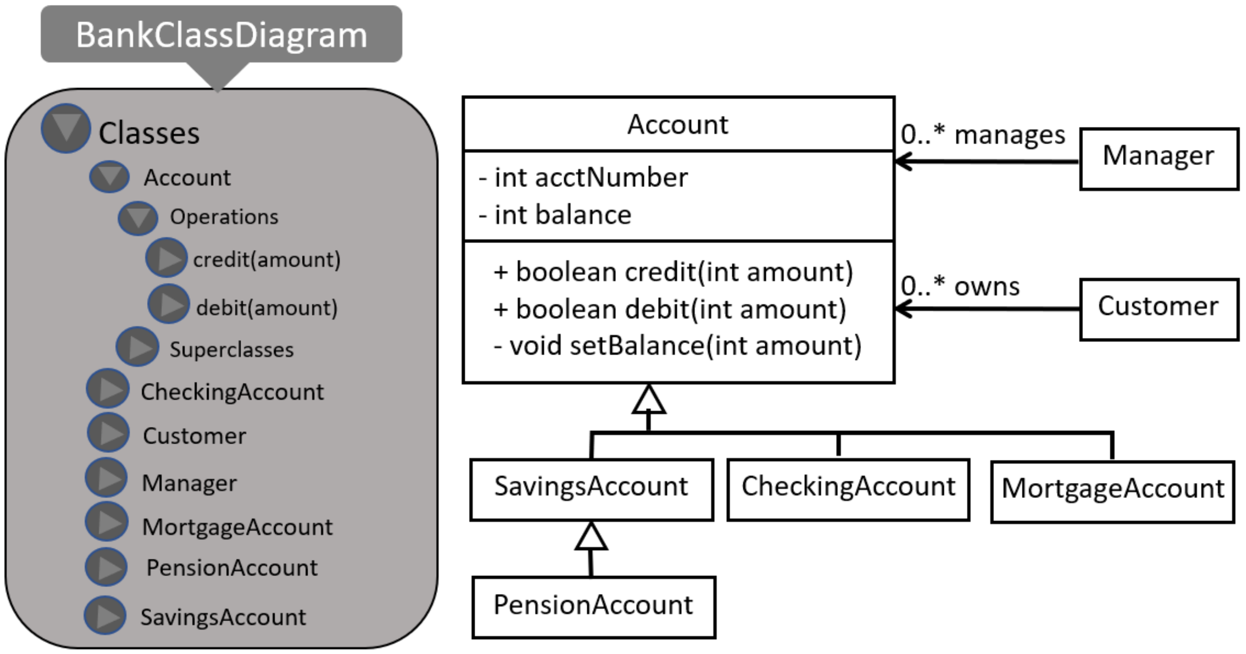


Figure 5.12: Bank Class Diagram

	condition	operator	element	operand	value
1	public operation	EqualTo	Operation	visibility	public
2	abstract class	EqualTo	Class	abstract	true

Table 5.2: Class Diagram Intra-language Filtering

5.6 Navigation Metamodel

This section describes our navigation metamodel that the designer of a language or modelling tool can use to define navigation mappings that configure our generic navigation bar. We elaborate our metamodel in the context of the Eclipse Metamodelling Framework (EMF), in which all metamodels are expressed using the metametamodelling language Ecore. As such, any model element that is part of a language metamodel and could be selected as the source of a navigation link is encoded as an instance of the class `EClass`.

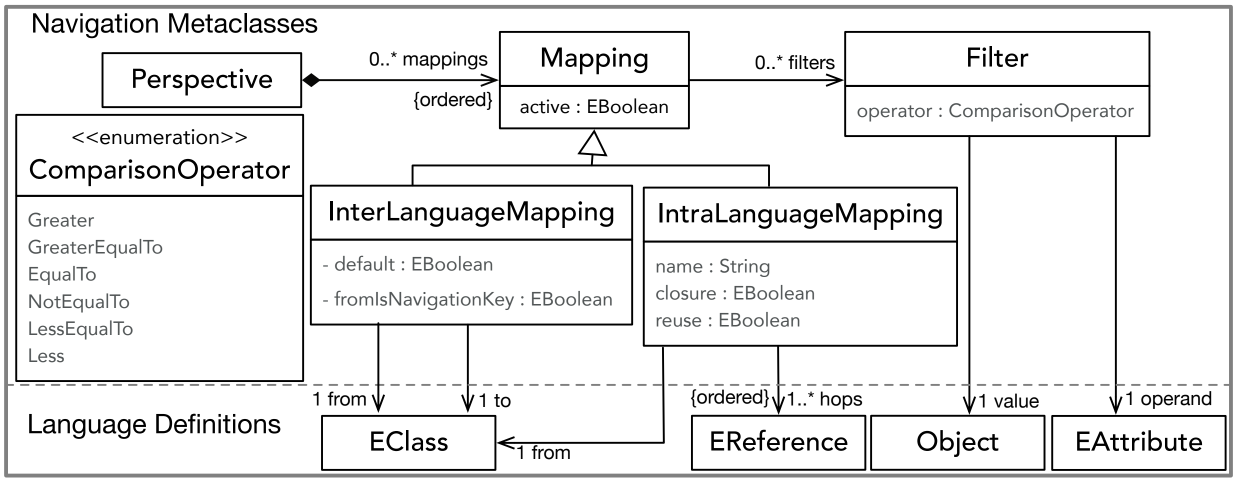


Figure 5.13: Navigation Metamodel

As explained with the examples above, for each `Perspective` there are two broad categories of navigation, namely intra-language and inter-language navigation, which are indicated by two metaclasses (`IntraLanguageMapping` and `InterLanguageMapping`)[1], see Figure 5.13. In intra-language, we navigate *from* a model or one of its model elements (represented as `EClass`) to one or several elements of the same language by following

[1]Recall that a *perspective* represents a purpose for using models expressed in one or several modelling languages during software development.

references. In language metamodels defined with Ecore, these references are instances of `EReference`. Since navigation might involve traversing several references, every `IntraLanguageMapping` therefore defines an ordered collection of `EReference` called *hops*.

Example 5.6.1. *When navigating from a class diagram to an operation of a class, the from reference would refer to the* `EClass` *ClassDiagram, the first hop would refer to the* `EReference` *classes, and the second hop to the* `EReference` *operations.*

Furthermore, each intra-language mapping has three attributes: `name`, `closure`, and `reuse`. The string attribute `name` allows the tool designer to specify the text that should appear in the navigation bar for this navigation. The boolean `closure` attribute can be set for any `IntraLanguageMapping` where the *from* `EClass` is identical to the model element referred to by the last *hop*. In this case, the navigation bar will traverse this mapping recursively and display all reached target model elements. In our example, `closure` is set when navigating from a class to its superclasses in order to display the entire superclass hierarchy in the navigation bar. The boolean `reuse` identifies an intra-language navigation mapping that requires a context switch.

In inter-language mappings, the navigation involves models of different software languages, e.g., navigating from an operation definition in a class diagram to the sequence diagram specifying the behaviour of the operation. Hence, for `InterLanguageMappings`, the *from* and *to* are always instances of `EClass`, and each mapping is a 1-to-1 relationship.

For `InterLanguageMapping`, the name of the `to` element is displayed in the navigation bar. Finally, the `default` attribute specifies whether the source of an inter-language navigation mapping identifies the default parent of a target model, and the `fromIsNavigationKey` attribute identifies key model elements (e.g., a feature) that need to be shown in the navigation bar instead of their model name.

As part of the *PML* metamodel, the `InterlanguageMapping` metaclass corresponds to the `LanguageElementMapping` metaclass (see Figure 4.7). For each instance of the `InterLanguageMapping`, there must be a corresponding instance of the `LanguageElementMapping`, where one of its two mapping-ends references a language element, which is also being referenced (i.e., either *from* or *to*) by the instance of the `InterLanguageMapping`. On the other hand, perspectives can have *LEMs* without corresponding inter-language mappings since all *LEMs* may not be required to be navigated. For example, a nested *LEM* does not require a navigation mapping since the nested mapping is mainly used to enforce consistency between model elements that participate in a *MEM*, which is typed by the parent *LEM*.

To support filtering of language elements, we attach a `Filter` to the `Mapping` metaclass, which is the superclass of the `InterLanguageMapping` and `IntraLanguageMapping` navigation mappings. This provides support for filtering within a model and between models potentially across language boundaries. Filtering is always applied on the `to` elements in the case of inter-language filtering, or to the elements designated by the `EClass` referred to by the last `hop` in the case of intra-language filtering. The `operator` attribute specifies the comparison operator for the filtering using pre-defined enumeration values as shown in Figure 5.13. A filter then compares the attribute value of the `operand` `EAttribute` with the `value` `Object` designated by the filter. E.g., the filtering conditions *public operations* and *abstract classes* from Table 5.2 use the `operator` *EqualTo*, the operands `visibility` of *Operation* and `abstract` of *Class*, and the values `public` and `true`, respectively. When several filter conditions are specified for a mapping, they are combined by an implicit logical AND.

To promote the generation of the navigation implementation, we favour the filtering approach against other notable model query languages (e.g., OCL). With the filtering approach, a perspective designer basically provides attribute values of the concerned model elements, and our framework then handles the implementation of the navigation mechanism. On the other hand, OCL can be used, too, to establish more complex filtering conditions, which can be used to generate the navigation implementation. However, the perspective designer is then required to face the full complexity of OCL, instead of our tailored generation approach.

Last but not least, the `active` attribute in the metaclass `Mapping` allows the navigation bar to be customized at runtime. For example, a modeller can toggle the active attribute to false if at some point he does not wish the operations of classes to show up in the navigation bar.

An implementation of our navigation bar ensures that the navigation information in the navigation bar is always up-to-date by registering as a listener to all model elements that are instances of EClasses involved in navigation mappings. Whenever a model is changed, the navigation bar is notified and the navigation links are adjusted according to the occurrences of the mappings in the model.

5.7 Summary

Model-driven engineering is a conceptual development framework where models of the system under development are created and manipulated using different formalisms at different levels of abstraction. Separation of concerns is further promoted when working with multi-view modelling, software product lines, and domain-specific modelling languages. While this separation into many interrelated models has many benefits, it also makes it harder for the developer to determine the relevant context when looking at a model, and to navigate from one model to related ones.

In this chapter, we present a metamodel that covers two categories of navigation, intra-language and inter-language navigation. The metamodel allows the designer of a modelling tool to generically capture the relevant navigation links between model elements in a set of models manipulated for a given purpose. It is done by establishing inter-language and intra-language mappings designating the relevant metaclasses and references in the metamodels of the involved languages. We illustrated the effectiveness of our navigation metamodel by examples that involved feature models, class diagrams, and sequence diagrams, but our approach can be applied to any modelling language that is defined by a metamodel.

We furthermore show how this generic information can be used to visualize the current context of a model with a navigation bar, and how to populate the navigation bar with navigation links. When a navigation link is clicked, we either highlight the chosen model element if that element is located in the current model, or we navigate to the model that contains the model element and update the navigation bar to reflect the new context.

This chapter is based on the following publications:

1. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Generic Navigation of Model-Based Development Artefacts. 11th Workshop on Modelling in Software Engineering (MiSE 2019), Montreal, Canada, May 2019. IEEE CS, 35-38. DOI: 10.1109/MiSE.2019.00013.

2. Ali, H., Mussbacher, G., and Kienzle, J. (2019) Generic Graphical Navigation for Modelling Tools. 11th System Analysis and Modeling Conference (SAM 2019), Munich, Germany, September 2019. Fonseca i Casas, P., Sancho, M.R., and Sherratt, E. (Eds.),

System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Springer, LNCS 11753:44-60. DOI: 10.1007/978-3-030-30690-8_3.

Chapter 6

Mappings and Generic Templates in Multi-Language Perspectives

This chapter provides more in-depth details about *LEMs* which were introduced in Section 4.3.1 and then presents the generic templates which are used to generate the implementation of the perspective actions that maintain the consistency conditions in *LEMs*.

6.1 Mappings

The specification of the *LEMs* and then the implementation of the perspective actions, as well as the navigation mechanism, are non-trivial tasks. Hence, we have defined two domain-specific languages (DSLs) which aim to assist a perspective designer specify the perspective actions as well as the *LEMs* between different language elements. Each relationship (i.e., `LanguageElementMapping`, see Figure 4.7) comprises two mapping-ends (i.e., `MappingEnd` metaclass). The mapping-end refers to one side of the mapping which references the corresponding language element and defines its multiplicity constraint (i.e., the number of allowable instances of each corresponding language element). In this work, we cover all four different multiplicity possibilities, which constrain the number of possible *MEMs* for a set of models: (1) *Optional* (i.e., 0..1 multiplicity), (2) *Compulsory* (i.e., 1 multiplicity), (3) *Optional-Multiple* (i.e., 0..* multiplicity), and (4) *Compulsory-Multiple* (i.e., 1..* multiplicity). Since each mapping-end can have one out of four possible multiplicities, there are 16 possible combinations for one *LEM*.

Furthermore, the designer needs to ensure that these consistency conditions are maintained at the model level by the perspective actions. The designer can manually implement the perspective actions to enforce the consistency conditions, but this is error-prone and tedious. Hence, we generate the perspective actions with the help of generic templates. As a result, the perspective designer only needs to focus on specifying the relationships between different language elements, and is not bothered with the detailed implementation of the perspective actions. In *PML*, mappings (*MEMs*) are not ordered, and hence our algorithm deals with them in some arbitrary order. Since the language element mappings are all binary and involve different pairs of metaclasses, the order of execution of our algorithm that deals with the primary effect has no influence on the end result.

However, when dealing with a chain of mappings (i.e., the primary element is mapped to other elements, which are mapped to yet other elements), there are two choices: traverse these mappings in a depth-first manner or breadth-first manner. Both ways would lead to consistent results, but the results would not necessarily be identical. In our implementation we chose a depth-first traversal, because it allows us to avoid the complexities that are associated with keeping track of all the traversed model elements to effectively propagate the chained changes when the element is selected during a subsequent iteration.

6.2 Generic Templates

In this section, we present the templates that maintain equivalency, equality, and multiplicity constraints in a generic way in situations where the constraints are based on mappings between model elements in the different models. A template covers a set of bidirectional relationships, each between two language elements, and dictates a sequence of actions (which include calls to respective language actions) to maintain model consistency. Considering the LEM_OSD as shown in Figure 4.8, a corresponding template dictates the steps of a *redefined* perspective action (*create sequence diagram*) that whenever a sequence diagram is created, an existing operation must be mapped (i.e., *MEM*) with the new sequence diagram or the actual language action, *createOperation*, is called to create a new operation and then establish a mapping (i.e., *MEM*) with the new sequence diagram. We have considered n-ary relationships but decided to not support them at this point, because we have not seen a need and standard language engineering environments such as MOF also do not support n-ary relationships for language elements. Our approach establishes the conceptual relationship with a *LEM*, and then uses the corresponding template to generate the *redefined* perspective actions. In a perspective, a language element can have multiple *LEMs* potentially across several language boundaries.

At each mapping-end, templates are applied to the corresponding *create*, *update*, and *delete* language actions. Hence, there are six templates required for a mapping-end combination, and potentially 96 templates to cover all 16 mappings, as shown in Table 6.1. An *update* template traverses existing *MEMs* to update mapped model elements, regardless of how many model elements are mapped. Thus, the *update* template is not shown in Table 6.1, because the same template can be applied to each mapping combination regardless of its multiplicities. This reduces the number of potential templates to 65. Similarly, an evaluation of all mappings for *delete* and *create* further reduces the number to 16. We include a description of the generic steps of the update, delete (D1-D3), and create (C1-C12) templates in Appendix A. In the next sub-section, we present the generic template workflow.

#	*LEM* Mapping	*from* mapping-end			*to* mapping-end		
		Multiplicity	Create	Delete	Multiplicity	Create	Delete
1	Compulsory Optional	1	C1	D1	0..1	C10	D2
2	Compulsory Compulsory	1	C2	D1	1	C2	D1
3	Compulsory Optional-Multiple	1	C3	D1	0..*	C6	D2
4	Compulsory Compulsory-Multiple	1	C4	D1	1..*	C6	D3
5	Optional Optional	0..1	C9	D2	0..1	C9	D2
6	Optional Compulsory	0..1	C10	D2	1	C1	D1
7	Optional Optional-Multiple	0..1	C11	D2	0..*	C5	D2
8	Optional Compulsory-Multiple	0..1	C12	D2	1..*	C5	D3
9	Compulsory-Multiple Optional	1..*	C5	D3	0..1	C12	D2
10	Compulsory-Multiple Compulsory	1..*	C6	D3	1	C4	D1
11	Compulsory-Multiple Optional-Multiple	1..*	C8	D3	0..*	C7	D2
12	Compulsory-Multiple Compulsory-Multiple	1..*	C8	D3	1..*	C8	D3
13	Optional-Multiple Optional	0..*	C5	D2	0..1	C11	D2
14	Optional-Multiple Compulsory	0..*	C6	D2	1	C3	D1
15	Optional-Multiple Optional-Multiple	0..*	C7	D2	0..*	C7	D2
16	Optional-Multiple Compulsory-Multiple	0..*	C7	D2	1..*	C8	D3

Table 6.1: Mapping Templates

6.3 Generic Template Workflow

This section presents an overview of the generic template workflow as shown in Figure 6.1. The basic workflow of the template starts with a request to *create* (**Redefined Create Action**), *delete* (**Redefined Delete Action**), or *update* (**Redefined Update Action**) a model element, e.g., create an operation in a class diagram model. When an edit request is made from a model editor, *PML* intercepts the language action call and then directs it to the corresponding *redefined* perspective action, which ensures that the consistency conditions, as specified in the corresponding *LEMs*, are maintained. In Figure 6.1, the blocks

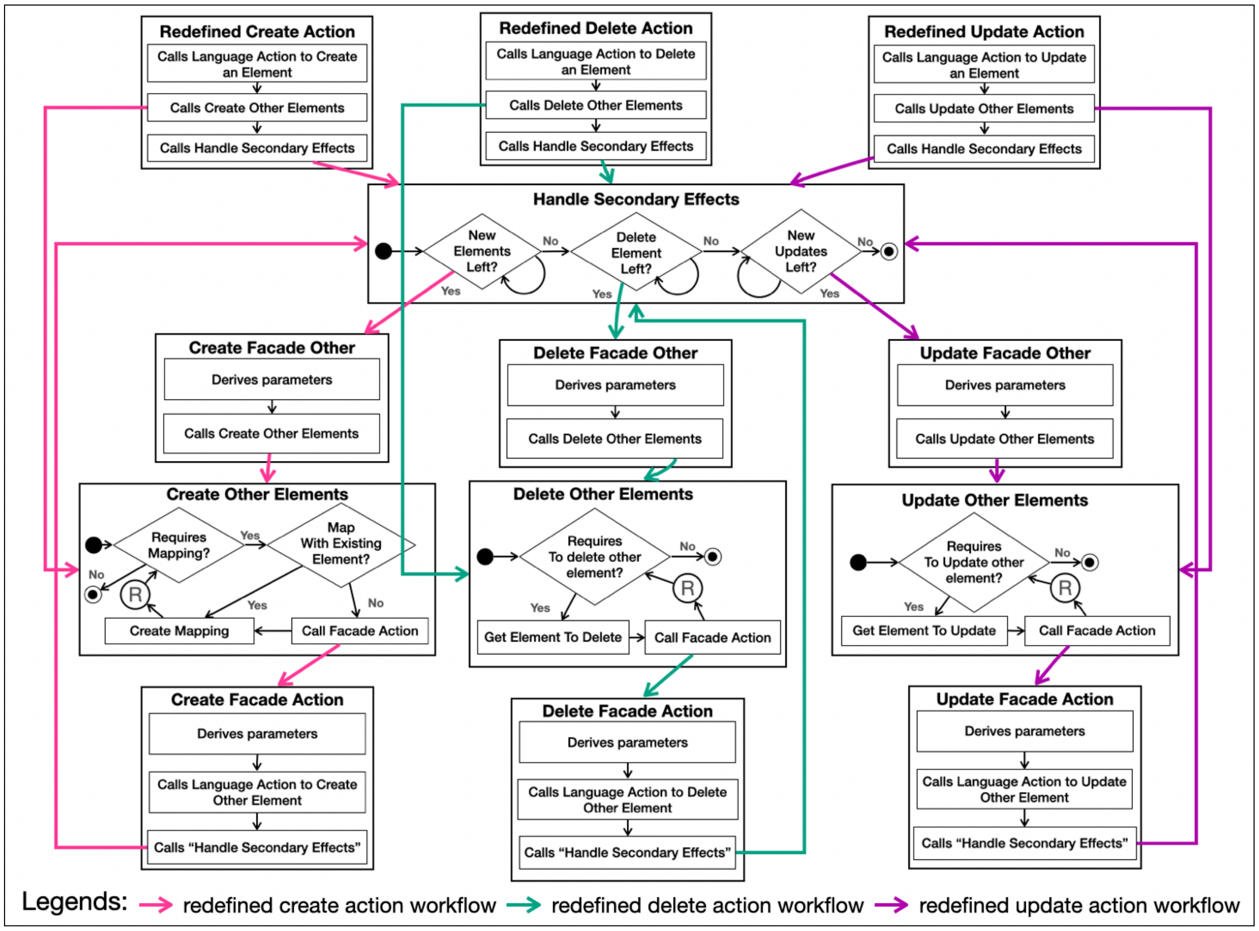

Redefined Create Action
Calls Language Action to Create an Element
Calls Create Other Elements
Calls Handle Secondary Effects
Redefined Delete Action
Calls Language Action to Delete an Element
Calls Delete Other Elements
Calls Handle Secondary Effects
Redefined Update Action
Calls Language Action to Update an Element
Calls Update Other Elements
Calls Handle Secondary Effects
Handle Secondary Effects
New Elements Left?
Delete Element Left?
New Updates Left?
No
Yes
Create Facade Other
Derives parameters
Calls Create Other Elements
Delete Facade Other
Derives parameters
Calls Delete Other Elements
Update Facade Other
Derives parameters
Calls Update Other Elements
Create Other Elements
Requires Mapping?
Map With Existing Element?
Create Mapping
Call Facade Action
R
Delete Other Elements
Requires To delete other element?
Get Element To Delete
Call Facade Action
Update Other Elements
Requires To Update other element?
Get Element To Update
Call Facade Action
Create Facade Action
Derives parameters
Calls Language Action to Create Other Element
Calls "Handle Secondary Effects"
Delete Facade Action
Derives parameters
Calls Language Action to Delete Other Element
Calls "Handle Secondary Effects"
Update Facade Action
Derives parameters
Calls Language Action to Update Other Element
Calls "Handle Secondary Effects"
Legends: redefined create action workflow redefined delete action workflow redefined update action workflow

Figure 6.1: Redefined Perspective Action Workflow

Redefined Create Action, **Redefined Delete Action**, and **Redefined Update Action** represent a *redefined* perspective action (`RedefinedAction`) which references a language action (`LanguageAction`) with a *create*, *delete*, and *update* action type, respectively, see Figure 4.7.

For each *redefined* perspective action, *PML* follows three steps to ensure that the rules of the perspective in the multi-language system are maintained. The first step is to call the requested language action, i.e., original language action, to *create*, *delete*, or *update* the model element.

Example 6.3.1. *An example of an original language action (i.e., externally defined language action) is the* createOperation *action in the class diagram language.*

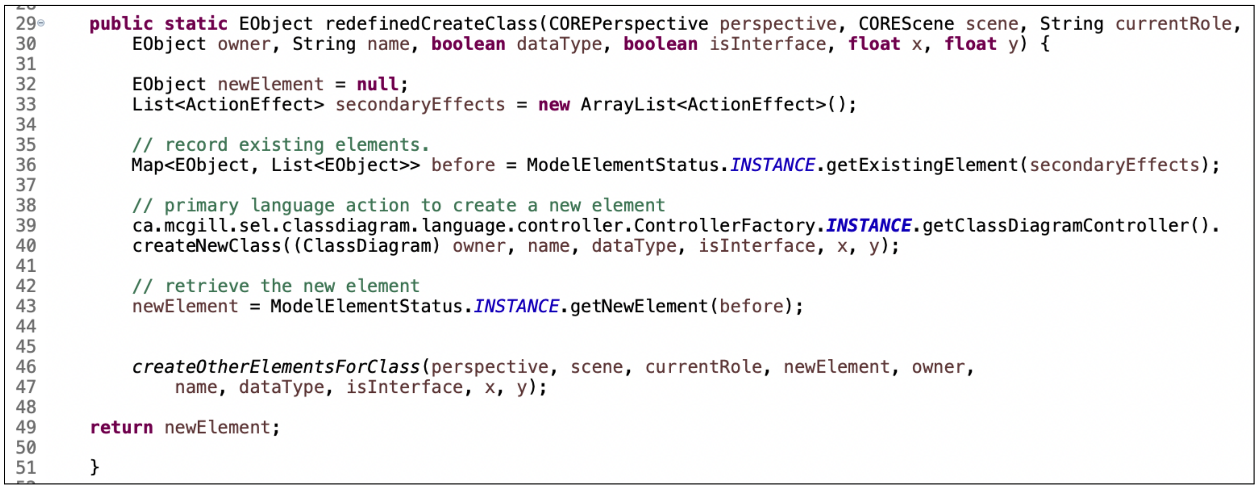
The externally defined language action is represented by the language action (`LanguageAction`) in the *PML* metamodel (Figure 4.7). Since the execution of this language action may require to *create*, *delete*, or *update* other model elements (as specified in the corresponding *LEM*), the second step calls the corresponding perspective recursive action, i.e., **Create Other Elements**, **Delete Other Elements**, or **Update Other Elements**, as shown in Figure 6.1. **Create Other Elements**, **Delete Other Elements**, and **Update Other Elements** represent the implementation of the set *reusedActions* (see Figure 4.7) which refer to language actions (`LanguageAction`) with a *create*, *delete*, and *update* action type, respectively. The role of each perspective recursive action is to propagate the primary effects of the language action.

Example 6.3.2. *Creating an* operation *in a class diagram may require to create a* sequence diagram *in a sequence diagram language and to recursively create a* responsibility *in a use case map language due to the creation of the* sequence diagram.

On the other hand, the execution of the original language action can have secondary ripple effects. For instance, when the *create sequence diagram* action is called to create a *sequence diagram* in a sequence diagram language, the execution of this language action can create a new lifeline type in the sequence diagram. This secondary ripple effect needs to be handled by the *perspective* when the type(s) of the affected model element(s) participate in another $LEM(s)$, e.g., the `LifeLineType` metaclass is mapped with the `Class` metaclass from a class diagram language. The causal effect of this complex language action is handled by the block **Handle Secondary Effects**, i.e., the *secondaryEffects* of the `LanguageAction` metaclass in Figure 4.7. Hence, the last step calls **Handle Secondary Effects** which takes care of the changes due to secondary effects in a similar way as the primary ones were taken care of to ensure that all the consistency conditions of the perspectives are maintained. An excerpt of the Java code *generated* for the *redefinedCreateClass* action is shown in Figure 6.2, while the pseudocode (generic template) for a *redefined* perspective action is contained in Appendix A . Line 39 calls the original language action to create the class, while line 46 calls the corresponding perspective recursive action. Note that the language action does not have secondary effects, hence, there is no code related to secondary effects.

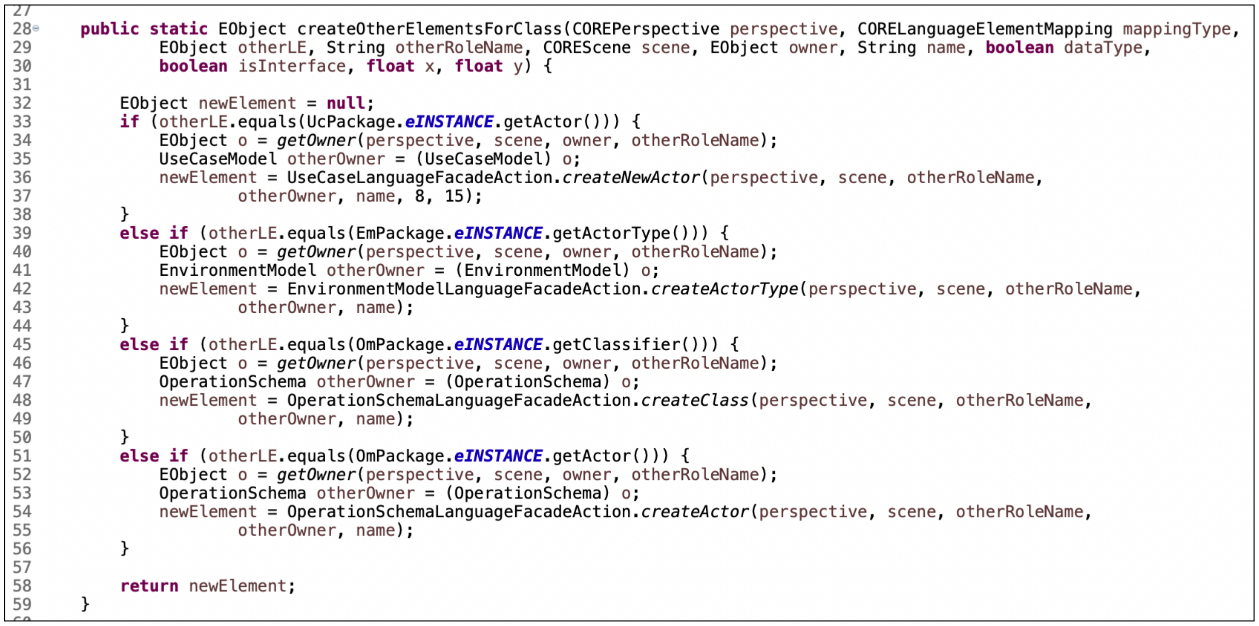
Create Other Elements first checks if the new element (i.e., *primary* element) requires to be mapped (*MEM*) with another model element (i.e., *other* element). If yes, the template evaluates if this mapping can be established with an existing model element or proactively creates a new element, with **Create Facade Action**, and then establishes the *MEM*. Whenever a new mapping is established, the templates recursively checks if the recently mapped *other* element requires another *MEM*. The recursion continues until the recently mapped *other* element requires no *MEM*. An excerpt of the Java code generated [1]

[1]The template used to generate this Java code is contained in Appendix A.



```java
    public static EObject redefinedCreateClass(COREPerspective perspective, COREScene scene, String currentRole,
        EObject owner, String name, boolean dataType, boolean isInterface, float x, float y) {

        EObject newElement = null;
        List<ActionEffect> secondaryEffects = new ArrayList<ActionEffect>();

        // record existing elements.
        Map<EObject, List<EObject>> before = ModelElementStatus.INSTANCE.getExistingElement(secondaryEffects);

        // primary language action to create a new element
        ca.mcgill.sel.classdiagram.language.controller.ControllerFactory.INSTANCE.getClassDiagramController().
        createNewClass((ClassDiagram) owner, name, dataType, isInterface, x, y);

        // retrieve the new element
        newElement = ModelElementStatus.INSTANCE.getNewElement(before);


        createOtherElementsForClass(perspective, scene, currentRole, newElement, owner,
            name, dataType, isInterface, x, y);

    return newElement;

    }
```

Figure 6.2: Redefined Create Class Action



```java
    public static EObject createOtherElementsForClass(COREPerspective perspective, CORELanguageElementMapping mappingType,
            EObject otherLE, String otherRoleName, COREScene scene, EObject owner, String name, boolean dataType,
            boolean isInterface, float x, float y) {

        EObject newElement = null;
        if (otherLE.equals(UcPackage.eINSTANCE.getActor())) {
            EObject o = getOwner(perspective, scene, owner, otherRoleName);
            UseCaseModel otherOwner = (UseCaseModel) o;
            newElement = UseCaseLanguageFacadeAction.createNewActor(perspective, scene, otherRoleName,
                    otherOwner, name, 8, 15);
        }
        else if (otherLE.equals(EmPackage.eINSTANCE.getActorType())) {
            EObject o = getOwner(perspective, scene, owner, otherRoleName);
            EnvironmentModel otherOwner = (EnvironmentModel) o;
            newElement = EnvironmentModelLanguageFacadeAction.createActorType(perspective, scene, otherRoleName,
                    otherOwner, name);
        }
        else if (otherLE.equals(OmPackage.eINSTANCE.getClassifier())) {
            EObject o = getOwner(perspective, scene, owner, otherRoleName);
            OperationSchema otherOwner = (OperationSchema) o;
            newElement = OperationSchemaLanguageFacadeAction.createClass(perspective, scene, otherRoleName,
                    otherOwner, name);
        }
        else if (otherLE.equals(OmPackage.eINSTANCE.getActor())) {
            EObject o = getOwner(perspective, scene, owner, otherRoleName);
            OperationSchema otherOwner = (OperationSchema) o;
            newElement = OperationSchemaLanguageFacadeAction.createActor(perspective, scene, otherRoleName,
                    otherOwner, name);
        }

        return newElement;
    }
```

Figure 6.3: Facade Action for Create Class Action

for the **Create Facade Action** (*createOtherElementsForClass*) is shown in Figure 6.3. Lines 33, 39, 45, and 51 compare the corresponding language element whose instance needs to be created with all the possible language elements in the *LEMs*. Considering the `Actor` language element in the use case language, the facade action handles the parameter mappings in lines 34-35 and then calls the original language action to create the actor at line 36 .

Similarly, **Delete Other Elements** ensures that when a model element (i.e., *primary* element) is deleted, then *other* mapped element may be required to be deleted. This is the case especially when the *other* model element has a *Compulsory* multiplicity relationship with the deleted element; hence, the *other* mapped element is not allowed to exist when the *primary* element has been deleted. The **Delete Other Elements** block first checks if the execution of the delete language action requires to delete another model element. If yes, the template retrieves the *other* element that needs to be deleted and then deletes the element, with **Delete Facade Action**. Furthermore, the template recursively checks if the recently executed *delete* action of the *other* mapped element requires to delete another element. The recursion, also, continues until the recently executed *delete* action does not require another model element to be deleted.

Update Other Elements is the same as the **Delete Other Elements**, except that the retrieved model element needs to be updated, instead of deleted. Moreover, the *redefined* update action does not depend on the multiplicities of both mapping-ends; instead, it uses the synchronized attributes of the model elements in question (i.e., nested mapping) to propagate the *update* changes. In general, the *redefined* create and delete perspective actions ensure that the multiplicity constraints are maintained between equivalent model elements, while the *redefined* update perspective action enforces the equality constraints.

As presented above, each of the **Create Other Elements**, **Delete Other Elements**, or **Update Other Elements** blocks calls the **Create Facade Action**, **Delete Facade Action**, or **Update Facade Action** to *create*, *delete*, or *update* a model element, respectively. The primary aim of each facade action is to call different language actions to *create*, *delete*, or *update* model elements. As shown in Figure 6.1, each *facade* action first handles the derivation of parameters from the initial parameters (i.e., the parameters used to call the *redefined* language action) to the parameters of each corresponding language action. Then, the **Create Facade Action**, **Delete Facade Action**, or **Update Facade Action** calls the corresponding language action to create, delete, or update, respectively, the *other* model element. Similar to the *redefined* perspective action, each *facade* action calls a language action which may have secondary effect(s). Hence, each *facade* action calls **Handle Secondary Effects** to propagate the potential changes accordingly.

Each time **Handle Secondary Effects** is called, it first checks if the recently executed language action created new *secondary* elements, other than the *primary* element in question. For example, creating a sequence diagram can automatically create a lifeline type; hence, the new element due to the secondary effect (in this case) is the lifeline type. Note that the workflow only checks for new elements which affect a *LEM* in the *perspective*. Furthermore, the workflow iterates over all the *secondary* elements, and then successively calls **Create Facade Other**. Similar to other *facade* actions, **Create Facade Other** derives the corresponding parameters of **Create Other Elements** from the initial parameters, and finally, calls **Create Other Elements** to propagate the *create* changes accordingly.

Further, **Handle Secondary Effects** iterates over all the deleted elements due to *secondary* effects. For each deleted element, the workflow calls the **Delete Facade Action** which derives the parameters of the **Delete Other Elements** from the initial parameters, and then calls the **Delete Other Elements** to propagate the *delete* changes accordingly. Similarly, **Handle Secondary Effects** calls **Update Facade Other** for each updated element due to *secondary* effects. Again, **Update Facade Other** derives the corresponding parameters of the **Update Other Elements** from the initial parameters, and then calls the **Update Other Elements** to propagate the *update* changes accordingly.

6.4 Template Workflow Example

In this example, we demonstrate the workflow of a *redefined* create operation language action in a class diagram language. This example is based on the following *LEMs* as shown in Figure 4.8: (1) *Operation* metaclass from the class diagram language (*Compulsory* mapping-end) and *Responsibility* metaclass from the use case map language (*Optional* mapping-end) - *LEM_OR*; (2) *Operation* metaclass from the class diagram language (*Compulsory* mapping-end) and *SequenceDiagram* metaclass from the sequence diagram language (*Optional* mapping-end) - *LEM_OSD*; and (3) *Class* metaclass from the class diagram language (*Compulsory* mapping-end) and *LifeLineType* metaclass from the sequence diagram language (*Optional-Multiple* mapping-end) - *LEM_CLLT*. Each of the *LEMs* has a nested mapping (i.e., synchronized mapping) between the *name* attributes of the metaclasses. The concrete workflow for this example is shown in Figure 6.4.

When the language action (*create operation*) is called from the editor of the class diagram language, *PML* intercepts the call and then redirects it to the *redefined* perspective action (*createOperation*), i.e., **Redefined Create Action**, as shown in Figure 6.4. *createOperation*, first, calls the *create operation* language action to create the

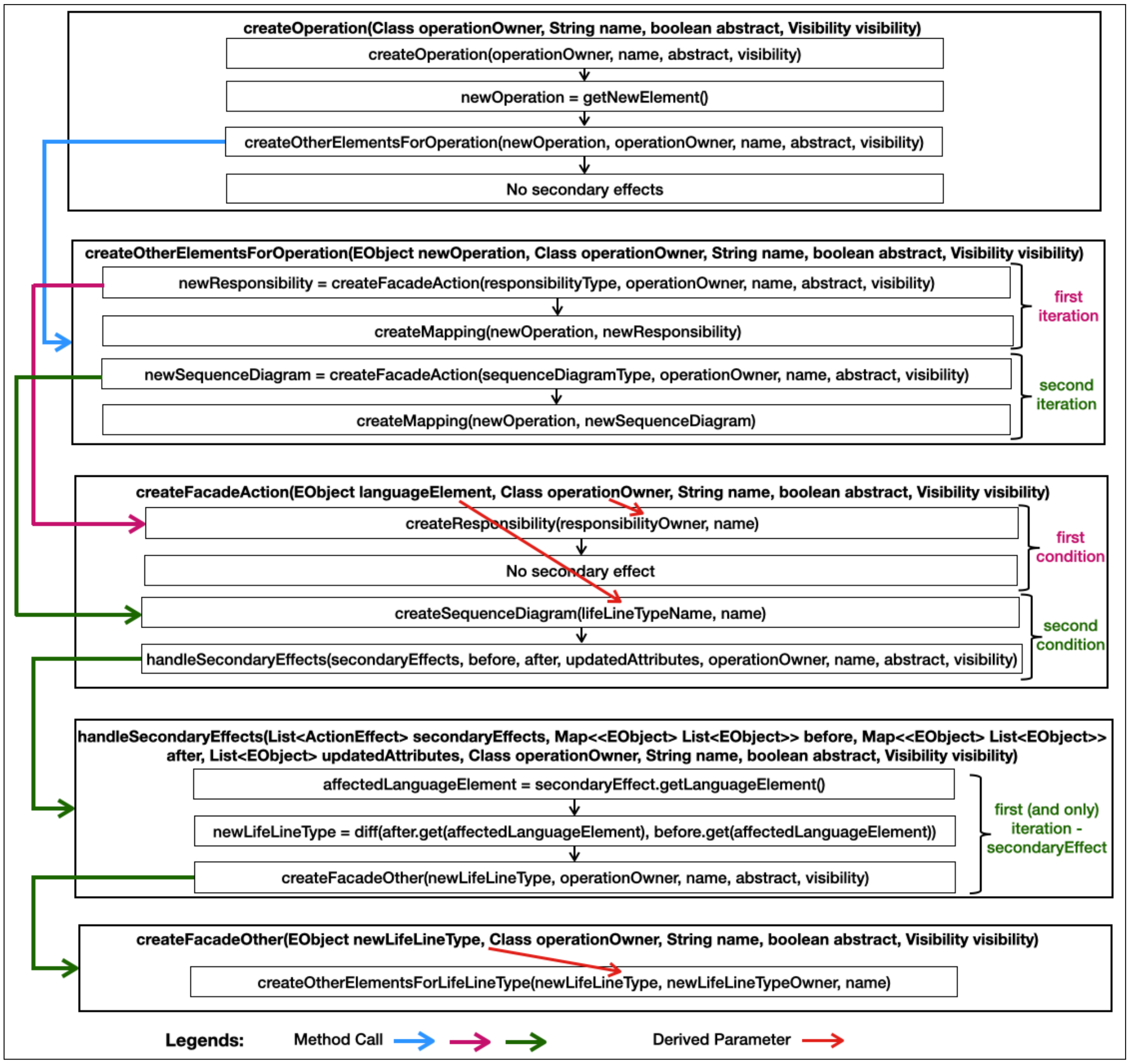

createOperation(Class operationOwner, String name, boolean abstract, Visibility visibility)
createOperation(operationOwner, name, abstract, visibility)
newOperation = getNewElement()
createOtherElementsForOperation(newOperation, operationOwner, name, abstract, visibility)
No secondary effects
createOtherElementsForOperation(EObject newOperation, Class operationOwner, String name, boolean abstract, Visibility visibility)
newResponsibility = createFacadeAction(responsibilityType, operationOwner, name, abstract, visibility)
createMapping(newOperation, newResponsibility)
first iteration
newSequenceDiagram = createFacadeAction(sequenceDiagramType, operationOwner, name, abstract, visibility)
createMapping(newOperation, newSequenceDiagram)
second iteration
createFacadeAction(EObject languageElement, Class operationOwner, String name, boolean abstract, Visibility visibility)
createResponsibility(responsibilityOwner, name)
No secondary effect
first condition
createSequenceDiagram(lifeLineTypeName, name)
handleSecondaryEffects(secondaryEffects, before, after, updatedAttributes, operationOwner, name, abstract, visibility)
second condition
handleSecondaryEffects(List<ActionEffect> secondaryEffects, Map<<EObject> List<EObject>> before, Map<<EObject> List<EObject>> after, List<EObject> updatedAttributes, Class operationOwner, String name, boolean abstract, Visibility visibility)
affectedLanguageElement = secondaryEffect.getLanguageElement()
newLifeLineType = diff(after.get(affectedLanguageElement), before.get(affectedLanguageElement))
createFacadeOther(newLifeLineType, operationOwner, name, abstract, visibility)
first (and only) iteration - secondaryEffect
createFacadeOther(EObject newLifeLineType, Class operationOwner, String name, boolean abstract, Visibility visibility)
createOtherElementsForLifeLineType(newLifeLineType, newLifeLineTypeOwner, name)
Legends:
Method Call
Derived Parameter

Figure 6.4: Workflow Example of a Redefined Create Perspective Action

operation. Note that, by definition, the parameters of *createOperation* can be directly used to call the *create operation* language action. Since the *Operation* metaclass participates in both *LEM_OR* and *LEM_OSD*, *createOperation* further calls *createOtherElementsForOperation* to **Create Other Elements** and also to establish *MEMs* with the new operation, respectively. However, *createOperation* does not call **Handle Secondary Effects**. While updating a class is a secondary effect of creating an operation and a *LEM* with a lifeline type exists for the class (see Figure 4.8), the effect is not a relevant effect. It is not a relevant effect, because the kind of update performed for the class is adding the operation to the list of operations of the class. However, this update does not require to *create*, *update*, or *delete* the other model element based on the *LEM*, i.e., the lifeline type. Hence, a secondary effect is not defined for the *create operation* language action.

createOtherElementsForOperation ensures that the consistency conditions of both *LEM_OR* and *LEM_OSD* are maintained. Assuming that the user has requested the system to create the corresponding elements, i.e., *responsibility* and *sequence diagram*, then *createOtherElementsForOperation* calls **Create Facade Action** to create the *responsibility* (first iteration, i.e., *LEM_OR*) and then the *sequence diagram* (second iteration, i.e., *LEM_OSD*). In the first iteration, *responsibilityType* is obtained from *LEM_OR*, and in the second iteration *sequenceDiagramType* is obtained from *LEM_OSD*. Furthermore, *createOtherElementsForOperation* creates *MEMs* between *newOperation* and *newResponsibility* (first iteration) and then between *newOperation* and *newSsequenceDiagram* (second iteration).

To create the *responsibility*, *createFacadeAction* calls the *createResponsibility* language action (first condition), i.e., when the *languageElement* is the *Responsibility* metaclass. The parameters of the language action can be derived from the parameters of the facade action, provided that the perspective designer provides the definitions on how to derive the parameters in a perspective specification (see `DerivedParameter` in Figure 4.7). In this method call, the *operationOwner* parameter is used to determine the required *responsibilityOwner*. On the other hand, the *name* attribute is directly used to call the *createResponsibility* language action. Since creating a *responsibility* does not require to *create*, *update*, or *delete* another model element (see Figure 4.8), the facade action (*createFacadeAction*) does not call **Handle Secondary Effects** during the first condition.

Similarly, considering the second condition, i.e., when the *languageElement* is the *SequenceDiagram* metaclass, *createFacadeAction* calls the *createSequenceDiagram* language action to create the *sequence diagram*, while the parameters *lifeLineTypeName* and *name* of the language action can be derived from the parameters of the facade action. Unlike the *first condition*, *createFacadeAction* handles a secondary effect during the second condition, because creating a sequence diagram automatically creates a lifeline type which needs to be mapped with a class in a class diagram language (*LEM_CLLT*). Hence, *createFacadeAction* calls *handleSecondaryEffects* to propagate the secondary effect of creating a *sequence diagram*.

handleSecondaryEffects iterates through all *secondaryEffects* – in this case there is only one – and first retrieves the affected language element, i.e., *LifeLineType* metaclass (see *PML* metamodel in Figure 4.7). The *affectedLanguageElement* is used to retrieve the new lifeline type instance by computing the difference between the list of existing lifeline types *before* and *after* creating the new sequence diagram. Finally, *handleSecondaryEffects* calls *createFacadeOther*, i.e., **Create Facade Other**. In *createFacadeOther*, the parameters are derived based on the definition in the perspective specification and *createOtherElementsForLifeLineType* is called to propagate the effects of creating a lifeline type. Finally, *createOtherElementsForLifeLineType* retrieves the existing class (i.e., the container of the new operation) and then establishes a *MEM* between the class and the new lifeline type. *createOtherElementsForLifeLineType* is not shown in Figure 6.4 since it has the same basic structure as *createOtherElementsForOperation*, i.e., **Create Other Elements**. Appendix A presents the details of the generic templates used by the workflow, which include update templates, delete templates, and create templates.

6.5 Summary

This chapter presents more in-depth details about *LEMs* and then generic templates, which are used to generate the implementation of the perspective actions that enforce the consistency conditions in *LEMs*. Each *LEM* comprises two mapping-ends, whereas each mapping-end refers to one side of the mapping which references the corresponding language element and defines its multiplicity constraint. We cover all four different multiplicity possibilities, which restrict the number of possible *MEMs* for a set of models. We apply generic templates at each mapping-end to generate the corresponding perspective actions that enforce the consistency conditions in *LEM*.

Furthermore, we cover one generic *update* template, three generic *delete* templates, and twelve (12) generic *create* templates. In addition, we show a detailed generic template workflow and then demonstrate the workflow of a *redefined create* operation language action in a class diagram language.

This chapter is based on the following publications:

1. Ali, H., Mussbacher, G., and Kienzle, J. (2020) Action-Driven Consistency for

Modular Multi-Language Systems with Perspectives. 12th System Analysis and Modeling Conference (SAM 2020), Montreal, Canada, October 2020. ACM, 95-104 DOI: 10.1145/3419804.3420270. (Acceptance rate: 62)

2. Schiedermeier, M., Li, B., Languay, R., Freitag, G., Wu, Q., Kienzle, J., Ali, H., Gauthier, I., and Mussbacher, G. (2021) Multi-Language Support in TouchCORE. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS 2021), pp. 625-629, DOI: 10.1109/MODELS-C53483.2021.00096

3. Ali, H., Mussbacher, G., and Kienzle, J. (2021) Perspectives to Promote Modularity, Reusability, and Consistency in Multi-Language Systems. Innovations in Systems and Software Engineering, Special issue on Software and Systems Reuse, DOI: 10.1007/s11334-021-00425-3

Following the in-depth presentation of more details about *LEMs* in the chapter, the next chapter presents some advanced features of a perspective.

Chapter 7

Advanced Perspectives

In this chapter, we present advanced features of a perspective, i.e., *set roles* and conditional *LEMs*. A *set role* exists in a perspective when there can be zero to many language models that play the same role. Here, we discuss perspectives with *set role* and then explain how we improve the *PML* framework to support this new feature. Furthermore, we present conditional *LEM*, i.e., a *LEM* that encompasses conditional equivalency. A conditional equivalency applies the equivalence to a subset of model elements, which dictates the state (*active* or *inactive*) of a *LEM*, provided that there is at least one model for each corresponding language role. A *perspective* applies the constraints of *active LEMs* and ignores the constraints of *inactive LEMs*. The conditional *LEMs* allow perspective designers to specify perspectives with more tailored *MEMs*, instead of mapping all elements that are instances of the referenced language elements, which can create an invalid *MEM*.

7.1 An Overview of Perspectives with Set Role and Conditional *LEM*s

This section provides an overview of a perspective with a *set role*, as well as the conditional *LEMs*. Figure 7.1 shows a *perspective* (EM_OM perspective) that combines the Environment Model language and the *Operation Model* language for a modelling purpose. In addition, the *perspective* comprises three different language element mappings: *LEM_MTOS*, *LEM_MTM*, and *LEM_ATA*. For a set of models built according to the *perspective*, there is always one instance of the Environment Model role and zero to many instances of the Operation Model role. Hence, the EM_OM perspective is a perspective with a *set role*, because it comprises a language role (Operation Model role), which can have zero to many instances. The following sections present some key challenges with perspectives with *set* role as well as the conditional *LEMs*.

7.1.1 Perspectives with Set Roles

So far, we have assumed that each language role in a perspective always has exactly one instance model that plays the role. However, this 1-to-1 relationship between a language role and its instances is often not the case, especially in a *perspective* that comprises behavioural languages, e.g., sequence diagram and operation model languages, to name a few. A *class diagram and sequence diagram* perspective always has exactly one class diagram role model and many sequence diagram role models, i.e., another example of a perspective with a *set role*. This 1-to-many relationship between a language role and its instances raises further challenges, including *zero model* and *multi-model* problems. A *zero model* simply means that a language role in a perspective does not yet have any model. On the other hand, *multi-model*

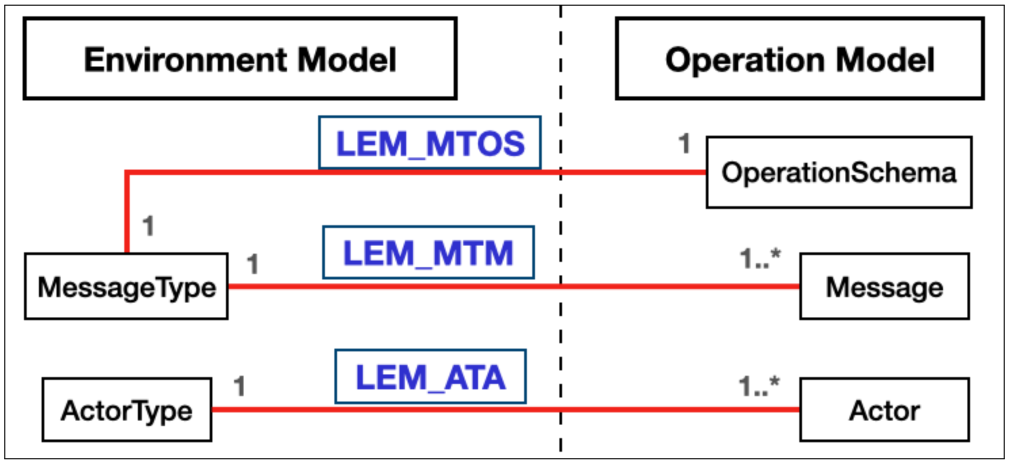


Figure 7.1: Examples of *LEMs* in a Perspective with Set Role

indicates that a language role has more than one model in a *perspective*. Furthermore, there is an ambiguity of the mapping-end multiplicity (e.g., 1..* for Message in Figure 7.1) due to set roles. In the following, we provide more details of these problems and then present how we address the challenges.

Zero Model

PML manipulates model elements to ensure consistency between models built according to a *perspective*. However, it makes no sense to try and interpret or enforce constraints that involve set roles when the role has zero instances, i.e., there is no model corresponding to the role (i.e., zero model). In our previous EM_OM perspective example this is potentially the case: there is always one instance of the Environment Model role (one model), but the Operation Model role might not have any instances at a given point in time (zero model). It becomes therefore unrealistic to enforce the consistency constraints specified in the *LEMs* (LEM_MTOS, LEM_MTM, and LEM_ATA). This situation requires perspectives to mark the corresponding *LEM* as *inactive*, i.e., a *LEM* that references a language role without a model. For instance, the EM_OM perspective marks LEM_MTOS, LEM_MTM, and LEM_ATA as *inactive* when there is zero instance of the Operation Model role.

Multi-Model

On the other hand, when a language role has more than one model, e.g., two Operation Model role instances for the EM_OM perspective, it becomes ambiguous as to which model should be proactively used for a specific *MEM*. For example, in the case where the Operation Model set role has two models, creating a *message type* in an Environment Model requires to create at least one corresponding *message* in one of the two Operation Model role instances, i.e., LEM_MTM multiplicity constraints (1..*), and then establish a *MEM* between the two elements. To prevent an inconsistency that might result from this action, the *perspective* can randomly select one of the two models and then create the corresponding message in the model. However, this random selection may not suit the need of the modeller.

Insufficient Mapping-End Multiplicity

Furthermore, since a perspective with a *set* role can have many instances of the language role, the existing mapping-end multiplicity is not sufficient to handle the multiplicity constraint conditions. The mapping-end multiplicities in Figure 7.1 indicate the minimum and maximum allowable instances of the referenced language element that can be mapped to a corresponding model element. For example, a *message type* can only be mapped with one of the *operation schema*s across all models in the Operation Model role. Similarly, an *actor type* can be mapped with 1 or many actors in all models in the Operation Model role. Note that once an *actor type* is mapped with one of the actors in an instance of the Operation Model role, the multiplicity constraint is satisfied.

However, some multiplicity conditions may require the perspective to map a single element from a language role model to a corresponding single element in each of the corresponding language role models. For example, given that an instance of a `MessageType` is required to be mapped with exactly one instance of the `Message` in each Operation Model role instance, then the multiplicity (1..*) is not sufficient to handle the constraint condition. Hence, the *perspective* requires to track multiplicity constraints between a set of language role models as well as between a pair of models, each from a mapping-end in a *LEM*.

7.1.2 Perspective with Conditional ***LEMs***

Another special case in this EM_OM perspective is that a language element (*MessageType*) from the Environment Model role is mapped to two different language elements (`OperationSchema` and `Message`) from the Operation Model role. Often, a language element is not expected to be mapped with different language elements from another language role unless the language element in question has a distinguishing property (or attribute) that dictates the condition for the corresponding *MEMs*.

Hence, a *LEM* can now depend on the properties of the referenced language elements whose values dictate the model elements that participate in a particular *MEM*, i.e., conditional *LEMs*. For each conditional LEM, the perspective filters the potential model elements, i.e., all instances of the referenced language role elements, to determine the model elements that qualify to participate in a *MEM* based on the conditions of the *LEM* in question.

Example 7.1.1. *LEM_MTOS and LEM_MTM* LEMs *are encoded with constraints which dictate that only input* message type *is required to be mapped with an* operation schema*, while an output* message type *is required to be mapped with* messages *in the Operation Model role (see Figure 7.1).*

Hence, each *LEM* may be *active* or *inactive* depending on whether an instance of the `MessageType` is an input *message type* or an output *message type.* With an input *message type*, LEM_MTOS is *active* while LEM_MTM is *inactive.* Conversely, with an output *MessageType*, LEM_MTM is *active*, while LEM_MTOS is *inactive.* Note that a *LEM* can have more than one constraint. Hence, such a *LEM* is *active* for a given model element if all the constraints are true.

In the following sections, we explain how we address the challenges outlined above in our PML implementation with *MDE* technologies, including *OCL.*

7.2 Set Role and Conditional LEM Features

To support perspectives with a *set role* and conditional *LEM*, we improve the *PML* framework with the new features shown in Figure 7.2, which is an improved version of Figure 7.1 (EM_OM perspective).

7.2.1 Language Role Multiplicity

First, we introduce a *language role* multiplicity, i.e., a multiplicity constraint that dictates the allowable number of language role models. For a set of models built according to the *perspective*, there is exactly one instance of the Environment Model role and zero to many instances of the Operation Model role (set role), as indicated by the blue background text at the top of each modelling language. In this chapter, we focus on *set role* since language role with 1 multiplicity is the same as the language roles presented so far in this thesis.

7.2.2 Model Multiplicity

Second, we incorporate a mapping-end multiplicity between a pair of models, each from a participating language role in the perspective, i.e., *model* multiplicity. The *model* multiplicity can dictate, for example, that an instance of the `MessageType` (Environment Model role) is required to be mapped with exactly one instance of the *Message* in each model of the Operation Model role instances, i.e., 1-to-1 relationship between a pair of models.

To this end, we improve the *PML* framework with a new type of mapping-end multiplicity (*model* multiplicity). An improved version of EM_OM perspective (Figure 7.1) is shown in Figure 7.2, which shows the representations of both the regular mapping-end multiplicity and the *model* mapping-end multiplicity.

7.2.3 Conditional LEM

Third and finally, we can now encode a *LEM* with constraints, which dictate allowable model elements that can participate in a *MEM*. As shown in Figure 7.2, *LEM_MTOS LEM* comprises a conditional equivalency that only allows an input *message type* to be mapped with a corresponding *operation schema* in the Operation Model role. Similarly, *LEM_MTM LEM* only allows an output *message type* to be mapped with the corresponding *message*s in the Operation Model role.

This section introduces the three new features to support *perspectives* with a set role, as well as conditional *LEM*. The first feature, *language role multiplicity*, restrains the

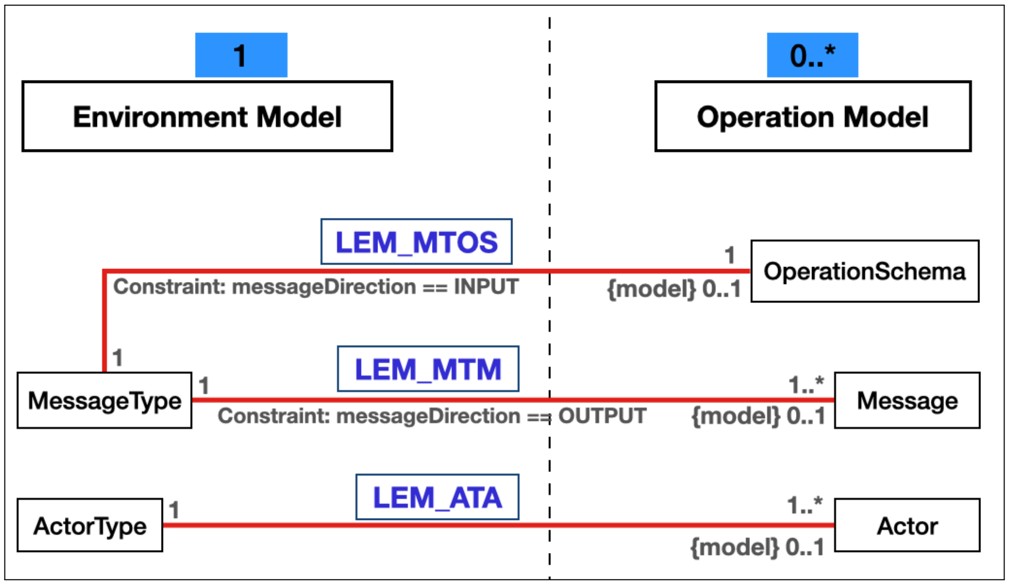


Figure 7.2: Examples of *LEMs* with Set Role and Conditional LEMs

allowable number of language role models. The second feature, *model* multiplicity, establishes multiplicity constraints between a pair of models, each from a language role, while the third feature, *conditional LEM*, outlines conditions that qualify model elements that can participate in a *MEM* based on the *LEM* in question. The following section provides more details on how we use OCL to implement the new features.

7.3 Implementations with OCL

This section presents more details about the *set role* and conditional equivalency with illustrations on how we implement the new features in *PML* framework.

7.3.1 Set Role Constraints

To address the challenges discussed earlier in this chapter, i.e., *zero model*, *multi-model*, and *insufficient mapping-end multiplicity* challenges, we now support perspectives with *set role* constraints. The *set role* constraints are constraints that implement *language role* multiplicity and *model* multiplicity. The implementation of the constraints is generic; hence, each *set role* constraint can be applied to all perspectives. Hence, regardless of the language role multiplicities and model multiplicities specified by a perspective designer for a specific *perspective*, the generic *set role* constraints can enforce the designer's specifications.

Set Role Constraints that Address Zero Model Challenge

When a language role does not have a model in a perspective, it becomes non-deterministic on how to maintain the consistencies of the *LEMs* that reference language elements that are contained in the language role. In this case, the *set role* constraints can be leveraged to set the concerned *LEM inactive* until the language role in question has at least one model. In this section, we demonstrate how we use OCL to implement *set role* constraints that address the *zero model*.

Taking into account the *perspective* shown in Figure 7.2, when a modeller is elaborating an instance of the Environment Model role without a corresponding instance of the Operation Model role, *set role* constraints can be used to mark *LEM_MTOS*, *LEM_MTM*, and *LEM_ATA inactive*. Setting these *LEMs* inactive is essential because there is no corresponding instance of the Operation Model role to compare with the instance of the Environment Model role.

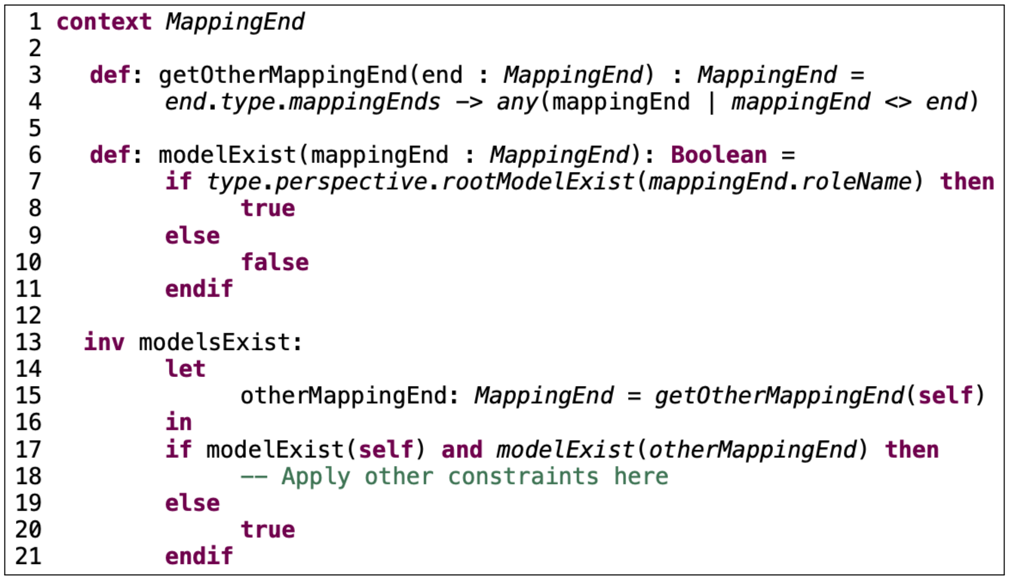
A snippet of a *set role* constraint with OCL, which evaluates whether there is a language role model, is shown in Figure 7.3. The OCL constraint (*modelsExist*) at line 13 checks that the language roles for both the current *mapping-end* (*self*) and the corresponding *mapping-end* (i.e., *otherMappingEnd*) have at least one model in a given perspective. Hence, other constraints can be applied if this condition is true (line 17), i.e., activating the effects of other *LEMs* constraints.

The *set role* constraint in Figure 7.3 uses two OCL helper functions (lines 3 - 11). The function *getOtherMappingEnd* (lines 3 - 4) retrieves the corresponding mapping-end of the mapping-end in question (*self*). On the other hand, the *modelExist*(*mappingEnd*) function (lines 6 - 11) checks if there is at least one model for the language role that contains the referenced language element from the *mappingEnd.* This function uses another helper function at line 7 (*rootModelExist*).

Recall that a perspective combines different languages for a modelling purpose. Hence, some of the OCL constraints written for the *PML* framework (e.g., *modelExist*) are expected to traverse the metamodels of the reused languages. However, to our knowledge, OCL does not support constraints that traverse instances of metaclasses from different metamodels. This OCL limitation implies that the function (*rootModelExist*), at line 7, cannot be defined within the *set role* constraints.

The *rootModelExist* function determines (with the role name) whether a root model of the concerned language role exists, and this operation requires access to the root metaclass of each language to perform its task. This function, however, cannot be implemented within the *set role* constraints because the language metamodels are separate from the *PML* metamodel.

To address this challenge, we define Java utility functions that are contained in the *PML* metamodel. These functions call the constraints of each language role in question and then return the result to the *set role* constraints. The *set role* constraints workflow is



```
context MappingEnd

  def: getOtherMappingEnd(end : MappingEnd) : MappingEnd =
        end.type.mappingEnds -> any(mappingEnd | mappingEnd <> end)

  def: modelExist(mappingEnd : MappingEnd): Boolean =
        if type.perspective.rootModelExist(mappingEnd.roleName) then
              true
        else
              false
        endif

  inv modelsExist:
        let
              otherMappingEnd: MappingEnd = getOtherMappingEnd(self)
        in
        if modelExist(self) and modelExist(otherMappingEnd) then
              -- Apply other constraints here
        else
              true
        endif
```

Figure 7.3: Invariant Constraint (root model exist)

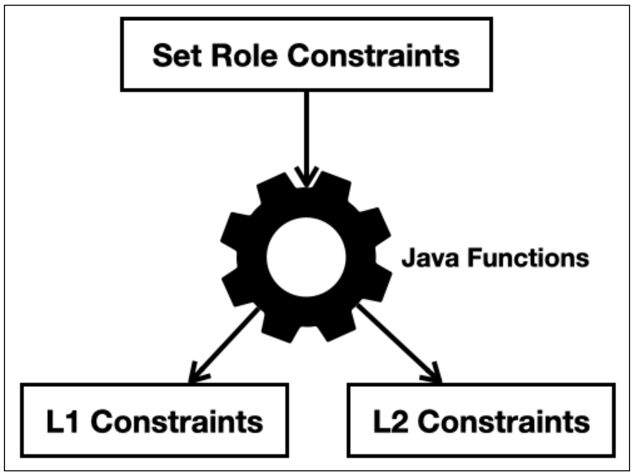


Figure 7.4: Set Role Constraints Workflow

shown in Figure 7.4. *Set role* constraints communicate with the Java functions, while the Java functions communicate with the corresponding individual language role constraints to evaluate the expected constraints. An example of a Java utility function is *rootModelExist* (line 7), which is defined in the `Perspective` metaclass.

With this definition of the *set role* constraint, *LEMs* that reference language roles with zero model are *inactive*. Hence, freeing the modeller from an unrealistic consistency management situation.

Set Role Constraint that Address Multi-Model Challenge

With a *multi-model* challenge, a perspective can proactively propagate the effects of a language action by randomly selecting one of the corresponding language role models. However, this random selection may not represent the desire of the modeller. To this effect, we proactively propagate the effects of a language action to corresponding language roles, provided that each corresponding language role has exactly one model. This approach implies that some constraint conditions in *LEMs* may be broken, e.g., creating a *message type* in the Environment Model role (see Figure 7.2) does not create any corresponding *message* if there is more than one Operation Model role instances in the perspective.

To accommodate the existence of broken consistencies in a perspective with a *set* role, we apply the *set role* constraints to display warning messages to users when some *LEMs* consistency rules are broken. With this warning approach, the user is required, for example, to create a *message* in his desired Operation Model role instance, which then automatically links with the existing *message type* and the broken consistency is resolved. Furthermore, this warning approach allows modellers to focus on a model while *PML* maintains consistencies in the background, especially with corresponding *regular* language roles, i.e., language roles that have exactly one model in a perspective.

A *set role* constraint that checks if an element is missing a compulsory mapping is shown in Figure 7.5. Line 2 represents the helper functions that are depicted in Figure 7.3. This snippet also demonstrates how we apply the *set role* constraint (*modelsExist*, see Figure 7.3) at line 7 in Figure 7.5. Lines 9 and 10 check if the corresponding mapping-end cardinality is compulsory (1) or compulsory-multiple (1..*), i.e., those mapping-ends that require to create the corresponding model element. At line 10, the constraint ensures that an instance of the referenced language element exists(*elementExist*) and it requires a *MEM* (*missingMapping*). Assuming that the current *LEM* is the *LEM_MTOS* in Figure 7.1, *missingMapping* returns *true* for each input *message type*, but returns *false* for each output *message type*. Hence, no warning is displayed for an output message even when the corresponding mapping-end multiplicity is compulsory. *missingMapping* is a Java utility function defined in the `MappingEnd` metaclass. However, if the current *LEM* is the *LEM_MTM* in Figure 7.1 and *missingMapping* returns *true* for an output *message type*, then a warning is displayed for the output message.

With this warning approach, perspectives can proactively propagate effects of a language action to the corresponding language role models. However, the perspective displays a warning message for each element that does not have a required *MEM*, which arises because the corresponding language plays a *set* role in the perspective.

7.3.2 PML with Model Multiplicity

In this section, we explain how we improve the *PML* framework to support the *model* multiplicity. We make two changes in the existing *PML* framework to support the *model* multiplicity as presented below.

```
context MappingEnd
  . . .
  inv missingCompulsoryMapping:
    let
      otherMappingEnd: MappingEnd = getOtherMappingEnd(self)
    in
    if modelExist(self) and modelExist(otherMappingEnd) then
           -- Apply other constraints here
           if (otherMappingEnd.cardinality = Cardinality::compulsory or
                  otherMappingEnd.cardinality = Cardinality::compulsoryMultiple) then
                  if elementExist(self) and missingMapping(self) then
                         false
                  else
                         true
                  endif
           else
                  true
           endif
    else
           true
    endif
```

Figure 7.5: Set Role Constraints (missing compulsory mapping)

First, we incorporate an additional attribute (*individualModelCardinality*) in the `MappingEnd` metaclass, see *PML* metamodel in Figure 4.7. This attribute allows a perspective designer to specify both the *regular* multiplicity (*cardinality*) as well as the *model* multiplicity (*individualModelCardinality*).

Second, we analyze the possible combinations between the *regular* multiplicity and the *model* multiplicity for each mapping-end, see Table 7.1. We realize that three cases need additional checks compared to the implementation of the *regular* multiplicity with the generic templates, see Chapters 4 and 6. The table shows all the valid combinations of *regular* multiplicity and *model* multiplicity per mapping-end. Note that a *model* multiplicity is required to be a subset of the corresponding *regular* multiplicity, e.g., the combination of *regular* multiplicity **0..1** and *model* multiplicity **1..*** is an invalid mapping-end.

The three special cases that require additional checks for *model* multiplicity are shown in rows 3, 4, and 6, in Table 7.1. For each case, we still apply the *regular* multiplicity, as detailed in Chapter 6, but now require to ensure that the corresponding *model* multiplicity

	Regular Multiplicity	Model Multiplicity	Effective Multiplicity
1	0..1	0..1	regular
2	1	0..1	regular
3	0..*	0..1	regular but model max 1
4	1..*	0..1	regular but model max 1
5	1	1	regular
6	1..*	1	regular but model max 1
7	0..*	0..*	regular
8	1..*	0..*	regular
9	1..*	1..*	regular

Table 7.1: Regular Multiplicity and Model Multiplicity

constraint is satisfied. The *model* multiplicity of each special case requires a maximum of 1, i.e., *model max 1* (see Table 7.1). On the other hand, rows 1 - 2, 5, and 7 - 9 do not require additional checks for the *model* multiplicity, because satisfying the *regular* multiplicity also (always) satisfies the *model* multiplicity.

Hence, to support *model* multiplicity in the *PML* framework, we apply the same algorithm for the regular multiplicity, but perform additional checks for the *model* multiplicity constraints when we have a mapping-end with the multiplicity combination that corresponds to row 3, 4, or 6.

7.3.3 Conditional Equivalency

Conditional equivalency constraints dictate model elements that are qualified to participate in a *MEM* based on a particular *LEM*, i.e., a model element can be used to establish a *MEM* with another element if the element in question satisfies the conditional equivalency constraints of the concerned *LEM*.

Example 7.3.1. *A `MessageType` with* INPUT *as the value of the* messageDirection *attribute is required to be mapped with a corresponding `OperationSchema` in the Operation Model role. On the other hand, an* OUTPUT *`MessageType` is required to be mapped with at least one `Message` in the Operation Model role. Hence, a `MessageType` can be qualified to be mapped with an `OperationSchema` or a `Message` in the Operation Model role depending on the value of its* messageDirection *attribute.*

Unlike the generic *set role* constraints, the conditional equivalency constraints are defined for a given *LEM* in a *perspective*. Hence, they are defined together with each *perspective* and are encapsulated within the *PML LEM*, see Figure 7.2, as well as the *PML* architecture in Figure 4.1. Since we use a generative approach to implement perspectives, implementation of these constraints is also generated. We use our domain-specific language to specify the *LEMs* as well as the conditional equivalency constraints, and then our code generator is applied to generate the implementation of the mappings and the conditional equivalency constraints. In the following, we illustrate how we apply conditional equivalency constraints to implement conditional *LEMs*.

Conditional *LEMs*

To address the conditional *LEMs* challenge, we use constraints to dictate the desired *LEM* for a specific *MEM*, which is inferred from the properties of the participating model elements.

To demonstrate the constraint, we consider LEM_MTOS and LEM_MTM *LEMs* (see Figure 7.2), an input *message type* is required to be mapped with an *operation schema*, while an output *message type* is required to be mapped with at least one *message* in the Operation Model role instances. This specification implies that an output *message type* may exist and is not mapped with an operation schema, although `MessageType` has a compulsory mapping-end (i.e., 1 multiplicity) with the `OperationSchema`. Conceptually, this violates the multiplicity constraints; however, an output *message type* does not need to be mapped with an *operation schema*.

To accommodate the two conflicting scenarios, we supplement *PML LEM* with constraints to determine whether a *LEM* is *active* or *inactive* for a given model element. Since we generate the constraint together with the implementation of the host *LEM*, the constraint dictates when the effects of a language action can be propagated based on the attribute values of the concerned model elements. An implementation of LEM_MTOS and LEM_MTM *LEMS* ensures that creating an input *message type* creates a corresponding *operation schema* (and then establishes a *MEM* between the two new elements) and does not create any corresponding *message* in the Operation Model role. On the other hand, the implementation ensures that the corresponding perspective action creates a *message* in the Operation Model role instance after creating an output *message type* and then creates a *MEM* between the two new elements.

Note that the input or output *message type* is deduced from the `Message` attribute (*messageDirection*) in the Environment Model (see Figure 4.2). Creating a *message* also creates a *message type* as a secondary effect. Hence, the *messageDirection* value can be used to determine if the new *message type* is an input or output message type.

While the constraints dictate which *LEM* to use to create a *MEM* between a pair of model elements, the *set role* constraints issue a warning when an element is missing a compulsory mapping.

Example 7.3.2. Set role *constraints display a warning message when an output* message type *is not mapped to an* operation schema *because* ***`MessageType`*** *has a corresponding compulsory mapping-end with the* ***`OperationSchema`****.*

Since this warning is not needed, we introduce a Java function that calls the generated implementation of the conditional equivalency constraint in question and then returns the result to the corresponding *set role* constraint. In this case, the result informs the *set role* constraint that a *MEM* is not required between an output *message type* and an *operation schema*. Hence, the *set role* constraints do not display a warning message for a model element when the concerned model elements do not satisfy the constraints, see Figure 7.5 and Section 7.3.1. The *missingMapping* function at line 11 is a Java function which calls the implementation of the respective constraints to ascertain whether the concerned element is truly missing a *MEM*.

7.4 Summary

This chapter presents advanced features of a perspective, which include *set roles* and conditional *LEMs*. A language plays a role in a perspective which can be either a *regular role* or *set role*. A *regular* language role indicates that there is exactly one instance of the language role exists in the perspective. However, a *set* language role designates that zero to many instances of the language role can exist in the perspective. On the other hand, conditional *LEM* comprises conditional equivalency constraint. A conditional equivalency applies the equivalence to a subset of model elements, which dictates the state (*active* or *inactive*) of a *LEM*, provided that at least one model exists for each corresponding language role.

These new features introduce more challenges in the *PML* framework, which include *zero model*, *multi-model*, and *insufficient mapping-end multiplicity* challenges, as well as filtering of model elements that qualify for a particular *MEM*. We address these challenges by introducing *language role* multiplicity, *model* multiplicity, and conditional *LEM*. The *language role* multiplicity dictates the allowable number of language role models. *Model* multiplicity establishes multiplicity constraints between a pair of models, each from a language role. The conditional *LEM* ensures that the right model elements are selected for a particular *MEM*

We implement the new features with OCL, which often communicates with a Java source code to retrieve an invariant result from a language role specific constraint. Conditional equivalency constraints are encapsulated within a *LEM* and a perspective designer can leverage our DSL to specify the constraints in a respective *LEM*. Furthermore, the designer can generate the implementation of the constraints, which frees the designer from the tedious and error-prone implementations.

In the following chapter, we present our DSLs and code generators. The DSLs allow perspective designers to register languages and then specify perspectives based on the registered languages. In addition, the code generators generate the implementation of both language registrations and perspective specifications.

Chapter 8

DSL for Languages and Perspectives

In this chapter, we describe the two *PML* DSLs, which allow perspective designers to register languages and then specify perspectives based on the registered languages. In addition, we present the code generator for each DSL, which is used to generate the implementation of the language registration as well as the complete implementation of each *perspective*. In the following, we provide a brief background of Xtext and Xtend, which are the language frameworks for the development of both the DSLs and the code generators, respectively. Furthermore, we explain how we use the two language frameworks to develop our DSL and code generators.

8.1 Xtext

Xtext is an open source Eclipse framework for the development of textual programming languages and domain-specific languages (DSLs) [52]. Xtext covers major aspects of a language infrastructure, which include, but are not limited to, the parser, interpreter, code markers, error description, syntax highlighting, and outline view. Xtext is a powerful tool that allows users to quickly define the grammar for their languages and automatically generate editors for such languages. Similar to metamodelling, language grammar represents the abstract syntax of the textual language. In the following subsections, we briefly describe three Xtext grammar rules that are relevant to this doctoral research work: terminal, type, and enumeration rules.

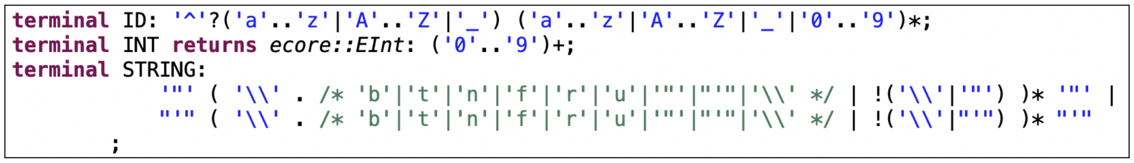
8.1.1 Terminal Rules

Terminal rule returns an atomic value, e.g., a string or an integer value. The terminal rule is also referred to as token rules or lexer rules, and, by convention, its names are written in uppercase after the keyword *terminal*. Xtext has some predefined terminal rules including ID, STRING, INT, and URI. Examples of terminal rule specifications (ID, INT, and STRING) are shown in Figure 8.1, which depend on the regular expressions to realize its values. We do not define new terminal rules in this work; however, we use most of the predefined terminal rules to implement our DSLs.

8.1.2 Type Rules

Similar to metamodelling, the type rule defines the concepts, as well as their relationships, that are required to implement a DSL. The type rule follows the object-oriented paradigm, and Xtext uses the type rule to derive metamodels as well as the modelling editors for the DSL under development. Each type rule generates a corresponding *class* in the metamodel of the language, and the name of the type rule corresponds to the name of the *class*. In addition, a type rule can contain keywords, terminals, assignments (i.e., compositions), and cross references (i.e., associations).

A type rule has two main categories of operators: abstract syntax operator and concrete



```
terminal ID: '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING:
            '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
            "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
        ;
```

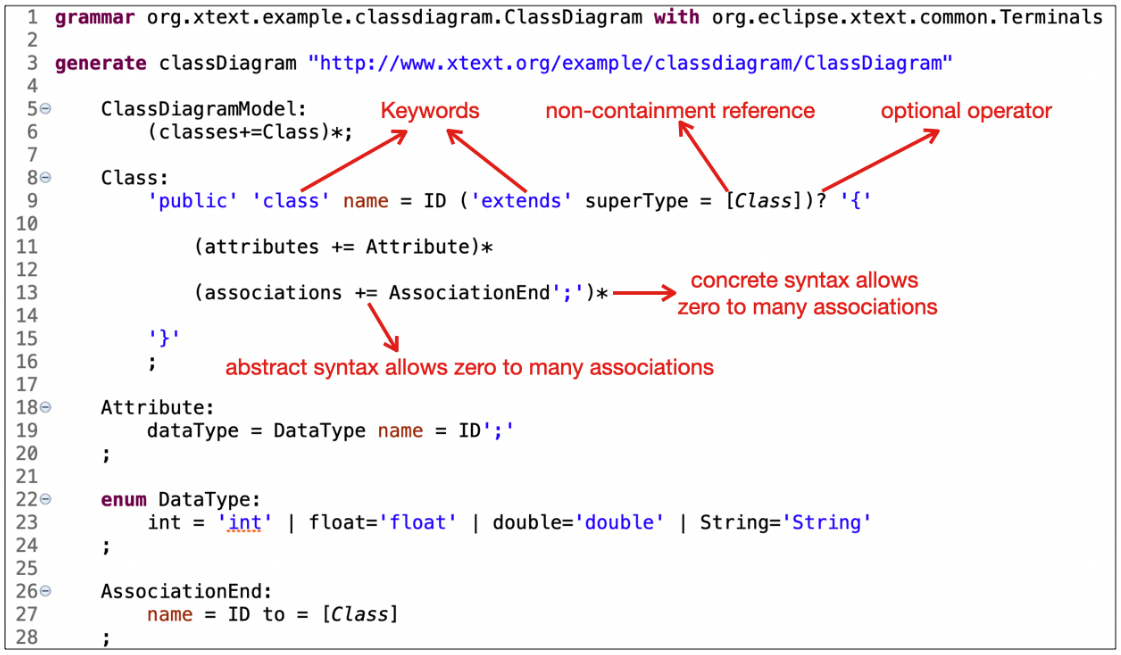
Figure 8.1: Examples of Xtext Terminal Rules

syntax operator. The abstract syntax dictates the multiplicity relationship between two type rules, which are += and =. The += operator allows zero, one, or many elements while = allows zero or one element. On the other hand, concrete syntax operators include * (zero to many), + (one to many), ? (zero or one), and a white space (one element).

Figure 8.2 shows a sample grammar (excerpt) for the implementation of the *class diagram* language (see Figure 3.4). Lines 8 to 16 define a type rule (*Class*). A *Class* can contain several *attributes* (line 11) and *associations* (line 13), as well as reference another *Class* as a *superType* (line 9). From line 18 to 28, the grammar contains two additional type rules (*Attribute* and *AssociationEnd*) and one enumeration rule *DataType* (see the subsection below). The *Attribute*, *AssociationEnd*, and *DataType* rules are referenced from the *Class* type rule. The generated metamodel (abstract syntax) from the gammar is shown in Figure 8.3

8.1.3 Enumeration Rules

The enumeration rule allows language designers to derive enumeration literals from string values. The Enumeration rule is commonly used to create data types with specific values. Figure 8.2 (lines 22-24) shows an example of an enumeration rule.



```
grammar org.xtext.example.classdiagram.ClassDiagram with org.eclipse.xtext.common.Terminals

generate classDiagram "http://www.xtext.org/example/classdiagram/ClassDiagram"

    ClassDiagramModel:
        (classes+=Class)*;

    Class:
        'public' 'class' name = ID ('extends' superType = [Class])? '{'

            (attributes += Attribute)*

            (associations += AssociationEnd';')*

        '}'
        ;

    Attribute:
        dataType = DataType name = ID';'
    ;

    enum DataType:
        int = 'int' | float='float' | double='double' | String='String'
    ;

    AssociationEnd:
        name = ID to = [Class]
    ;
```

Keywords
non-containment reference
optional operator
concrete syntax allows zero to many associations
abstract syntax allows zero to many associations

Figure 8.2: Class Diagram Sample Grammar (excerpt)

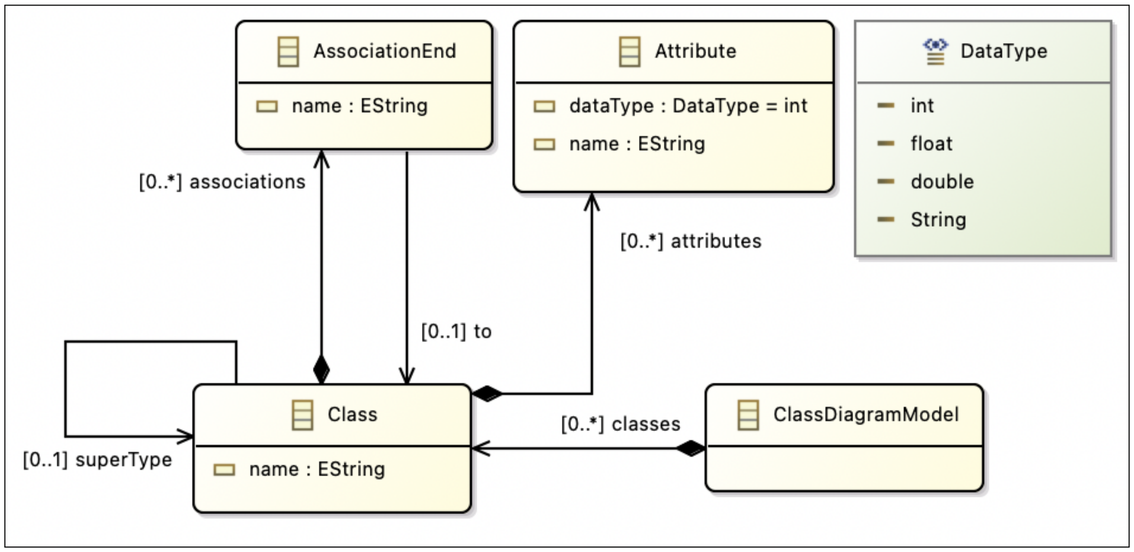


Figure 8.3: Generated Class Diagram Metamodel

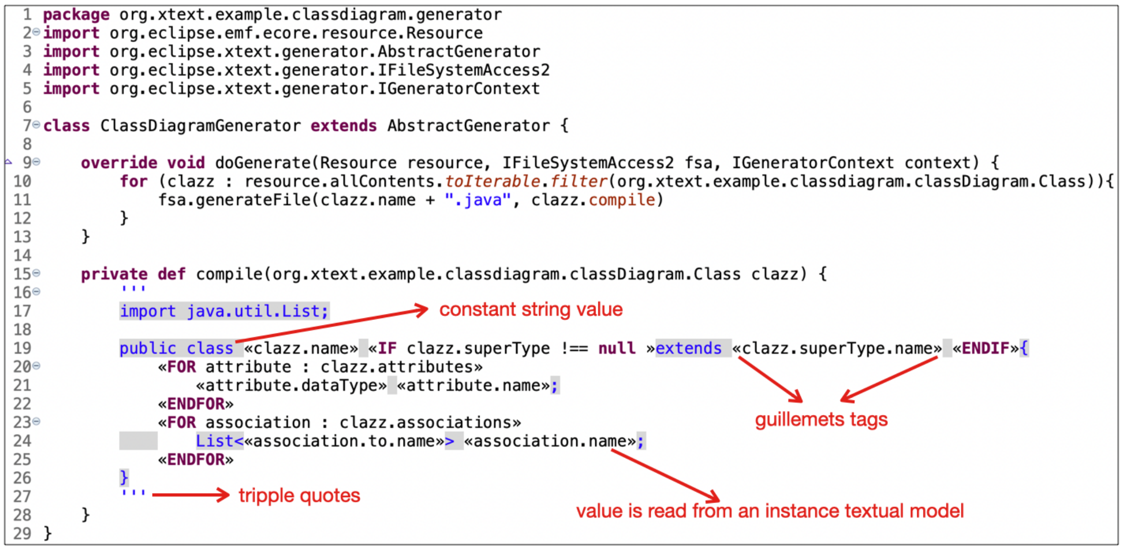
8.2 Xtend

Xtend is a statically typed textual programming language that compiles to the Java source code [53]. Similar to Java, Xtend supports several programming features, including the template-based code generator, extension methods, lambda expressions, generics, annotations, and operator overloading. In this work, we focus on the template-based code generator, which we use to generate the implementation of language registrations and perspectives (including perspective actions).

Figure 8.4 shows an Xtend class, which contains the template specification for the definition of a code generator for the class diagram language (see Figure 8.2). The Xtend class contains the *doGenerate* method (similar to Java *main* method), which creates the file that will contain the generated source code (lines 9 - 13)

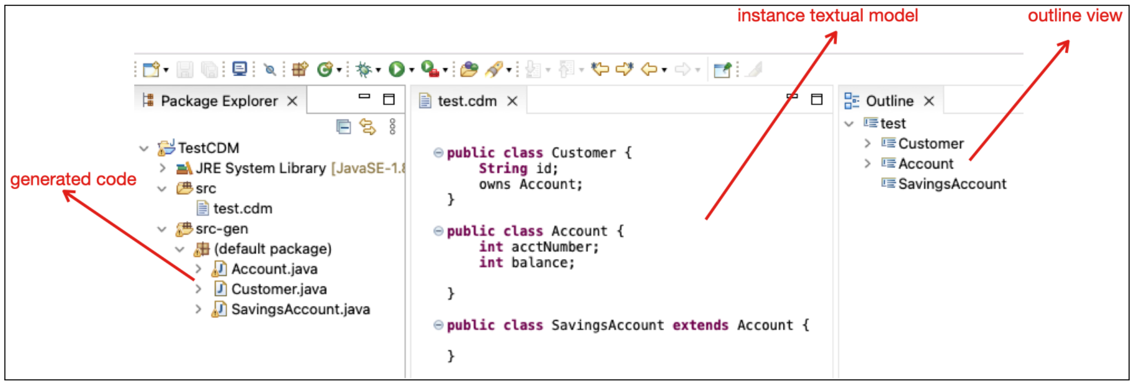
Xtend template is enclosed with *triple quotes* (see Figure 8.4) and supports string concatenation with white space. Dynamic values are injected with interpolation expression, which is expressed with guillemets tags (see Figure 8.4). Expressions in an Xtend template can span multiple lines and an expression can be contained in another expression. String characters that are contained in a template but not in an expression are generated without any modification.

An annotated template-based code generator for the *class diagram* language (see Figure 8.2) is shown in Figure 8.4. The code generator generates a Java file for each *class* defined in the *class diagram* language (lines 9 - 13). Lines 17 and 19 contain some string literals, as well as variables such as the class name. Lines 20 to 22 iterate over all the contained attributes of the *class* in question, and then generate an equivalent Java definition for each *attribute* (line 21). Similarly, lines 23 to 25 iterate over all the association ends from the current class to another referenced class, and then generate an



```
package org.xtext.example.classdiagram.generator
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext

class ClassDiagramGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
        for (clazz : resource.allContents.toIterable.filter(org.xtext.example.classdiagram.classDiagram.Class)){
            fsa.generateFile(clazz.name + ".java", clazz.compile)
        }
    }

    private def compile(org.xtext.example.classdiagram.classDiagram.Class clazz) {
        '''
        import java.util.List;

        public class «clazz.name» «IF clazz.superType !== null »extends «clazz.superType.name» «ENDIF»{
            «FOR attribute : clazz.attributes»
                «attribute.dataType» «attribute.name»;
            «ENDFOR»
            «FOR association : clazz.associations»
                List<«association.to.name»> «association.name»;
            «ENDFOR»
        }
        '''
    }
}
```

Figure 8.4: Class Diagram Sample Code Generator



```
public class Customer {
    String id;
    owns Account;
}

public class Account {
    int acctNumber;
    int balance;

}

public class SavingsAccount extends Account {

}
```

Figure 8.5: Class Diagram Sample Model

equivalent Java definition for each *association end* (line 24).

With the definition of both grammar and code generator for the *class diagram* language, a software modeller can now use the language to define textual class diagram models and then generate code from the instance model. The *class diagram* language editor (see Figure 8.5) encompasses the textual model (instance of the DSL, see Figure 8.2), the generated code (instance of the code generator, see Figure 8.4, and the outline view. In the next section, we present our DSLs and code generators, which are developed with Xtext and Xtend frameworks, respectively.

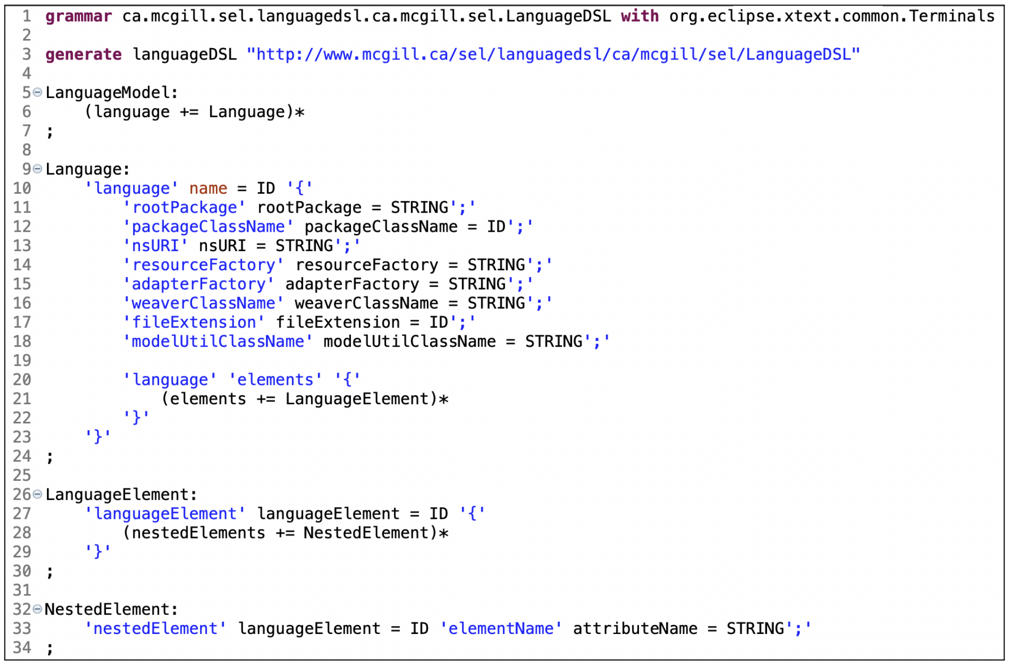
8.3 Definition of PML DSLs and the Code Generators

In this section, we explain how we apply the Xtext language workbench to define the grammar for both *language registration* and *perspective* DSLs. In addition, we present the code generators for both DSLs.

8.3.1 DSL Definition for Language Registration

To register languages in a tool that supports *PML*, e.g., TouchCORE, we use the Xtext workbench, as detailed above, to develop the *language registration* DSL. The DSL allows perspective designers to register languages so that their perspectives can reuse the registered languages. Details of the DSL grammar are shown in Figure 8.6

The DSL grammar captures the details required to register languages in a multi-language software tool. To register a language, the perspective designer is required to instantiate part of the *PML* metamodel (Figure 4.7), which contains the desired concepts, as well as their attributes for each language, i.e., `ExternalLanguage` and `LanguageElement`



```
 1 grammar ca.mcgill.sel.languagedsl.ca.mcgill.sel.LanguageDSL with org.eclipse.xtext.common.Terminals
 2 
 3 generate languageDSL "http://www.mcgill.ca/sel/languagedsl/ca/mcgill/sel/LanguageDSL"
 4 
 5 LanguageModel:
 6     (language += Language)*
 7 ;
 8 
 9 Language:
10     'language' name = ID '{'
11         'rootPackage' rootPackage = STRING';'
12         'packageClassName' packageClassName = ID';'
13         'nsURI' nsURI = STRING';'
14         'resourceFactory' resourceFactory = STRING';'
15         'adapterFactory' adapterFactory = STRING';'
16         'weaverClassName' weaverClassName = STRING';'
17         'fileExtension' fileExtension = ID';'
18         'modelUtilClassName' modelUtilClassName = STRING';'
19 
20         'language' 'elements' '{'
21             (elements += LanguageElement)*
22         '}'
23     '}'
24 ;
25 
26 LanguageElement:
27     'languageElement' languageElement = ID '{'
28         (nestedElements += NestedElement)*
29     '}'
30 ;
31 
32 NestedElement:
33     'nestedElement' languageElement = ID 'elementName' attributeName = STRING';'
34 ;
```

Figure 8.6: Grammar Definition for Language Registration

metaclasses and their attributes. Of course, the perspective designer can implement a language registration manually; however, this approach is tedious and error-prone. To this effect, the DSL aims to streamline this registration process with a textual programming language that allows a perspective designer to provide language details which can be used to generate the implementation of the language registration. As shown in Figure 8.6, we define a *type rule LanguageModel* (lines 5 - 7), which contains a set of languages, since the DSL can be used to capture several languages. For each language *type rule* (lines 9 - 24), we define the attributes that are required to instantiate the concerned part of the *PML* metamodel and generate the implementation of a language registration (lines 10 - 18). As shown in the *PML* metamodel, an external language (or language here) contains a set of language elements; this containment relationship is defined in the DSL grammar in lines 20 - 22. Furthermore, the referenced language element, i.e., a *type rule*, from the *Language* type rule, is defined in lines 26 - 30. And finally, a nested language element *type rule* is defined in lines 32 - 34.

We believe that creating instances of this DSL grammar to register languages with an automated code generator reduces the effort required to implement *language registration* manually. Additionally, a code generator approach often produces more error-free implementations compared to the manual approach. Several instances of the language registration grammar are contained in Appendix C.

8.3.2 Code Generator for Language Registration

To generate the implementation of a language registration, i.e., to instantiate part of the *PML* metamodel, which is related to the external languages, we use the Xtend framework to define the code generator, which is based on the grammar of the language registration DSL (see Figure 8.6). Hence, the code generator, i.e., a text-to-text model transformation, accepts an instance of the DSL (*source* model) to produce a Java source code (*target* model) for each *language registration.* The Java source code implements the language registration, and the complete details of the code generator are available in the PML GitHub repository.

8.3.3 DSL Definition for Perspectives

The perspective DSL allows a perspective designer to reuse registered languages, specify *LEMs* between different language elements from different languages, and generate the corresponding perspective actions for each *LEM.* A perspective designer can manually implement the perspectives, but it is prone to errors and a daunting task. On the other hand, a perspective designer can leverage our DSL, which comes with a code generator, to create perspectives. The code generator generates the complete implementation of a perspective, including the *redefined* perspective actions. This DSL and the generative approach aim to reduce perspective implementation efforts and potentially eliminate errors in the generated code.

An excerpt of the grammar for the development of the perspective DSL, using the Xtext language workbench, is shown in Figure 8.7. This grammar aims to capture all the details to instantiate the *perspective* metamodel, which can be used to implement the perspective, including the perspective actions that manage consistencies between model elements. An instance of this DSL grammar can contain zero to many perspectives. Hence, we define a type rule *PerspectiveModel*, which contains a set of perspectives that can be integrated into a tool (lines 7 - 9).

Furthermore, we define the *Perspective* type rule (lines 11 - 43), which allows a perspective designer to capture the details required to implement a perspective, including reusing registered languages. To distinguish between different perspectives in a tool, we require a perspective designer to specify a unique name for each perspective (line 13). A perspective combines different languages, each playing one or more roles in the perspective. A perspective designer can dedicate one of the language roles as a *default* role, e.g., the class diagram being the default role for the *class diagram and sequence diagram* perspective. A default role gives the corresponding model a higher priority during navigation, i.e., in a multi-language perspective system, a model corresponding to the default language role is presented first. However, the user can navigate to other models from the default model. Hence, we define a *default* language role at line 14, which allows a perspective designer to specify a *default* language role for his perspectives. This *default* role name must be the same as one of the language roles; otherwise, the *default* role is invalid.

A *perspective*, as well as language models, often changes when a user is developing

```
grammar ca.mcgill.sel.perspectivedsl.ca.mcgill.sel.PerspectiveDSL
with org.eclipse.xtext.common.Terminals

generate perspectiveDSL
"http://www.mcgill.ca/sel/perspectivedsl/ca/mcgill/sel/PerspectiveDSL"

PerspectiveModel:
    (perspectives += Perspective)*
;

Perspective:
    'perspective' '{'
        'name'':' name = STRING';'
        ('default'':' isDefault = [RoleName]';')?

        ('savePerspective'':' savePerspective = STRING';')?
        ('saveModel'':' saveModel = STRING';')?

        'currentPerspective' ':' currentPerspective = STRING';'
        'currentRoleName' ':' currentRoleName = STRING';'

        ('model' 'factory' 'facade' 'calls' '{'
            (modelFacades += FacadeCall)*
        '}')?

        'role' 'names' '{'
            (roleNames += RoleName)*
        '}'

        'model' 'cardinalities' '{'
            (modelCardinalities += ModelCardinality)*
        '}'

        'languages' '{'
            (languages += Language)*
        '}'

        ('mappings' '{'
            (mappings += LanguageElementMapping)*
        '}')?

    '}'
;
```

Figure 8.7: Part of the Perspective DSL Grammar

language models. For example, creating a *MEM* modifies a *perspective*, while calling a language action to create, delete, or update a model element modifies the corresponding language model. Hence, to support persistence of perspectives and language models, we encode two properties for the *Perspective* type rule, i.e., *savePerspective* and *saveModel* in lines 16 - 17. These two optional attributes allow a perspective designer to provide API calls that save the current perspective and the current model, respectively. In addition, for a streamlined modelling workflow and communication between the modelling tool and the perspectives, a multi-language modelling environment requires to uniquely identify current perspective, i.e., the current perspective being used by a user, and the current model, i.e., the current model the user is elaborating on. Hence, a perspective designer is required to encode two API calls that can be used to retrieve the current perspective and the current model (lines 19 - 20), respectively. Note that a role name is used here to retrieve the corresponding model that plays the role.

To manage the template workflow (see Figure 6.1) between a root model element and a regular model element, i.e., model elements other than the root element, a perspective designer is required to specify the facade actions (lines 22 - 23). Other facade actions, i.e., the facade actions between regular model elements, are contained in the corresponding *redefined* perspective actions, which are presented in the complete grammar definition (see Appendix B). Note that *modelFacades* (line 23) is not contained in the *redefined* perspective actions, which are specified within the reused language, because generating the implementation for creating and managing consistencies involving root model elements is handled at the perspective level. This is the case since root model elements require dedicated handling because creating each root model requires the tool to configure the model persistence and then link the model with the corresponding language role in the perspective. This is quite different from creating a regular model element with an existing root model. The complete DSL grammar is shown in Appendix B, which contains the model factory and facade details.

Furthermore, a perspective designer is required to define a set of role names (lines 26 - 28), each referencing a reused language (see Figure 4.7). Also, to support perspectives with *set role*, the grammar allows a perspective designer to specify the model cardinalities for each language; see the complete grammar in Appendix B to understand the definition of the *ModelCardinality* type rule. And finally, a perspective designer can reuse registered languages (lines 34 - 36) and then specify *LEMs* between language elements across language boundaries (lines 38 - 40). Appendix B contains the full definition of all the type rules and enumerations used in the grammar. In addition, an instance of the perspective DSL grammar is contained in Appendix D, which details the definition of a *perspective* that combines five different modelling languages.

8.3.4 Code Generator for Perspective Implementation

The perspective code generator generates the complete implementation of a perspective based on an instance of the perspective DSL, see the previous subsection. The code generator instantiates some parts of the *PML* metamodel including `Perspective`, `LanguageElementMapping`, and `MappingEnd`. In addition, the code generator applies the generic templates (see Chapter 6) to generate the implementation of perspective actions (*redefined* create perspective actions and *redefined* delete perspective actions). Note that we do not generate the *update* perspective action because the implementation does not change across different perspectives.

For each perspective, we generate several Java classes depending on whether the perspective is a *single-language* or a *multi-language* perspective. The *doGenerate* method for the implementation of perspectives is shown in Algorithm 1. The *doGenerate* method iterates through all the perspectives that are specified in the perspective DSL, lines 1 - 11. For each perspective, the code generator generates the perspective Java file, including the *LEMs*, at line 3.

Algorithm 1: doGenerate(Resource resource)

```
1  for perspective : resource.perspectives do
2      // creates perspectives
3      generateFile(PerspectiveName.java, perspective.compile)
4      for language : perspective.languages do
5          if containsRedefinedAction(perspective, language) then
6              // creates redefined perspective action
7              generateFile(RedefinedPerspectiveAction.java,
                 RedefinedAction.compileActions(perspective, language)
8              // creates the corresponding facade action
9              generateFile(FacadeAction.java,
                 FacadeActionGen.compileFacadeActions(perspective, language)
10         // creates the model factory
11         generateFile(ModelFactory.java,
             ModelFactory.compileCreateModel(perspective)
```

To implement perspective actions, our template-based code generator follows the workflow of our generic templates shown in Figure 6.1. For each language (see lines 4 -9 in Algorithm 1), we generate a Java file (line 7) that implements all the *redefined* perspective actions for the language in question, and then we generate another Java file for the corresponding facade actions (line 9). Both the *redefined* perspective actions and the facade actions are generated for languages that have *LEMs* in a perspective. However, a *single-language* perspective does not have any *LEM*; therefore, neither *redefined* perspective actions nor facade actions are required to implement a *single-language* perspective.

Furthermore, for both the *single-language* and *multi-language* perspectives, we generate the *model factory* Java class (line 11), which is responsible for managing the creation of root model elements for each reused language in a perspective. In addition, the *model factory* factory maintains all the consistencies between the root model elements, as well as the consistencies between the regular model elements and the root model elements. The complete definition of the code generator, including the Java classes, is available in the PML GitHub repository.

8.4 Summary

This chapter presents the two DSLs, which allow perspective designers to register languages and then specify perspectives based on the registered languages. In addition, we present the code generator for each DSL, which is used to generate the implementation of the language registration, as well as the complete implementation of each perspective.

First, we provide a brief background of Xtext and Xtend, which are the language frameworks for the development of both the DSLs and the code generators, respectively. Xtext is a language workbench for the development of domain-specifc languages, which allows users to quickly define language grammar and automatically generate the language editors. On the other hand, the Xtend framework is a statically typed textual programming language that compiles to Java source code. We leverage the Xtend template-based feature to implement the code generators for our DSLs. And finally, we explain how we use both the Xtext and Xtend to develop our DSLs and the code generators.

In the following chapter, we present the evaluation of our study with different real-life multi-language methodologies. Furthermore, we investigate the navigation facilities of several popular modelling tools and compare them with the *PML* generic navigation based on selected navigation features.

Chapter 9

PML Validation

In this chapter, as a proof-of-concept, we carry-out different feasibility studies to ascertain the possibility of using our framework to register languages, define perspectives, and build models according to the respective perspectives. In this regard, we validate our framework with three notable multi-language modelling methodologies. First, we present a *Fondue Requirement* perspective, which combines five different modelling languages with the TouchCORE tool. Second, we present a multi-language *perspective* that targets component modelling. Here, we illustrate our approach with the *Palladio Component Model* (PCM), a group of four modelling languages that specialize in performance prediction of distributed architectures. Third and the last, we evaluate *PML* with another popular multi-language methodology, the *User Requirements Notation* (URN), which combines three different modelling languages for elicitation, specification, and analysis of software requirements.

Furthermore, we analyse the navigation facilities of several popular modelling tools and evaluate whether our navigation mechanism can support the discovered navigation facilities. In addition, we investigate selected modelling tools, including ArgoUML, StarUML, MagicDraw, Visual Paradigm, and Papyrus, to discern how *PML* navigation stands out compared to other navigation mechanisms.

The following sections present the three different multi-language modelling environments and then the navigation comparative study.

9.1 Fondue Requirement Perspective

In this section, we illustrate our framework with a real-world multi-language perspective that oversees the consistency of requirements models that are built when using the model-driven software development methodology Fondue [54]. Fondue is a further development of the second-generation object-oriented software development methodology called Fusion [55] developed and used by Hewlett Packard in the mid 90's. Furthermore, Fondue groups many languages with complex relationships based on UML models, which showcase all the features that we cover in this research work. The perspective combines a class diagram, a use case diagram, an environment model (communication diagram), a use case maps model, and several operation models (descriptions of system behaviour) for the purpose of requirement elicitation and specification. The modeller uses a class diagram language to capture the concepts of the problem domain as well as their relationships, and the expected interactions between the environment and the system with a use case diagram. Furthermore, the environment model defines the system's interface, i.e., the boundaries of the system and the operations that can be performed on the system (`in` messages called *system operations*) and the outputs produced by the system (`out` messages). The use case maps model defines the allowable sequences of interactions that the system may have with its environment over its lifetime, while the operation models describe the desired effect of each system operation on the conceptual state of the system. Figures 3.4 and 4.2 to 4.5 show the metamodel excerpts for the modelling languages and some of their corresponding language actions. It is important in this case that the five different types of models (views) of the system are coherent. Also, to promote maintainability and modularity, each language should be allowed to evolve independently, i.e., without direct links to other languages in the modelling environment.

In this example, we demonstrate how a perspective designer can leverage our *PML* framework to *specify*, *capture*, and then *maintain* the consistency conditions (equivalency, equality, and multiplicity constraints) between different model elements in each model. A perspective groups different languages for a modelling purpose. Hence, the first step is to define the purpose of the perspective as shown in Section 9.1.1. With a well defined purpose, the perspective designer then registers different languages, which will collaborate to fulfil the purpose of the system as detailed in Subsection 9.1.2. In Subsection 9.1.3, we briefly explain how the designer specifies the *perspective*, which combines the registered languages to satisfy the aim of the system, while Subsection 9.1.4 details how to instantiate the *perspective* and then integrate it with a software tool. Finally, Subsection 9.1.5 discusses the relevance of perspective actions in the multi-language perspective.

9.1.1 Purpose of Fondue Requirement Perspective

The first role of the perspective designer is to define the purpose of the perspective. A perspective's modelling purpose encompasses several consistency conditions which need to be maintained in the modelling environment. In this example, the *fondue requirement perspective*, the designer aims to maintain the following equivalency and multiplicity constraints in the multi-language modelling environment. Furthermore, for each of the equivalency constraints, an equality constraint ensures that the names of the elements are synchronized.

- **R1:** Each actor type from the environment model must have a corresponding actor class outside the system boundary in the domain model, i.e., class diagram model.
- **R2:** Each input message in the environment model must have a corresponding operation modelled by an operation schema in the operation model and vice versa.
- **R3:** Each output message in the environment model must have at least one corresponding message that is generated from within one of the operation models.
- **R4:** An actor that appears in the operation model must be part of the environment model and an actor type from the environment model must have at least one corresponding actor in the operation model.
- **R5:** All classes used in the scope of the operation model must be part of the domain model.
- **R6:** Every system operation must appear at least once in the use case maps.
- **R7:** Every input and output message in the environment model must appear at least once in the use case maps as a responsibility reference and each responsibility reference must have a corresponding message.
- **R8:** Every actor in the use case diagram must appear as an actor class outside the system boundary in the domain model.

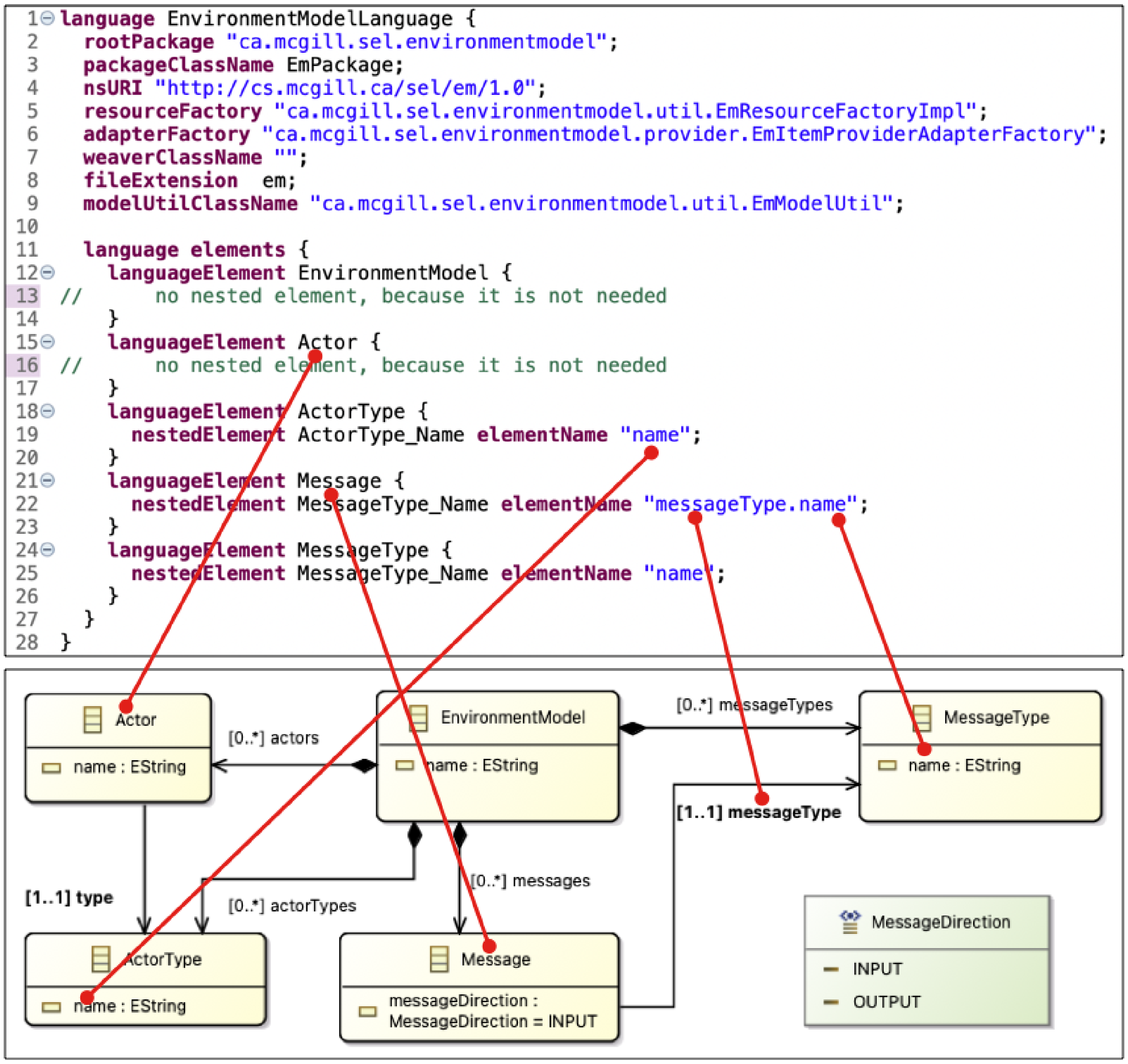
9.1.2 Language Registration

The second step is then to register those modelling languages in the multi-language modelling environment whose combination is capable of fulfilling the consistency conditions specified above. This step is performed once for each language. Hence, registered languages can be reused across different perspectives. An annotated sample language registration with our DSL is shown in Figure 9.1, which captures the details of the environment model language (lines 2 - 9) and its elements (lines 11 - 27).

The language definition encodes the packages for importing the language elements (line 2, i.e., the metaclasses of the language metamodel). The package class (line 3, i.e., the container of the language elements) allows the perspective to retrieve the object representation of each language element from the corresponding metamodel. Also, the language details in lines 4 - 7 are primarily used to register the language metamodel and its resources in the multi-language modelling environment. The language elements compartment shows the name of each required language element and optional nested elements; e.g., the *ActorType* language element has a name attribute as its nested element.

A nested language element can refer to an attribute, which is contained in another language element, provided there is a navigable link from the language element in question to the nested element. For example, the nested element of `Message` is the name attribute contained in the `MessageType`. This is the case since a message is visualized to a user with the name of its type. Figure 9.1 shows the correspondences between the language registration specification and the metamodel of the language.

With this specification, a *perspective* designer can register the language in the desired modelling environment tool, e.g., TouchCORE, and then reuse the registered languages across several perspectives. This modular registration of the languages, i.e., each language is



```
language EnvironmentModelLanguage {
  rootPackage "ca.mcgill.sel.environmentmodel";
  packageClassName EmPackage;
  nsURI "http://cs.mcgill.ca/sel/em/1.0";
  resourceFactory "ca.mcgill.sel.environmentmodel.util.EmResourceFactoryImpl";
  adapterFactory "ca.mcgill.sel.environmentmodel.provider.EmItemProviderAdapterFactory";
  weaverClassName "";
  fileExtension  em;
  modelUtilClassName "ca.mcgill.sel.environmentmodel.util.EmModelUtil";

  language elements {
    languageElement EnvironmentModel {
//        no nested element, because it is not needed
    }
    languageElement Actor {
//        no nested element, because it is not needed
    }
    languageElement ActorType {
      nestedElement ActorType_Name elementName "name";
    }
    languageElement Message {
      nestedElement MessageType_Name elementName "messageType.name";
    }
    languageElement MessageType {
      nestedElement MessageType_Name elementName "name";
    }
  }
}
```

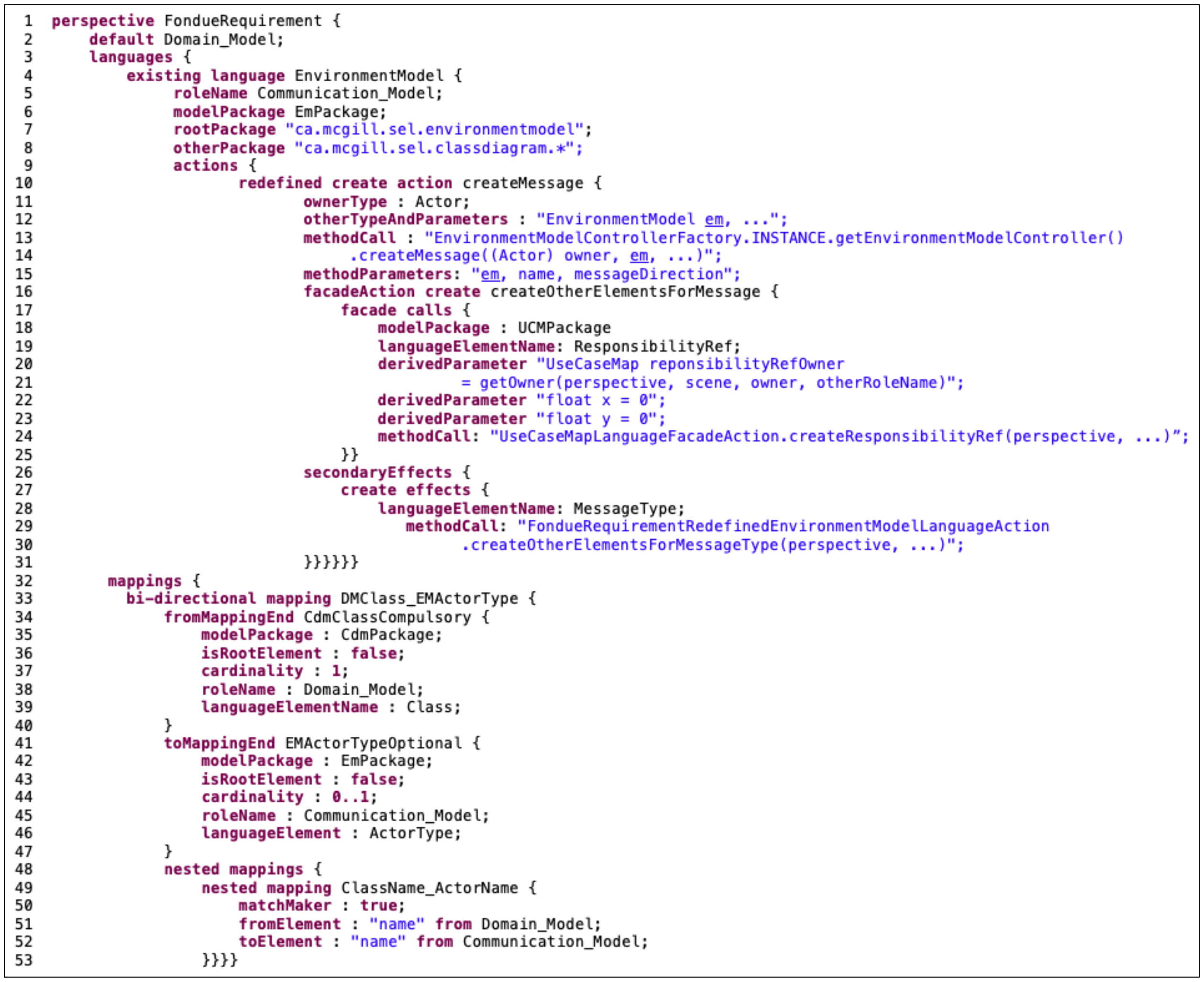
Figure 9.1: Annotated Environment Model Language DSL Model

registered independently of other languages, promotes the reusability as well as the evolution of the languages, because each language can be updated without facing the complexities of a combined metamodel. To reuse each language in a *perspective*, the perspective establishes *LEMs* between language elements across language boundaries. Also, the perspective can then *re-expose*, *hide*, or *redefine* the language actions, i.e., the externally defined language actions. A perspective captures the details of these language actions (see `LanguageAction` metaclass in Figure 4.7) so that the generic templates can be applied to generate the implementation of the concerned perspective actions. A similar specification exists for the other languages (i.e., the use case diagram, the class diagram, the use case maps, and the operation model). Appendix C shows the complete language registration specifications.

9.1.3 Specification of the Perspective

This section demonstrates how a perspective designer employs our DSL to specify a *perspective* with the registered languages. With a specified perspective, the perspective actions are fully generated based on the generic templates (see Sections 6.2 and 8.3.4) and the language specification (see Figure 9.1).

In Figure 9.2, the perspective designer provides information to define the perspective, perspective actions, and language element mapping(s). In this example, the default language role in the perspective is the *domain model* (line 2), which dictates the model to view by default. To create a multi-language perspective, the perspective combines a class diagram, a use case diagram, an environment model, use case maps, and operation models (lines 3 - 31). As an example, the specification shows how the *perspective* reuses the registered environment model language. The specification of the other languages is similar, and the complete definition of the perspective with our DSL is shown in Appendix D.



```
perspective FondueRequirement {
    default Domain_Model;
    languages {
        existing language EnvironmentModel {
             roleName Communication_Model;
             modelPackage EmPackage;
             rootPackage "ca.mcgill.sel.environmentmodel";
             otherPackage "ca.mcgill.sel.classdiagram.*";
             actions {
                    redefined create action createMessage {
                           ownerType : Actor;
                           otherTypeAndParameters : "EnvironmentModel em, ...";
                           methodCall : "EnvironmentModelControllerFactory.INSTANCE.getEnvironmentModelController()
                                .createMessage((Actor) owner, em, ...)";
                           methodParameters: "em, name, messageDirection";
                           facadeAction create createOtherElementsForMessage {
                               facade calls {
                                   modelPackage : UCMPackage
                                   languageElementName: ResponsibilityRef;
                                   derivedParameter "UseCaseMap reponsibilityRefOwner
                                            = getOwner(perspective, scene, owner, otherRoleName)";
                                   derivedParameter "float x = 0";
                                   derivedParameter "float y = 0";
                                   methodCall: "UseCaseMapLanguageFacadeAction.createResponsibilityRef(perspective, ...)";
                               }}
                           secondaryEffects {
                               create effects {
                                   languageElementName: MessageType;
                                      methodCall: "FondueRequirementRedefinedEnvironmentModelLanguageAction
                                            .createOtherElementsForMessageType(perspective, ...)";
                           }}}}}}
      mappings {
        bi-directional mapping DMClass_EMActorType {
            fromMappingEnd CdmClassCompulsory {
                modelPackage : CdmPackage;
                isRootElement : false;
                cardinality : 1;
                roleName : Domain_Model;
                languageElementName : Class;
            }
            toMappingEnd EMActorTypeOptional {
                modelPackage : EmPackage;
                isRootElement : false;
                cardinality : 0..1;
                roleName : Communication_Model;
                languageElement : ActorType;
            }
            nested mappings {
                nested mapping ClassName_ActorName {
                    matchMaker : true;
                    fromElement : "name" from Domain_Model;
                    toElement : "name" from Communication_Model;
                }}}}
```

Figure 9.2: Fondue Requirement Perspective

For each language reuse, the perspective designer first defines the role of the language in the perspective at line 5, e.g., the environment model plays the role of a communication model. Furthermore, the designer specifies the model package class, i.e., the container of the language elements, which is used to retrieve the corresponding object representation of the language elements during the generation of the *redefined* perspective actions, (line 6), and the root package (line 7). The *otherPackage* (lines 8) allows the *redefined* perspective action (e.g., *createMessage*) to interact with other reused language actions in the perspective. Other referred packages can be specified after line 8, see Appendix D for the complete definition of the perspective. The specifications in lines 5 - 8 allow the perspective to access and then reuse the corresponding registered language.

Next, the perspective designer explicitly specifies the *redefined* and *re-exposed* perspective actions, as well as implicitly any hidden actions. As shown in Figure 9.2, lines 9 to 31 depict a perspective action that redefines the *createMessage* language action in the environment model language. For each *redefined* perspective action (line 10), the designer specifies the type of an element that will contain the new element (line 11), the parameters with their types, excluding the element container (line 12), the API for the language action (line 13), and the method parameters, i.e., the parameters without their types (line 15). These details are used to generate the *redefined* perspective action based on the generic template shown in Figure 6.4 on page 115.

Each *redefined* action reuses at least one other language action (see *reusedActions* association in Figure 4.7). We implement the reused actions as facade actions (lines 16 - 25). Hence, the *redefined* perspective action augments both the *redefined* language action, e.g., *createMessage* in the environment model, and the other reused actions so that the consistency conditions of the concerned language elements are maintained. In this example, the designer specifies that when a *message* is created in the environment model, at least one *responsibility reference*, as specified in Section 9.1.1 (R7), is created and then mapped (i.e., *MEM*) with the *message*.

To fully generate the implementation of this *redefined* perspective action, the designer further specifies the model package class (line 18), as well as the language element (line 19), of the other element. At line 20, the perspective designer encodes the parameters of the other element, i.e., deriving the parameters that are required to create the *responsibility reference*. The parameters can be derived from the initial parameters of the *createMessage* action. The *derivedParameter* accommodates any lines of string values which represent valid Java code. The designer can provide other parameter values which cannot be derived from the original parameters by using valid literal value(s) (lines 22 - 23) or asking the modeller to provide the needed parameter value. And finally, for each reused action, the language action is called, at line 24, to create the model element in question, e.g., *responsibility reference*. Similarly, the details of other *reused* actions are shown in Appendix D.

Since the creation of a model element, e.g., creating a *message* in the environment model, can have secondary effects, the designer provides the details of such effects in lines 26 - 31. In our example, the secondary effects of creating a *message* includes creating a *message type* in the environment model. To this end, similar to the *reused actions*, the designer specifies the language element of the secondary element, i.e., *MessageType* at line 28. To effectively propagate the effects of creating the secondary element, the recursive method (lines 29 - 30) *createOtherElementsForInputMessageType* is called to propagate the changes accordingly (see *Create Other Elements* in Figure 6.1).

To *re-expose* a language action, the structure of the *redefined* action is followed, but excluding the *facadeAction* in lines 16 - 25 and with the *reexpose* keyword instead of the *redefined* keyword (see line 10). If a language action is not re-exposed or redefined it is hidden from the user. This is the case, for example, for the *createOperation* action of the class diagram language, since in our example class diagrams are used for the purpose of domain modelling, which does not require operations.

To establish a *LEM* between the *Class* metaclass from the class diagram language and the *ActorType* metaclass from the environment model, the designer creates a *Compulsory Optional* mapping (lines 33 - 53). At line 33, the designer encodes the direction of the navigation between the instances of the referenced language elements across language boundaries, which can be either *bi-directional* or *uni-directional* navigation. Note that the designer can ignore the requirement to specify the navigation, which means that a user will not be provided with a navigation facility between the concerned model elements. Each *LEM* comprises the *from* mapping-end (lines 34 - 40), *to* mapping-end (lines 41 - 47), and optional nested mapping (lines 48 - 53).

For each mapping-end, the designer specifies the model package *CdmPackage* (line 35), which is required to retrieve the object representation of the language element `Class` (line 39). Furthermore, the designer specifies whether the language element is a root element (line 36), since a root element is handled differently during the implementation of the perspective actions. The multiplicity of the *from* mapping-end is 1 (i.e., Compulsory) for the *Class* metaclass, while it is 0..1 (i.e., Optional) for the *to* mapping-end of the *ActorType* metaclass.

Further, the designer creates a nested mapping (lines 48 - 53) between the *Class* name and the *ActorType* name which is contained in the *LEM* between the *Class* and the *ActorType* metaclasses. This nested mapping ensures that the names of a mapped class and actor type (i.e., a *MEM* exists) are always the same during run-time. Also, by setting the mapping as *matchMaker* in line 50, the name of a given class can be used to find the corresponding actor type with the same name and vice versa.

This language element mapping implies that the *create* type of the applied template is *C1* (*Compulsory Optional*) for the action to create a class, while it is *C10* (*Optional Compulsory*) for the action to create an actor type, see Table 6.1. On the other hand, the *delete* type is *DELETE_OTHERS* for the action to delete a class, while it is *JUST_DELETE* for the action to delete an actor type.

The complete *LEMs* that satisfies all the consistency conditions outlined in Section 9.1.1 are shown in Figure 9.3. Each *LEM* is represented as a red link between the concerned language elements, while the *multiplicity* of each mapping-end is shown at the corresponding end of each *LEM* link. We include *model* multiplicity for models that play a *set role* in the perspective, i.e., operation model with 0..* instance cardinalities (text on blue background). As shown, each consistency condition (e.g., *R1*) corresponds to the link with the label (e.g., *R1*). Each *LEM* establishes a nested mapping between name attributes. The name attribute can be directly contained in the language element in question, which is not shown for simplicity. Alternatively, the name attribute can be contained in another language element, which is shown for clarity. For example, the name attribute for the `Message` (R7) refers to the name attribute of the `MessageType`, as shown in Figure 9.3.

Finally, based on the perspective specified with our DSL, the perspective actions are fully generated with our code generator (see Section 8.3.4), which is based on the generic templates (see Section 6.2) and the language specification (see Figure 9.1).

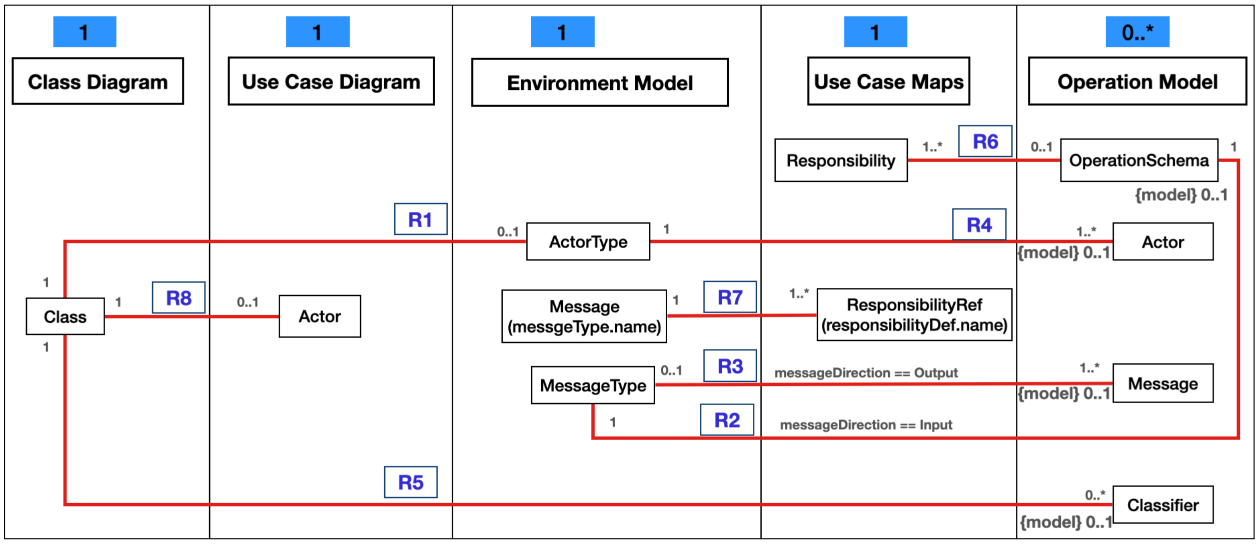


Figure 9.3: LEMs in a Fondue Requirement Perspective

9.1.4 Instantiation of the Perspective

The final step is to instantiate the *PML* framework from an instance of the DSL (Figures 9.1 and 9.2) by applying a transformation based on the generic templates, and then integrating the perspective including its perspective actions with a software tool. In this example, we integrate the perspective actions with the TouchCORE software tool [16] by intercepting the language action calls, e.g., create new class, with the corresponding *redefined* perspective action.

Once the perspective actions are in place, when a user requests to create a domain model class element (e.g., a customer in a bank application), the *PML* framework first calls the class diagram language action, *createClass*) to create the *Customer* class and then asks the user whether to create the optional corresponding element in the environment model (*R1*, i.e., a customer actor type in the environment model). Hence, if the user selects *yes*, the *PML* calls the respective language action to create the corresponding *Customer* actor type with the same name, and then establishes a *MEM* between the two instances. Since the *ActorType* in the environment model is mapped to *Actor* in the operation model (*R4*) and the operation model plays a *set role*, *PML* applies the *set role* constraints, i.e., ignores the *LEM* (R4) if there is no operation model or displays a missing *MEM* warning to the user, when there is at least one operation model exists. With the warning approach, the user can request to create a *Customer* actor in the operation model, which automatically links to the *Customer* actor type in the environment model.

Since the *Class* metaclass is mapped (i.e., *LEM*) with two other language elements - *Actor* in the use case diagram (*R8*) and *Classifier* in the operation models (*R5*), *PML* also applies the corresponding *create* types (i.e., *C1* (Compulsory Optional) for *R8* and *C3* (Compulsory Optional-Multiple) for *R5*).

As a second example, consider a user creating an actor (e.g., a *Manager* actor) in the use case diagram. Then, the *PML* framework, first, searches for a corresponding domain model class in the class diagram model. If the search returns a corresponding class which has the same name as the name of the new actor, *PML* creates a *MEM* between the new actor and the corresponding domain model class. Otherwise, *PML* proactively creates a corresponding domain model class with the same name and a *MEM* (which will then trigger the same change propagation as explained earlier).

As a final example, consider that a user creates a message in the environment model. In that case, *PML* first creates the message and then some responsibility references in the use case maps (based on the number specified by the user), and then successively establishes *MEMs* between the new message and the corresponding new responsibility references. To create these responsibility references, the language action (*create responsibility reference*) may create a new responsibility as a secondary effect. The *Responsibility* metaclass has an

Optional relationship with the *OperationSchema* in the operation models. If we assume that the user decides not to create a corresponding operation schema, then the change propagation stops here.

However, creating a message in an environment model may also require to create the *type* of the message as a secondary effect. Hence, if an output message type is created, due to the secondary effects, the corresponding messages are created in the operation model. On the other hand, if the input message type is created, then, exactly one operation schema is created in the operation model. Consequently, creating an operation schema requires to create or use existing responsibilities in the use case maps model, depending on the number specified by the user. The new and the retrieved existing responsibilities are then successively mapped with the new operation schema. In the case of the existing responsibilities, each responsibility name must be the same as the name of the operation schema which must be the same as the name of the input message type, i.e., the responsibility that was created while creating the message in the environment model as the secondary effect of creating the corresponding responsibility reference has the same name as the operation schema and hence is used to establish a *MEM* with the new operation schema.

In all of these examples, the perspective ensures that the equivalency, equality, and multiplicity constraints between mapped model elements are maintained as they evolve.

9.1.5 Discussion of the Perspective Actions

To demonstrate the relevance of perspective actions in the multi-language perspective, we analyse the language actions for the five languages of the perspective based on the *LEMs* shown in Figure 9.3. This analysis aims to ascertain the impacts of perspective actions on those language actions aiming to maintain consistency conditions in the *requirement*

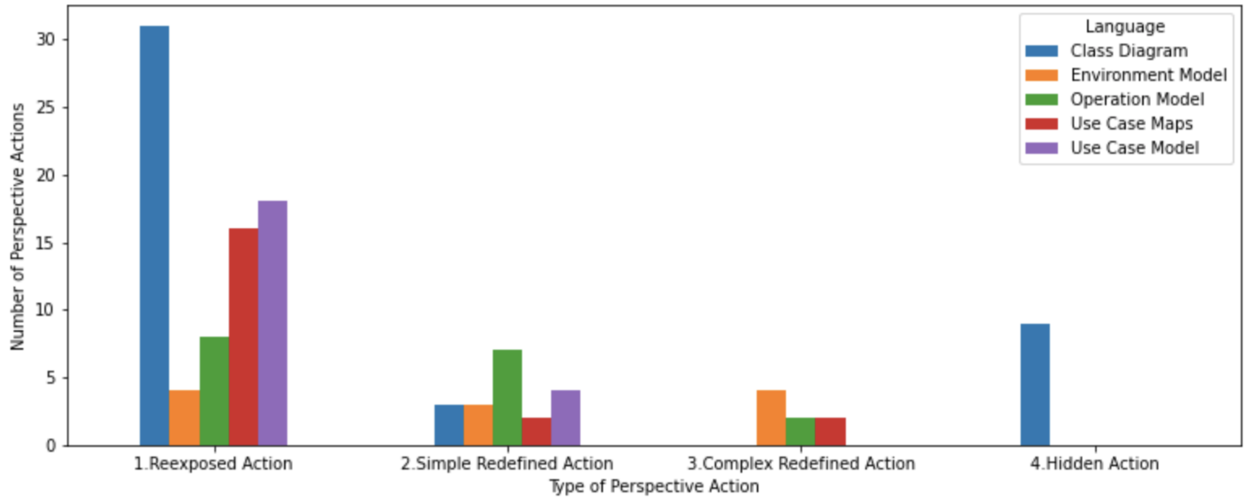


Figure 9.4: Distribution of Perspective Action in the Languages

multi-language modelling environment. In this multi-language perspective, there are 11, 22, 43, 20, and 17 language actions for the environment model, use case diagrams, class diagrams, use case maps, and operation model, respectively. The distribution of the types of perspective action (*re-exposed* action, *simple redefined* action, *complex redefined* action, and *hidden* action) across the five languages is shown in Figure 9.4. In this evaluation, we further categorise *redefined* perspective actions into *complex redefined* and *simple redefined* actions. In a complex redefined action, both the *primary* and *secondary* effects of the redefined language action affect the consistency conditions of at least one *LEM* in Figure 9.3. In the case of a simple redefined action, only the *primary* effect of the language action affects the consistency conditions of at least one *LEM* of the perspective.

As shown in the figure, the class diagram language has the highest number of both *re-exposed* and *hidden* actions, while the operation model has the highest number of *simple redefined* actions and the environment model has the highest number of *complex redefined* actions. Only the class diagram has hidden actions since the perspective reuses the class diagram for the purpose of domain modelling, which forbids the use of some language actions, especially language actions related to the `Operation` language element. This provides anecdotal evidence that hidden actions are particularly useful for languages with a broad application domain that may address several purposes. For example, a class diagram may be used for the purpose of metamodelling, domain modelling, and design modelling, each of which requires a different set of language elements.

For three of the five languages, the number of re-exposed language actions is higher than the number of all other language actions combined. For all five languages together, 68% of the perspective actions are re-exposed actions. This is followed by simple redefined actions with 17%, hidden actions with 8%, and complex redefined actions with 7%. This gives an indication that all types of perspective actions are needed and that the vast majority of them are actions that are straightforward to handle during the specification of a perspective, i.e., re-exposed and hidden actions. Simple and complex redefined actions require the perspective designer to identify language actions that affect *LEMs*. Simple redefined actions are also identified without much difficulty, because it is often clear from the name and parameters of the language action which language element is affected. This leaves only 7% of the actions that require a more detailed analysis to determine which *LEMs* are affected because of secondary effects. In the requirement perspective, only the environment model, operation model, and use case maps have complex redefined actions. In general, the number of language elements with *LEMs* influences the number of redefined actions (e.g., the number of redefined actions is higher for the environment model and operation model, which also have a higher number of language elements with *LEMs* (see Figure 9.3). Furthermore, a type-instance relationship (e.g., `Message` and `MessageType` in the environment model or `Responsibility` and `ResponsibilityRef` in use case maps) or a container-part relationship (e.g., `OperationSchema` and `Message` in the operation model) tends to lead to complex redefine actions, if both language elements are involved in a *LEM*.

We also investigate which generic templates are needed for this perspective. For each redefined action, the perspective action applies some generic templates to ensure that the consistency conditions of the concerned *LEMs* are maintained. In general, the update template is used in 23% of the cases, while a delete template is used in 27% of the cases, and a create template is used in 50% of the cases. The distribution of the types of applied generic templates is shown in Figure 9.5. While all update and delete templates are required by the perspective, only eight of the twelve create templates are needed. The *C7* (*Optional-Multiple Optional/Compulsory-Multiple*), *C8* (*Compulsory-Multiple Optional/Compulsory-Multiple*), *C9* (*Optional Optional*), and the *C11* (*Optional Optional-Multiple*) create types are not needed for this perspective. However, other examples exist for these four create types. *C7* may be needed for a perspective with sequence diagrams and operation models, as the `LifeLineTypes` from several sequence diagrams describe the same entity as the `Classifiers` from several operation model. *C8* may be needed for a perspective with operation models and a goal model, because the `Actors` from several operation models describe the same entity as the `Actors` in a goal model and an actor in an operation model must be related to an actor in a goal model. *C9* may be needed for a perspective with a class diagram and a goal model, as the `Class` in the class diagram may or may not be related to an `ActorType` in the goal model (not all actor types are classes, because some actor types model stakeholders that are not interacting directly with the system; not all classes are actor types, because some classes are related to domain concepts other than stakeholders). Finally, *C11* may be needed for the same perspective with a class diagram and a goal model, as the `Class` in the class diagram may

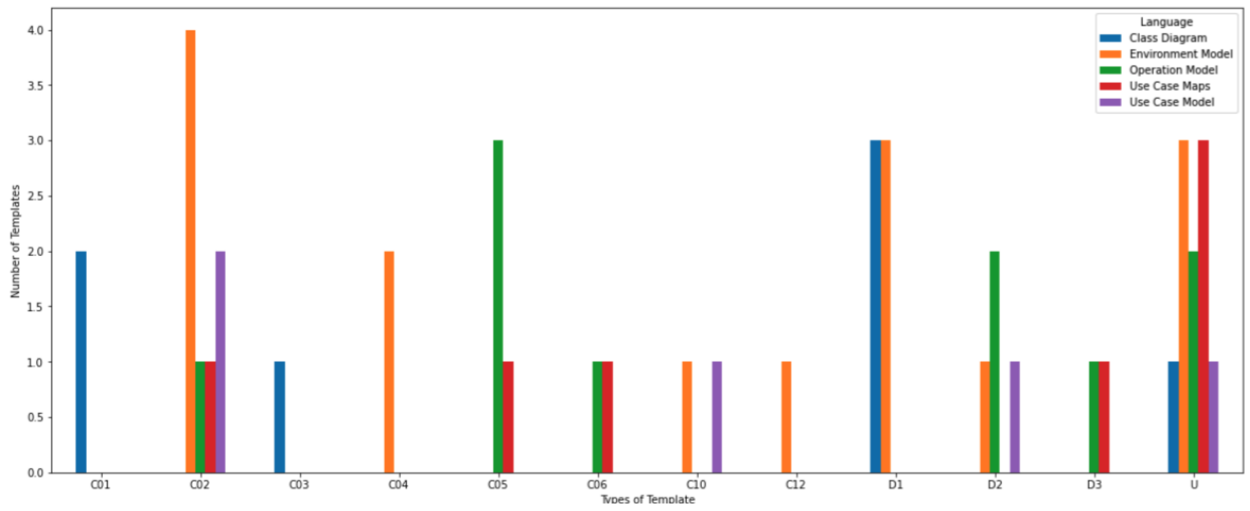


Figure 9.5: Distribution of Generic Templates Based on the Reused Language

or may not be related to one or several `Actor` (instances of `ActorType`) in the goal model.

Furthermore, the case study helped discover a special case. The language action to create an actor in the environment model does not have a *LEM* for its primary effect, but only for its secondary effects (i.e., creating an actor type). This special case is still covered by the proposed templates, because the redefined language action to create an actor still calls the recursive perspective action *createOtherElements* and then *handleSecondaryEffects*. However, since no *LEM* exists for the primary effect, nothing happens in *createOtherElements* and the template moves immediately on to the secondary effects.

This case study shows that the proposed perspective actions and their templates work to augment language actions in a multi-language modelling environment to enforce consistency conditions. To assist the perspective designer, the case study demonstrates how we apply the templates to fully generate the perspective actions which are then used to maintain the consistency conditions in the modelling environment. This generative approach frees the designer from a rigorous and error prone manual implementation of the perspective actions. Hence, the designer can focus on the specification of the *LEMs* (see Figures 9.1-9.3).

9.2 Palladio Component Model Perspective

In this section, we provide an overview of the Palladio Component Model (PCM) [56, 57], a domain-specific language that predicts the fufillment of extra functional properties of a software system, which include performance, maintainability, security, and reliability. PCM targets component-based software architectures, where different developer roles can be integrated to analyze and then predict the performance of an entire software system, especially during the design stage. Software architects can leverage the results of PCM analyses to revise their architecture as well as the design of the system. This early stage detection of the extra functional requirements bottlenecks helps to prevent the redesign of the whole system after production, which often involves a huge cost.

9.2.1 PCM Metamodel

The PCM metamodel comprises several independent domain-specific modelling languages, which are aligned with different developer roles in Component-Based Software Engineering (CBSE) [58]. CBSE fosters the development of complex systems by assembling basic software components to reduce the complexity as well as promote software reuse. In this section, we briefly explain some of the developer roles, i.e., component developer, software architect, system deployer, and domain experts roles, and the corresponding metamodel.

Component developers oversee the specification and the implementation of software components. In addition, they provide the description of each component extra-functional properties, which can be used by software architects to predict the performance of assembled components before deployment. Furthermore, system deployers provide more details about the performance of the architecture especially infrastructure/software of the system, e.g., speed of a CPU. And finally, domain experts assist software architects with the information to specify a usage model of the architecture.

The PCM metamodel provides a domain specific language for each developer role as shown in Figures 9.6 - 9.9. Developers specify and then implement components that are deposited in a repository with a DSL metamodel shown in Figure 9.6. Each `BasicComponent` encapsulates its implementation and indicates the required and provided interfaces. The software architects then use these components to build the architecture of the system with a DSL metamodel shown in Figure 9.7. Furthermore, the system deployer uses a DSL (see Figure 9.8) to creates the resource environment and then allocates individual components to the resources. And finally, the business domain experts construct the usage scenario and then connect it to the roles provided by the system with a DSL metamodel shown in Figure 9.9.

Although these four metamodels are independent, the role at each level of an abstraction depends on or influences other developer roles. Hence, it is essential that the DSLs coherently describe the system under development. In the next section, we present some consistency rules that govern the coherent development of a component-based system with PCM and then show how *PML* can be leveraged to assist different developers at different levels of abstraction to produce more consistent systems, while promoting the individual evolution of the DSLs.

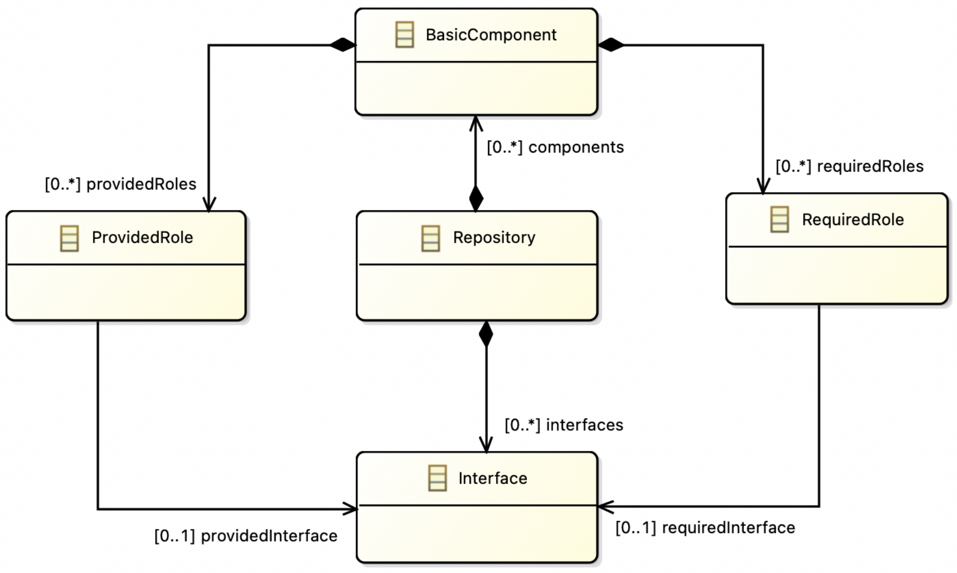


Figure 9.6: PCM Component Repository Metamodel (excerpt)

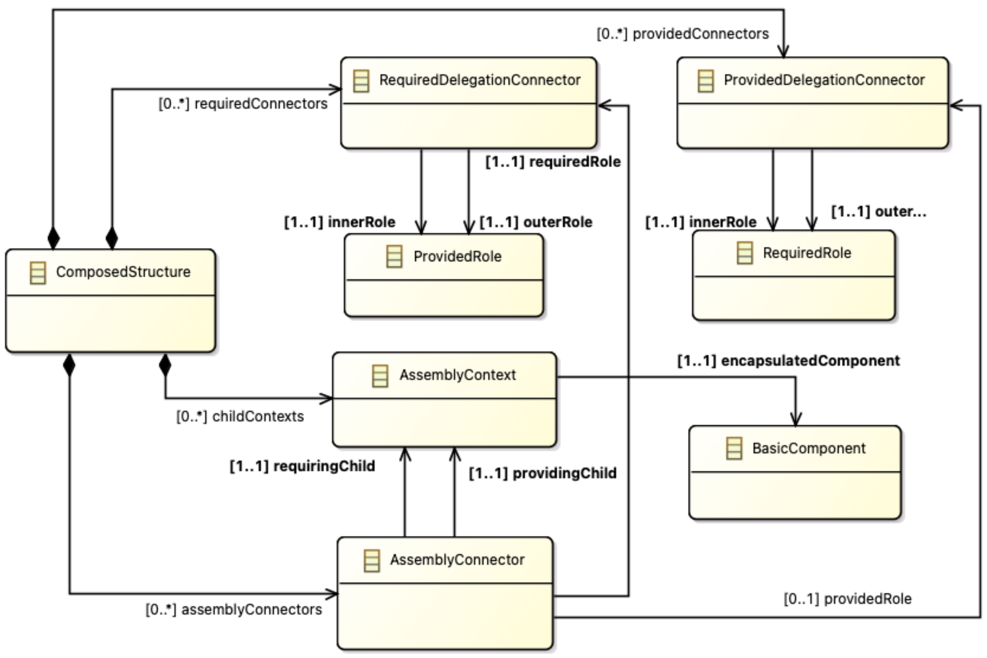


Figure 9.7: PCM Architecture Metamodel (excerpt)

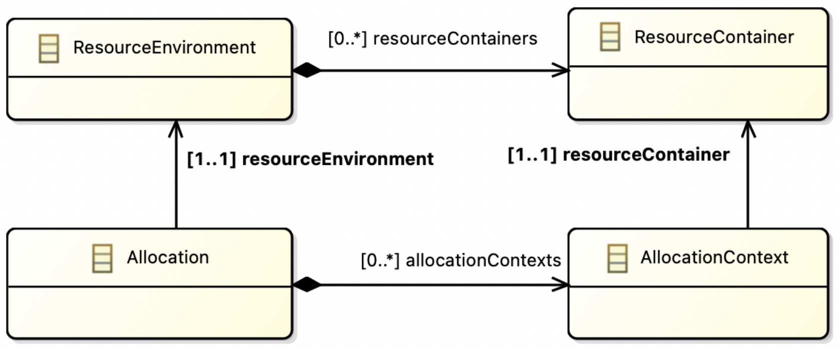


Figure 9.8: PCM Resource Metamodel (excerpt)

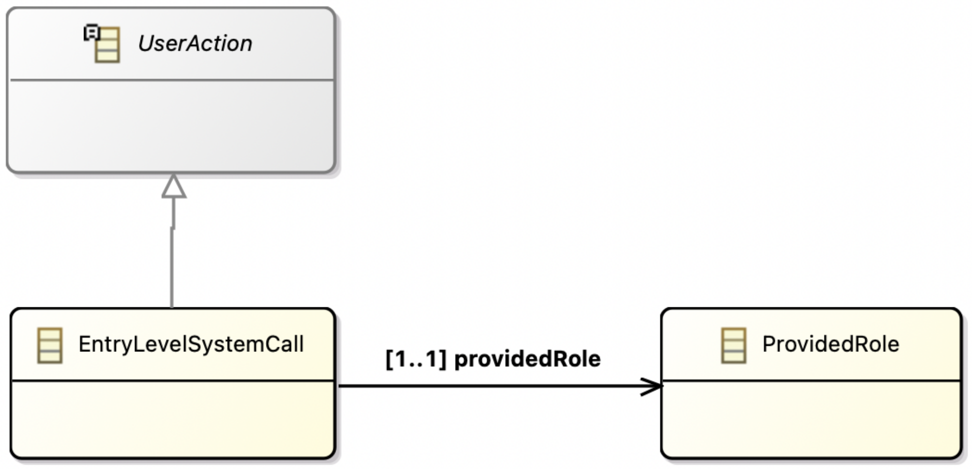


Figure 9.9: PCM Usage Metamodel (excerpt)

9.2.2 Purpose of the PCM Perspective

A PCM model comprises different individual models from different DSLs as presented above. Since these models describe a system, a perspective designer is required to specify the consistency conditions, i.e., purpose of the *perspective*, which need to be maintained in the modelling environment. In this example, a PCM perspective designer aims to maintain the following equivalency and multiplicity constraints in the multi-language modelling environment. Furthermore, for each of the equivalency constraints, an equality constraint ensures that the names of the elements are synchronized (if a name attribute exists).

- **R1**: Each *basic component* from the architecture model must have a corresponding *component* in the component specification model.
- **R2**: Each *provided role* from the architecture model must have a corresponding *provided role* in the component specification model.
- **R3**: Each *required role* from the architecture model must have a corresponding *required role* in the component specification model.
- **R4**: Each *allocation context* from the resource model must have a corresponding *assembly context* in the architecture model.
- **R5**: Each *provided role* from the usage model must have a corresponding *provided role* in the architecture model. On the other hand, each *provided role* from the architecture model must have at least one corresponding *provided role* in the usage model.

These rules are encoded in the *LEMs* for the PCM perspective as detailed in Figure 9.10. Similar to the Fondue Requirement perspective, a perspective designer can use our DSLs to register the languages and then specify the perspective. Furthermore, the designer can

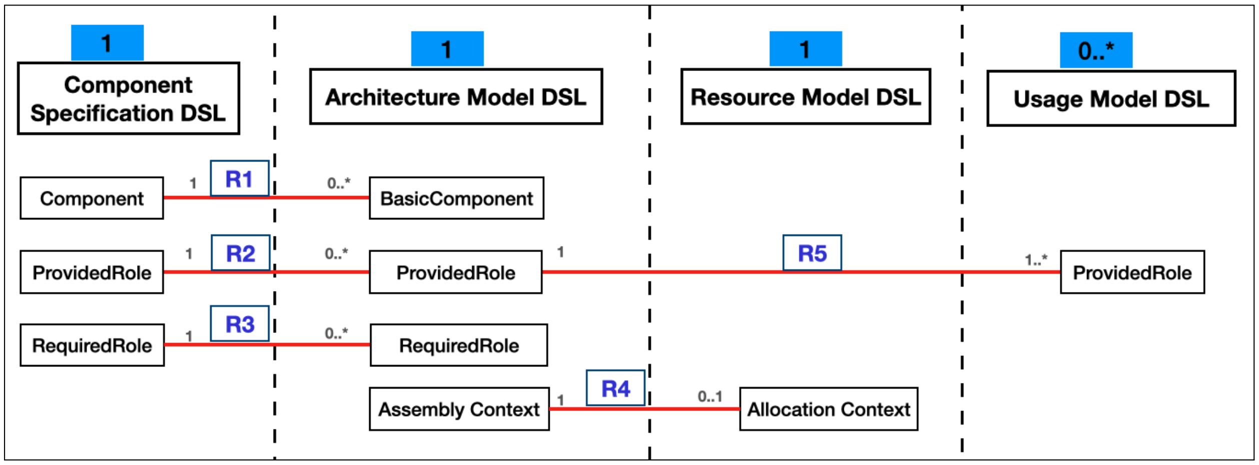


Figure 9.10: LEMs in a PCM Perspective

generate the perspective implementations and then integrate the perspective with a software tool. This example demonstrates another important use case, where the *PML* framework can foster a coherent development of a component-based software system. In addition, our framework supports the individual evolution of each language, which promotes the maintainability of the each system under development as well as the modelling environment.

9.3 User Requirements Notation Perspective

The User Requirements Notation (URN) [56] is a popular multi-language paradigm aimed at elicitation, specification, analysis, and validation of software system requirements. It combines two modelling languages: the goal-oriented requirement (GRL) and the use case maps (UCM). Here, we discuss URN extended with feature model (FM) language, because a feature model does have several relationships with both GRL and UCM when used together to describe the requirement of a system. For the rest of the thesis, we refer to URN as a multi-language modelling technology that combines these three languages. Therefore, the URN perspective comprises the above three modelling languages, as well.

9.3.1 Overview of the URN Perspective Languages

This section provides a brief overview of the three reused languages in the URN perspective. First, a GRL is a modelling language used for the specification of the intentions and business goals of different stakeholders (see Figure 9.11, GRL metamodel). A modeller uses the GRL language to capture and then analyze different intentions of stakeholders to ascertain how they impact the goal of a system.

Second, as described in *Fondue Requirement* (see Section 9.1 on page 156), use case maps model defines the allowable sequences of interactions that the system may have with its environment over its lifetime (see Figure 4.5, UCM metamodel).

Third and finally, the feature model describes the relationships among features, i.e., the set of feature configurations that produce valid products; see Section 5.3 (page 88) and Figure 5.5 (FM metamodel) for more details about the feature model.

These three languages are collectively used to elicit, specify, analyze, and validate system requirements. Hence, it is essential that the three different types of language models coherently describe a system under development. Similar to the *Fondue Requirement* and PCM perspectives, each language exists independently; which fosters modularity and maintainability of the system.

9.3.2 Purpose of URN Perspective

Similar to the *Fondue Requirement* and PCM perspectives, the purpose of the *URN* perspective is encoded with the *LEMs* shown in Figure 9.12. Each *LEM* captures the

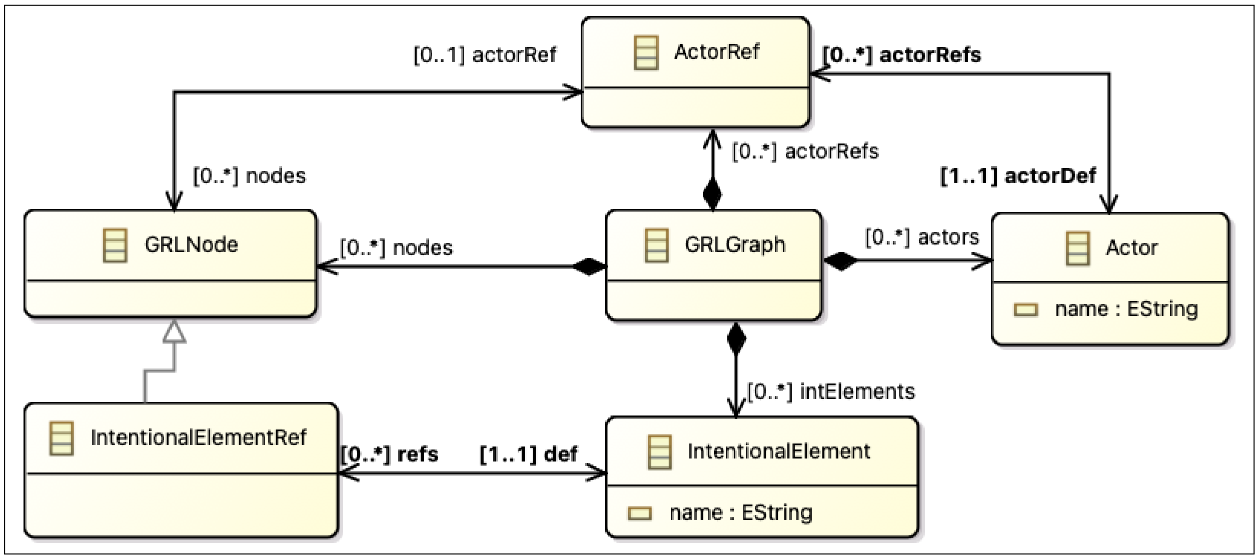


Figure 9.11: GRL Metamodel (excerpt)

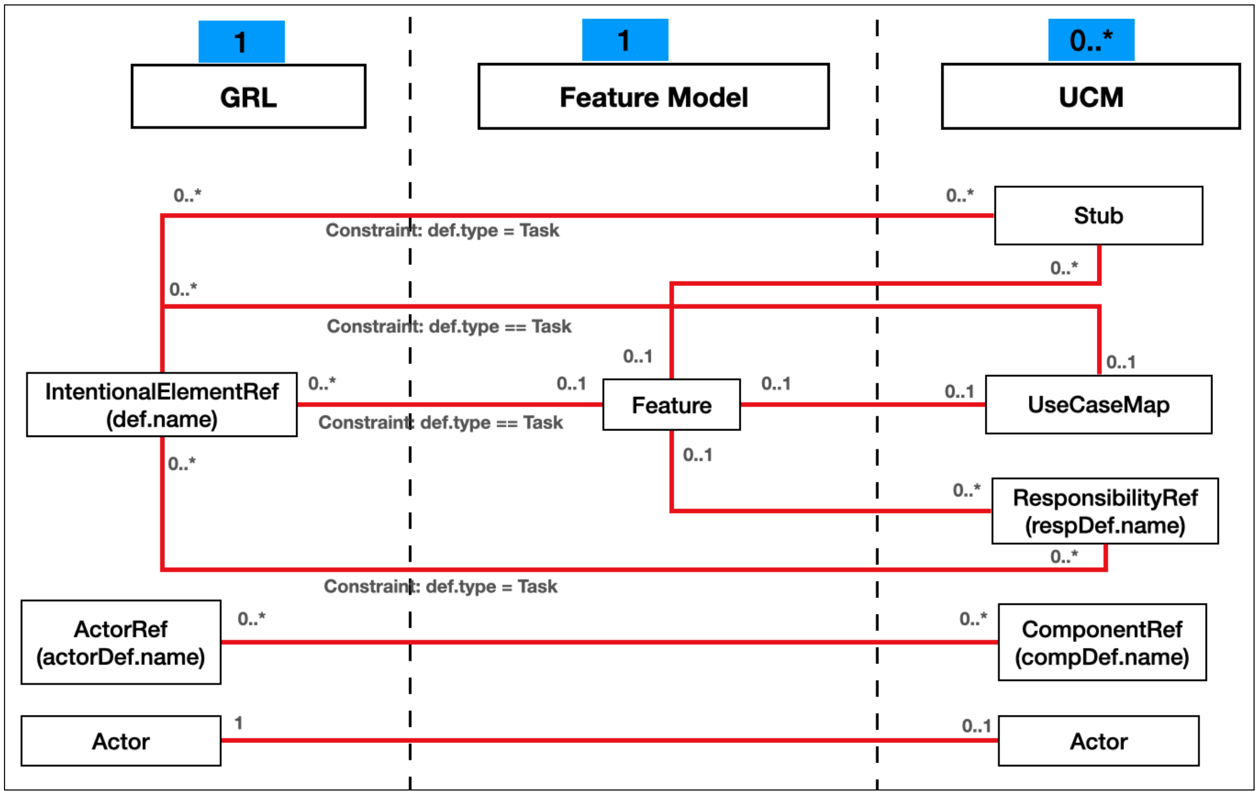

1
GRL
1
Feature Model
0..*
UCM
Stub
Constraint: def.type = Task
Constraint: def.type == Task
IntentionalElementRef (def.name)
Feature
Constraint: def.type == Task
UseCaseMap
ResponsibilityRef (respDef.name)
Constraint: def.type = Task
ActorRef (actorDef.name)
ComponentRef (compDef.name)
Actor
Actor
0..*
0..1
1

Figure 9.12: LEMs in a User Requirements Notation Perspective

equivalency constraints (including the equality constraints and the optional conditional constraints) to ensure that the three different languages coherently define a system, even as the software system evolves.

For example, the *LEM* between the `IntentionalElementRef` from the GRL language and the `Stub` from the UCM language dictates that each *intentional element reference*, of a type *Task* (i.e., conditional *LEM*), from the GRL model can be mapped with many *stubs* from the UCM model. Conversely, each stub can be mapped with many *intentional element references* from a GRL model, provided that each intentional element references is a *Task* type. The *equality* constraint of this *LEM* requires that a *definition* (`IntentionalElement`) of an *intentional element reference* must have the same name as the name of the stub. The encoded *def.name* in the *IntentionalElementRef* (see Figure 9.12) shows the navigable link to the name of the referenced *intentional element* (i.e., *defitnition*), where *def* is the role name of association from the `IntentionalElementRef` to the `IntentionalElement` and *name* is the name attribute in the `IntentionalElement`.

Hence, the *equality* constraint can be captured with attributes within the language element of interest or any other attribute contained in the language model, provided that the attribute is navigable from the language element in question. Note that we do not show the attribute used for the *equality* constraint if the attribute (e.g., *named* attribute) is contained in the language element of concern. Hence, the *Stub* has no encoded *name* attribute, although each stub is required to have the same as the corresponding intentional element references. Similarly, a Stub and a Feature are mapped using the same *name* attribute *equality* constraint. The same *name* attribute *equality* constraint also applies to a Feature and a UseCaseMap, as well as an Actor (GRL) and an Actor (UCM).

With these *LEMs*, a perspective designer can then register the languages, if not registered in the tool, and then specify the perspective with our DSLs. The process of implementing the perspective is the same as that of the *Fondue Requirement* perspective. Our aim here is to show that the *PML* framework can be used to implement the *URN* perspective.

9.4 PML Navigation Evaluation

In this section, we analyse the navigation facilities of several popular modelling tools and evaluate whether our navigation metamodel can support the discovered navigation facilities. We performed a Google search for "most popular UML tools". From the obtained list, based on the google search, we investigated the top 4, namely: StarUML (free), ArgoUML (free), Visual Paradigm Enterprise (commercial), and MagicDraw (commercial). We also selected Papyrus, as a representation of a popular modelling tool based on EMF, and finally TouchCORE [16], as a representative of a UML modelling tool that explicitly supports software product line modelling and model reuse. In each tool, we specified a class diagram, and defined the behavior of some operations using sequence diagrams or state machines. We then explored how the tools support navigation. We organized our findings under the headings intra-language navigation, inter-language navigation, filtering, element highlighting, navigation of inheritance hierarchy, feature-based navigation, and navigation across reuse boundaries.

9.4.1 ArgoUML

ArgoUML is an open source tool which supports all UML 1.4 diagrams [51], including Class Diagrams, Use Case Diagrams, and Sequence Diagrams. The basic navigation facilities of the tool include a *model explorer*, which hierarchically lists models and its diagrams, and an *edit* *toolbar*, which allows software developers to use or define their own *perspectives*. Perspectives in ArgoUML primarily define filtering conditions for model elements in the explorer.

- **Intra-Language Navigation:** ArgoUML supports navigation between model elements within a diagram. For example, in an ArgoUML project, the model explorer shows the list of diagrams and all the hierarchical elements. In a class diagram, a click on a tab shows the related model elements such as associations, inheritance, subclasses, and operations. This allows modelers to navigate between elements within a model.

- **Inter-Language Navigation:** In the *Diagram-Centric* perspective, the model explorer visualizes model elements under the diagrams in which they are used. Clicking on them opens the corresponding diagram and highlights the selected model element.

- **Filtering:** ArgoUML supports different kinds of filtering using their own notion of *perspective*. Each perspective specifies the kind of model elements to be shown in the explorer, e.g., *Class-Centric*, which configures the explorer to only display diagrams and classes, *Package-Centric*, which displays only the contents of a package including diagrams, classes, and associations. The tool allows modelers to define their own perspectives using existing rules by combining existing filtering conditions in the library.

- **Element Highlighting:** When a model element is selected in the model explorer, the element is highlighted in blue in the editor. However, the highlighting is only visible if the element is currently shown in the editor.

- **Inheritance Hierarchy:** ArgoUML defines an Inheritance-Centric *perspective*, which configures the model explorer to list all the model classes and their subclasses, hence allowing the modeller to easily explore the inheritance hierarchy.

Our proposed generic navigation approach can support all the navigation facilities that ArgoUML offers. The perspectives of ArgoUML can be represented as a filtering condition in our generic mechanism. For example, the Class-Centric perspective lists only diagrams and classes in the model explorer. With our approach, this can be done by setting the `active` flag of `Mapping` for all instances of `Class` (see Figure 5.2 and 5.13), and deactivating all other mappings. Our proposed navigation metamodel supports even finer-grained filtering based on attribute values. This is currently not possible in ArgoUML.

9.4.2 StarUML

StarUML is a modelling tool which is compatible with the UML 2.x standard. The tool supports 11 types of UML diagrams including Class Diagrams, Use Case Diagrams, Sequence Diagrams, and State Diagrams [50]. Notable navigation features in StarUML include the *model explorer*, the *diagram list*, and different categories of pallets, e.g., *Classes (Basics)*, which shows unique components for creating class diagrams, *Instances*, and *Annotations*.

- **Intra-Language Navigation:** The tool partially supports intra-language navigation. The model explorer shows the definitions of the model elements as well as their contents. For example, in a class diagram, a click on an arrow before a class reveals its operations, attributes, and associations. To visualize an element in a diagram, a modeller needs to right-click on the element and choose *Select In Diagram*. In the list of diagrams, however, it is not possible to navigate to the model elements contained in the diagrams.

- **Inter-Language Navigation:** The tool supports inter-language navigation using the model explorer. Clicking on a class shows the contained operations. Clicking on the operation displays the list of associated sequence diagrams, if any. To actually switch to the sequence diagram view, the modeler has to choose *Select In Diagram* by right-clicking the sequence diagram in the model explorer, or alternatively manually find and then select the desired sequence diagram in the diagram list.

- **Filtering:** StarUML has no support for filtering.

- **Element Highlighting:** Each model element in the currently displayed diagram can be highlighted in blue by selecting it in the model explorer. If the element is currently not visible in the editor, the modeler needs to right-click the element and choose *Select In Diagram*, which then switches the current view to the diagram containing the selected element and highlights it.

- **Inheritance Hierarchy:** StarUML partially supports navigation of the inheritance hierarchy in class diagrams. A modeler can navigate from a subclass to its parent class. However, it is not possible in StarUML to visualize the complete inheritance hierarchy of a given class.

Our proposed generic navigation approach can express all the navigation facilities that StarUML provides. Additionally, our approach supports filtering and displaying of the entire inheritance hierarchy.

9.4.3 MagicDraw

MagicDraw [59] supports all UML diagrams. The most significant navigation facilities of the tool include the *containment tree* (a window with a hierarchical list of model elements) and the *diagram tool window*, which groups existing diagrams based on their language.

- **Intra-Language Navigation:** MagicDraw provides a structured containment tree which facilitates navigation from a model element to its related elements. In the *containment tree*, when a plus (+) tab before a class is clicked, its contained elements, such as operation and attributes, are displayed. Clicking on a model element displays it in the diagram editor, switching diagrams if necessary. However, just like in StarUML, the containment tree in MagicDraw displays the model element definitions separately from the diagrams in which they are used in.

- **Inter-Language Navigation:** MagicDraw provides full support for inter-language navigation. A sequence diagram or activity diagram that is linked to an operation in a class diagram can be navigated to, either directly from the model element in the model editor or by choosing it in the model explorer containment tree. Such navigation opens the target diagram in the diagram editor.

- **Filtering:** The tool has several filtering conditions under three different categories, namely: *List*, *Inheritance*, and *Structural*. Each category has multiple options that can be turned on or off, e.g., Class, Actor, or Association. When a filtering condition is enabled, the corresponding model elements are hidden in the model explorer.

- **Element Highlighting:** MagicDraw highlights model elements selected in the model explorer in bold. A modeler can right-click an element in the explorer, select *Go To* and *Usage In Diagram* to switch the current view.

- **Inheritance:** Magic Draw supports immediate superclass navigation. In the containment tree of the model explorer, a superclass can be navigated to by clicking a plus (+) tab before its subclass. However, such navigation is only visible within the containment tree.

Our generic approach supports the navigation facilities of MagicDraw. The filtering in MagicDraw is at the granularity of model element types, i.e., every model element of a given type is either shown or not shown. Our generic mechanism supports this using the `active` flag in `Mapping` (see Figure 5.13). Unlike our approach, MagicDraw does not support filtering based on attribute values, e.g., to define a filter that displays only abstract classes.

9.4.4 Visual Paradigm Enterprise

Visual Paradigm supports UML 2 and SysML modelling [60]. The most notable navigation facilities of the tool include the *diagram backlog* (a tab which opens a list of model diagrams), *property* (a tab which opens editable properties of model elements), *model explorer* (a tab which opens the containment tree of model elements within models with their diagrams), and the *diagram navigator* (a tab which opens the list of supported diagrams).

- **Intra-Language Navigation:** Visual Paradigm fully supports intra-language navigation. A click on the tab *diagram navigator* reveals all the supported diagrams in a *containment tree*. In the containment tree, when a plus (+) tab before a diagram icon is clicked, its contained diagrams are shown with their model elements. E.g., a click on plus (+) before *class diagram* shows the class diagrams with their contained model elements such as operations, associations, and attributes. However, the navigation within the diagram navigator or *model explorer* does not automatically visualize an element in the editor, unless, a modeler selects to open the diagram.

- **Inter-Language Navigation:** Visual Paradigm Enterprise provides excellent support for inter-language navigation. A sequence diagram or a state diagram that is linked to an operation in a class diagram can be navigated to, either directly from the model element in the model editor or by choosing it in the diagram navigator or model explorer containment tree. The tool attaches an icon (which refers to the linked diagram) to a class containing the operation in the editor and adds a tab (which also refers to the linked diagram) under the class in the diagram navigator containment tree. Hence, a modeler can directly navigate to the linked diagram by using either link.

- **Filtering:** Visual Paradigm does not support filtering.
- **Element Highlighting:** A modeler can right-click an element in the explorer or diagram navigator and choose *Select In Diagram*. This takes the modeler to the diagram containing the element with the element being highlighted in bold, switching the current view if necessary.
- **Inheritance:** Visual Paradigm partially supports inheritance navigation. A modeler can navigate from a class in the model explorer to any of its direct superclasses, similar to navigating over association relationships. However, Visual Paradigm does not support displaying and navigating the entire inheritance hierarchy.

Our proposed generic navigation approach covers all navigation features of Visual Paradigm.

9.4.5 Papyrus:

Papyrus is a UML modelling tool based on EMF that supports many UML diagrams including Activity Diagrams, Class Diagrams, and Communication Diagrams. Similar to the above modeling tools, it supports navigation with a *model explorer* and from within the diagram editor.

- **Intra-Language Navigation:** The tool supports navigation of elements within a model, including traversing from a diagram to its elements. In the explorer, a modeler can navigate from a model element to its contained elements, e.g., from a class to its attributes and operations. A modeler can also navigate from elements in the editor to their definition in the model explorer, and vice versa.

- **Inter-Language Navigation:** Papyrus uses *hyperlinks* to establish relationships between two diagrams, e.g., between a class and an activity diagram or a state diagram. The tool organizes the inter-language links under the corresponding model elements in the model explorer. Hence, a modeler can use the model explorer to traverse across language boundaries.

- **Filtering:** The model explorer has two main views, models or diagrams. For each element shown in the diagram editor, its contents can be selectively hidden or shown by enabling or disabling filter options. For example in a class diagram, classes can be visualized with or without name, or with or without their attributes. Filtering based on attribute values is not supported.

- **Element Highlighting:** Selected model elements in the model explorer, are highlighted in the diagram editor. Navigating from the model explorer to an element opens up the diagram containing the element in case it was not previously shown.

- **Inheritance:** Papyrus supports navigating from subclasses to direct superclasses. In the model explorer, generalizations are also visualized under the subclass. A modeller can right-click the generalization, which shows up as a separate model element in the model explorer, and from there select *navigate*, then, *Go to target* or *Go to source* to highlight the superclass. Visalisation or navigation of the entire inheritance hierarchy is not supported.

All navigation facilities of Papyrus are supported by our proposed generic navigation approach.

9.4.6 TouchCORE

TouchCORE is a modelling tool for concern-oriented software design [16, 44], focussing specifically on feature-driven modularisation as required in SPLs. It also has explicit support for model reuse, and ships with a library of reusable models dealing with recurring design concerns, i.e., security, fault tolerance, and design patterns. The tool supports Feature Models, Goal Models, Class Diagrams, State Diagrams, and Sequence Diagrams. Its significant navigation features include a *model explorer* and traceability-based visualization capabilities.

- **Intra-Language Navigation:** When selected in the model explorer, model elements in the current diagram are highlighted in orange.
- **Inter-Language Navigation:** The TouchCORE model explorer allows the modeller to navigate between related diagrams, e.g. from an operation defined in a class diagram to an attached sequence diagram.
- **Element Highlighting:** Elements selected in the model explorer are highlighted in the current model in orange. Additionally, using traceability links, TouchCORE uses

color highlighting to visualize feature and reuse information. When TouchCORE is used to compose bigger models by combining models related to different features or models from the reusable model library, the origin of model elements can be visualized using a multi-colored highlighting scheme.

- **Filtering:** TouchCORE does not support filtering of model elements for navigation purpose.
- **Inheritance:** TouchCORE does not support navigation of the inheritance hierarchy.
- **Feature-Based Navigation:** Since TouchCORE was designed to specifically support SPL, there is excellent support for feature-based navigation, e.g., navigating from a feature in a feature diagram to the associated realization model(s). Conversely, when visualizing a model in the model editor, the associated features are displayed and can be navigated to easily.
- **Navigation Across Reuse Boundaries:** The tool keeps track of reuse dependencies of models. A modeler can navigate from a current model to the reused models via the model explorer. A click on the reused tab of the tool shows the list of reused models in the current model. Selecting a model from the list opens the diagram in the editor.

Our proposed generic navigation approach covers all navigation features of TouchCORE.

The summary of the navigation evaluation shown in Table 9.1 shows a summary of each tool's navigation capabilities. Each of the investigated tools has a model explorer, which in our proposed generic approach corresponds to the navigation bar. Our proposed generic mechanism covers all the navigation means provided by the surveyed tools. No tool offers complete support for all navigation features provided by our proposed navigation mechanism.

Only one tool supports the navigation of closures: ArgoUML supports the navigation of the entire inheritance hierarchy in a class diagram. Attribute-based filtering is not supported in any of the surveyed tools. However, we decided to include this feature in our proposed metamodel, because many development environments for programming languages have the ability to filter, e.g., by public elements. Of course, our proposed metamodel could also easily be expanded by allowing an arbitrary query expression for a mapping. We decided to not include this in the metamodel, as there does not seem to be a need for it according to our analysis of popular UML modelling tools.

Tool	Intra-Language	Inter-Language	Attribute Filtering	Activation Filtering	Element Highlighting	Inheritance Hierarchy	SPL	Model Reuse
ArgoUML	Yes	Yes	No	Yes	Yes	Yes	No	No
StarUML	Yes	Yes	No	No	Yes	Partial	No	No
MagicDraw	Yes	Yes	No	Yes	Yes	Partial	No	No
Visual Paradigm	Yes	Yes	No	No	Yes	Partial	No	No
Papyrus	Yes	Yes	No	Yes	Yes	Partial	No	No
TouchCORE	Yes	Yes	No	No	Yes	No	Yes	Yes

Table 9.1: Navigation Support of UML Tools

9.5 Summary

This chapter presents the validation of the *PML* framework. First, we show how we fully implement a real-world *multi-language* perspective (Fondue Requirement Perspective), which combines five different modelling languages (a class diagram, a use case diagram, an environment model, a use case maps model, and several operation models) for the purpose of requirement elicitation and specification. Each language plays either a *regular role* or a *set role* in the perspective. We outline the purpose of the perspective in terms of the consistency conditions that are to be maintained in the multi-language modelling environment. Based on the participating language and the consistency conditions, we show how a perspective designer leverages our framework to register the languages (with our

DSL) and then create the perspective (with our DSL) to maintain the consistency conditions. Furthermore, we demonstrate how the perspective ensures that the consistency conditions are maintained when a modeller modifies the language models built according to the perspective.

Furthermore, we evaluate our framework with two additional multi-language modelling methodologies: Palladio Component Model (PCM) and User Requirements Notation (URN). PCM is a component-based software architecture that predicts the fulfilment of extra functional properties of a software system, which include performance, maintainability, security, and reliability. On the other hand, URN is a popular multi-language paradigm aimed at elicitation, specification, analysis, and validation of software system requirements. For each methodology, we present the purpose of the perspective and then the *LEMs*, which include the consistency conditions. Here, we show how our framework supports all the consistency conditions as contained in the *LEMs*.

On the other hand, we validate that our generic navigation approach covers the navigation facilities provided by current modelling tools by conducting a survey of six popular UML modelling tools.

Our navigation approach is not tool specific and can be applied to any language and modelling environment that uses metamodels. The main benefit is that if a modelling environment adopts our generic navigation approach, setting up navigation when adding a new language to an environment becomes greatly simplified. In that case, language designers do not have to implement intra-navigation support from scratch during language design, but can customize the navigation bar simply by creating the appropriate intra-language mappings. To link the new models with models expressed in other languages already supported by the modelling environment, the corresponding inter-language mappings must be defined. With the increased adoption of Domain-Specific Languages (DSLs), this approach gives language designers essential support to rapidly define navigation within models expressed in the DSL as well as across model boundaries.

This chapter is based on the following publications:

1. Ali, H., Mussbacher, G., and Kienzle, J. (2020) Action-Driven Consistency for Modular Multi-Language Systems with Perspectives. 12th System Analysis and Modeling Conference (SAM 2020), Montreal, Canada, October 2020. ACM, 95-104 DOI: 10.1145/3419804.3420270. (Acceptance rate: 62

2. Schiedermeier, M., Li, B., Languay, R., Freitag, G., Wu, Q., Kienzle, J., Ali, H., Gauthier, I., and Mussbacher, G. (2021) Multi-Language Support in TouchCORE. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS 2021), pp. 625-629, DOI: 10.1109/MODELS-C53483.2021.00096

3. Ali, H., Mussbacher, G., and Kienzle, J. (2021) Perspectives to Promote Modularity, Reusability, and Consistency in Multi-Language Systems. Innovations in Systems and Software Engineering, Special issue on Software and Systems Reuse, DOI: 10.1007/s11334-021-00425-3

The next chapter compares our research work with other contemporary work that targets multi-language modelling environments.

Chapter 10

Comparison With Related Work

In this chapter, we compare our approach with other contemporary works focusing on multi-language systems and maintaining of their consistencies, as well as the navigation facilities among popular modelling tools. Our main assumptions include that collaborating languages in a multi-language environment exist independently and that language actions covering construction semantics are defined for each language.

10.1 Single Underlying Model

Some studies favour a single underlying model (SUM) to describe a multi-language system (e.g., Orthographic Software Modelling (OSM) by Atkinson *et al.* [13]) and then maintain their consistencies. In SUM, a single metamodel captures the conceptual relationships spanning across different language domains. The consistencies and invariant conditions are defined within the single model. While this approach may seem straightforward, the evolution of such systems may not be. Removing or adding a new language may require an understanding of the whole system and all its languages and hence substantial manual efforts.

Meier *et al.* [61] compare OSM, MoConseMI, and Vitruvius [9]. MoConseMI is a SUM approach that comes with a set of operators to transform initially separate metamodels into a single metamodel. This integration of separate metamodels aims to establish the required consistencies between different metaclasses across the metamodels. In the end, models are co-evolved along with the integrated metamodels, i.e., SUM metamodel. Although this approach recognizes the importance of separate metamodels in a multi-language modelling environment, it combines the metamodels to enforce consistencies between different models. This integration of metamodels does not promote the benefits of the separate metamodels, including software evolution and maintainability.

On the other hand, our approach keeps the metamodels separate and then establishes consistency relationships between different language elements, i.e., *LEMs*. The perspective specification (including the *LEMs*) is then used generates perspective actions with the help of our generic templates to maintain consistency across language boundaries while the independent metamodels do not require models to be co-evolved. Also, Shah *et al.* [62] present a framework where different model views can be generated from a common system model. On the other hand, our approach keeps the metamodels separate and automatically generates code to enforce consistency across different languages.

With Melange [63], a new DSL can be built by combining existing (legacy) DSLs. Melange defines a configuration language that has operators to *extend*, *restrict*, and *merge* existing languages. When using merging, the metamodels of the involved languages are combined, resulting in one single metamodel containing all concepts. The merge functionality of Melange can easily be used to create a SUM by combining several existing languages. As such, the advantages and disadvantages of the approach are the ones discussed earlier. However, our approach customises (by using templates) existing language actions (construction semantics) and automatically generates perspective actions with the help of these template (based on mappings between language elements) to enforce consistencies across different languages.

10.2 Virtual Single Underlying Model

The virtual single underlying model (VSUM) approach tries to address the challenges of SUM. In a VSUM approach, on which Vitruvius and *PML* are also based, languages are not directly combined. Instead, metamodels of different languages are kept separate and consistency relationships between the metamodel elements are maintained externally. In our work, the *perspective* maintains the consistency relationships and thus establishes the virtual model. The main advantages of VSUM are: (i) removing a language from a system does not affect all languages in the system; instead, the corresponding mappings and consistencies need to be pruned, (ii) adding a new language basically involves establishing the relationship between the new language elements with only those languages affected by the new language, i.e., likely not all in the system. The challenges of this approach, however, are the specification of the mapping and then maintaining the consistency between different language elements, which is non-trivial.

Although we support VSUM, our work also focuses on the manipulation of each language's construction semantics, i.e., the language actions required for the construction of models conforming to different languages, with the aim to add further flexibility in maintaining the consistencies while working with models built using a perspective. Our main focus lies in generating the perspective actions and then using them to prevent inconsistencies across different languages. Furthermore, our approach addresses multiplicity constraints (including *set role* constraints) and conditional constraints based on the *LEMs*.

10.2.1 Traceability

Software traceability establishes correspondences across different levels of software abstractions, e.g., requirement, design, and implementation levels of abstractions. In addition, traceability establishes links between different artefacts which include models, design specifications, and test cases [64]. Traceability offers several benefits such as requirement validation and verification, impact analysis, and safety assurance, which are often enforced by the regulatory bodies as a part of a software certification process [65, 66].

Software traceability can be captured and then maintained manually. However, it often requires a huge effort as well as high cost, and it is error-prone. Hence, researchers favour the automated generation of traceability links [67, 68], which includes Just in Time (Information Retrieval) [68], Machine Learning [69], and Process Generated approaches. The process-generated approach enforces link creation after software artefacts are created, which aligns with the creation of *MEM*s after creating model elements. Although, *MEM*s are a kind of traceability links which we create automatically, *MEM*s also ensure that the consistencies of a system under development are maintained according to the purpose of the perspective.

10.2.2 Query-Driven Soft Links

In a multi-language modelling environment, independent language models are often persisted in different directories, and related model elements are linked via regular associations that span across model boundaries. However, these cross-references are persisted using URIs that reference model elements across different storage locations.

Hegedüs *et al.* [70] claim that connecting model fragments across different storage locations with hard links is too rigid and error-prone. The authors present query-driven soft traceability links for models, which aims to foster soft links between model elements, instead of regular references. Software links are calculated on demand, based on a specified query. Hence, soft links are derived after loading language models in a tool. Also, the work issues a warning when a soft link is broken so that the user can fix it by creating or modifying existing model element. With the derivation and incremental model query, these soft links are not persisted; hence, avoiding the hard link between different models from different resources.

This work mimics the architecture of a VSUM, where an instance of the query language acts as a virtual model that oversees the relationships between language models. This work addresses an essential problem, i.e., eliminating hard links between different model elements at different resources. On the other hand, the *PML* framework offers the equivalency consistency condition, which establishes links between model elements across language models at different resources. Although we persist the links in the form of *MEMs*, we offer additional features that are not supported by the query-driven soft links. (1) Our framework proactively maintains consistency between models by creating/deleting/updating model elements, i.e., equality constraints. (2) We support multiplicity constraints, which restrain the number of allowable instances that can be connected between language models. And finally, (3), we provide a DSL which provides a 1-to-1 mapping between language elements to capture the required links during modelling time, instead of using a query language, which may require more effort to specify the query used to maintain the soft links.

10.2.3 Vitruvius

Vitruvius [9, 71, 72] presents a framework that supports flexible views [15] that may involve multiple models conforming to different metamodels. The approach is aligned with the Orthographic Software Modelling (OSM) approach by Atkinson et al. [13]. However, Vitruvius supports the simultaneous use of models expressed using multiple metamodels. Vitruvius provides several mechanisms for enforcing consistency at the metamodel level, hence forming in some way a *virtual* single underlying model. The work further uses these mechanisms to generate a model transformation which aims to restore consistency between mapped model elements.

Vitruvius reacts to changes in models and then alters models accordingly. The reactions work at the level of CRUD operations, i.e., atomic updates to the model. On the other hand, our approach generates perspective actions which aim to prevent inconsistencies from occurring. We further capture and then maintain the multiplicity constraints (including *language role* multiplicity and *model* multiplicity) of each mapping which is not addressed in Vitruvius.

10.2.4 View-Based

Another example of a VSUM approach is a *view-based* approach [73]. A *view-based* approach promotes predefined *viewpoints*, each representing a particular context of the system aiming to improve understanding of the system under development. Each *viewpoint* can contain information from different separate models, potentially expressed with different modeling languages. A recent survey was conducted on existing *view-based* approaches to identify their differences as well as their limitations [74]. The result of the survey reveals that existing *view-based* approaches lack flexibility, especially during the evolution of the system.

Furthermore, the authors propose several research areas in this domain, which include the *view updating problem*, *incremental view maintenance*, and *concrete syntax generation*, to name a few.

Marussy *et al.* [75] present a view model synchronization between a target and source models. This work employs model transformation (based on graph query pattern) to derive a target model (view model) from the source model. The transformation engine responds to aggregated changes in the source model observed in the graph query results (i.e., reactive responds), then builds and maintains a partial model (which may be inconsistent) to keep track of the changes. The target model is updated with the partial model once the partial model represents a valid instance (i.e., a consistent model) of the target metamodel.

Furthermore, Bruneliere *et al.* [76] state that model scalability limits the full adoption of *view-based* approaches in industry [77]. Hence, the work [76] promotes the creation of a scalable model view in a multi-language modeling environment.

A *view* in *view-based* approaches and a *perspective* in *PML* are orthogonal concepts. While a *view-based* approach combines fragments of different models into a *view* that addresses a specific context, *PML* ensures consistency among models in a *perspective*. Hence, different views could be created for the models within a *perspective* to highlight various context situations, while benefiting from the consistency management of *PML*.

10.3 Other Related Works

Another popular work related to the *PML* framework is the megamodel [78]. Megamodel describes MDE concepts (including models, languages, systems, and transformations) and their relationships. These concise definitions as well as their relationships aim to assist software stakeholders in reasoning about a complex software engineering process without entering into the details of the technological space involved. Technically, a megamodel is a model of MDE concepts and their relationships. Favre *et al.* [78] believe that software evolution can be modelled as a graph using the megamodel. This work is similar to the *PML* framework because it provides the definitions as well as the relationships between languages, metamodels, and models in a multi-language environment. However, *PML* focuses on the reuse of independent languages; coordination between languages; and maintenance of consistency between languages in a complex system that comprises multiple languages. Moreover, *PML* targets relationships between languages using language elements, while the megamodel defines relationships using an entire language, a model, a metamodel, or a transformation.

Combemale *et al.* [79] claim that current workbenches used to build domain-specific modelling languages (DSMLs) do not support coordination across different languages. Hence, they proposed globalization of modelling languages, i.e., the use of multiple languages to support a coordinated development of different aspects of a system. However, the authors maintain that the main challenge is how to realize the core multi-language relationships: interoperability, collaboration, and composition. In our approach, the *redefine* perspective action fosters interoperability and collaboration between different languages while the *PML* framework, in general, supports modular combination of independently existing languages.

Deantoni [80] advocates for explicit behavioural semantic definition for each language as well as ones that cover coordination patterns that exist between different languages. This approach is complementary to perspective actions in *PML*, which address the construction semantics of the individual language as well as the combined languages instead of behavioral semantics. Furthermore, *PML* supports modular reuse of languages as well as consistencies between different language elements. Steimann *et al.* propose a role based modularisation approach [81] which covers both the abstract syntax and semantics of a language, thereby promoting self-contained language components. Similarly, PML encodes construction semantics with the abstract syntax of a language. However, PML addresses consistency conditions between different language elements by the generation of perspective actions from generic templates.

Kolovos *et al.* [82] classify different types of relationships that could potentially exist between different language elements that collaborate in a software system and establish types of inconsistency that affect a given relationship. These classifications are automatically applied to detect and fix inconsistency between different language elements. Our approach leverages some of the relationships (consistency types) presented by Kolovos *et al.* However, similar to Vitruvius, the authors address inconsistency using a reactive approach while we use perspective actions to proactively prevent inconsistencies.

Cicchetti *et al.* present a bidirectional model transformation language (JTL) [83], which aims to support non-bijective model transformations and change propagation. JTL relies on a universal metamodel which may lead to complex model management and evolution, while *PML* does not use a single underlying model and instead groups different metamodels to serve as a modular underlying model. Furthermore, we generate perspective actions with the help of templates to maintain consistency across different language elements.

To support multi-language development environments (MLDEs), Pfeiffer and Wasowski [84] present a generic framework which allows software developers to work on different related models, potentially conforming to different languages. The authors identify four main requirements of MLDEs: *Visualization*, *Navigation*, *Static Checking*, and *Refactoring*, and demonstrate them by using *TexMo*. Similar to JTL [83], *TexMo* is built by using a single underlying approach. *PML* aims to address the identified requirements of MLDEs as well. First, *PML* uses existing language editors for *visualization* as much as possible. Second, our work on *navigation* (see Chapter 5) of both intra-language and inter-language elements may be applied to single-language and multi-language perspectives. Third, *static checking* could be based on the mappings and consistency conditions of a perspective. Finally, perspective actions could be used to fulfil the *refactoring* requirement.

Combemale *et al.* [35] present Concern-Oriented Language Development (COLD), which aims to promote modularity and reusability of language concerns. The COLD approach customizes a given DSL to conform to the architecture of a language concern, which includes three key interfaces: variability, customization, and usage interfaces. These interfaces facilitate the use of a language concern across different multi-modelling environment domains. The key difference between COLD and PML approaches is that COLD aims to promote modularity and reusability, while *PML*, in addition, promotes consistency between different language models. Also, COLD customizes a given language, whereas *PML* does not alter existing languages. Instead, *PML* combines different languages using the virtual single underlying model approach and then augments the language actions to promote reusability, modularity, and consistency.

Furthermore, König and Diskin [85] present a framework as well as an algorithm to check the consistency of inter-related models based on a specified constraint conditions. The authors favour *localization* approach against the *matching* and *merging* approaches. The localization focuses on the model elements that are affected by the concerned constraints, unlike the matching and merging approaches, which evaluate the whole model to detect sameness relationships/overlaps. This work focuses on model elements that are affected by the established constraints in a multi-model system, which is similar to our approach that focuses on language actions and the concerned model elements to prevent inconsistencies. However, the main difference between this work and *PML* is that we capture the constraints and then ensure (proactively) that the related models are consistent, but the authors do not enforce consistencies. Similarly, Leblebici *et al.* [86] present an approach for inter-model consistency checking by combining Triple Graph Grammars [87] with linear optimization techniques. However, unlike our approach, the work does not repair or prevent inconsistency.

To support the consistent evolution of models based on its metamodel as well as the intra-model constraints, Burdusel *et al.* [88] present automatic consistency preserving search operators based on Model-Driven Engineering and Search-Based Software Engineering [89] methodologies. This work aims to generate mutation operators that can edit or repair model elements with the aim of producing a model that satisfies the desired constraints. This work focuses on maintaining intra-model consistency, unlike our approach that handles inter-model consistency across language boundaries.

10.4 Navigation Related Work

This section compares *PML* navigation mechanism against contemporary navigation facilities among popular modelling tools. Navigation is an important mechanism to traverse, search, and retrieve information. Many studies have been done on how to improve navigation in software applications and web sites.

Santos et al. [12] investigate the effects of different types of menus in web site navigation, assessing the usability as well as performance of 8 different navigation mechanisms, each with distinctive properties. The study concludes by putting forward a horizontal menu, which is the base structure of the navigation bar presented in this thesis.

Burrel and Sodan [90] analyze six different types of menus contained in web pages of institutions. Considering the factors layout, ease of use, clarity of information, and ease of learning, they determine that navigation consisting of tabs, side navigation bars at the top and vertical menus on the left were the most favourite. We considered these insights when developing the navigation bar proposed in this thesis.

Finally, Kitajima et al. [91] present *CoLiDeS* (Comprehension-based Linked model of Deliberate Search), which is a model-based design methodology that website developers can follow to design better navigation for webpages. The main objective is to improve the user's success rate while searching for information on typical web sites.

To the best of our knowledge, there has been no prior work specifically on generic navigation for graphical modelling tools. Programming IDEs typically offer contextual menus that allow a developer to navigate within and across source code modules, e.g., from a method call to the method declaration. These relationships are typically inferred from static source code analysis. The following work targets advanced navigation in programming IDEs, and as such can also be applied for navigation in textual modelling languages.

Mylyn is a task and application lifecycle management (ALM) framework for the Eclipse IDE [39, 92]. In Mylyn, a developer can define tasks and declare which tasks he is currently working on. Mylyn then keeps track of code elements that are being looked at, created, or modified for each task. The developer can then use this information for task-based navigation.

Similarly, the FEAT plugin for Eclipse [93] allows the developer to define a high-level conceptual unit called *concern*, e.g., a feature, a nonfunctional requirement, a design idiom, or an implementation mechanism. When coding, a developer can deliberately associate code elements to the concern, slowly building up a concern graph that relates code elements that are scattered throughout multiple source code modules. Subsequently, the developer can use the concern graph for highlighting and navigation purposes.

10.5 Summary

This chapter compares the *PML* framework with contemporary multi-language modelling approaches. We further investigate the position of our navigation mechanism against other navigation facilities.

In all, our approach offers unique contributions to the software engineering community. First, we manipulate language actions to enforce consistency between language models. Second, we keep languages separate and then externally establish consistency relationships between language elements across the language boundaries. The consistency relationships, as well as other configurations, are encoded as a perspective with our DSLs. Furthermore, we generate the implementation of each perspective with our code generator, which is based on the generic template. The generated implementation oversees language registrations, *LEMs*, as well as perspective actions, which maintain consistency during modelling time.

Furthermore, our consistency rules cover multiplicity constraints, including *language role* multiplicity and *model* multiplicity, as well as conditional *LEM*. These contributions aim to promote consistency, reusability, and coordination in a multi-language modelling environment. Furthermore, we keep the languages separate to promote maintainability and evolution of both software systems and modelling tools.

In addition, we compare the *PML* navigation mechanism against contemporary navigation facilities among popular modelling tools. The comparison shows that navigation is an important feature in a modelling tool and our approach supports most of the current navigation facilities in a generic way. The next chapter presents the conclusion of this doctoral research work as well as the future work.

Chapter 11

Conclusion and Future Work

In this chapter, we first summarize this doctoral research work in Section 11.1 and then present the future work and research directions of this in Section 11.2.

11.1 Summary

Model-driven engineering is a conceptual development framework where models of the system under development are created and manipulated using different languages at different levels of abstraction. Separation of concerns is further promoted when working with multi-view modelling and domain-specific modelling languages. While this separation into many interrelated languages has many benefits, grouping the languages and then maintaining consistent relationships among models conforming to these languages is a non-trivial task.

In this doctoral research, we present *Perspectives for Multi-Language Systems* (*PML*) for maintaining consistency conditions including equivalency, equality, and multiplicity constraints across different model elements from different languages. *PML* is a framework that promotes modular combination of languages and facilitates consistency and reuse of an existing language across other languages and software systems. To this end, we have implemented two DSLs which can assist a perspective designer to combine different languages, specify perspective actions, and encode the relationships between different language elements with mappings, i.e., language element mappings (*LEMs*), even if cyclic consistency relationships and complex language actions affecting several elements with *LEMs* exist. For each *LEM* (which contains the two involved language elements, multiplicities, potentially nested *LEMs*), and optional constraints, we generate different perspective actions with the help of our code generator which is based on our generic templates to prevent inconsistencies at run-time. This allows the perspective designers to focus on these key relationships and frees them from the error-prone implementation of perspective actions.

As proof-of-concept, we implement our approach in the TouchCORE tool, and illustrate our approach by combining five different modelling languages (class diagrams, use case diagrams, collaboration diagrams, use case maps, and a domain-specific modelling language) for the purpose of requirement elicitation. This case study shows (i) that all proposed perspective actions are needed to maintain consistency conditions in a multi-language modelling environment, (ii) that all templates are needed (one for update, three for delete, and twelve for create), and (iii) that only a small percentage of perspective actions involve complex language actions requiring a larger specification effort.

Furthermore, we validate our approach with two notable multi-language modelling technologies: *User Requirements Notation* and *Palladio Component Model*. Here, we focus on the relationships between different languages in each *perspective* and then show how the *PML* framework can be applied to specify those relationships and then preserve the consistencies during the modelling time. The two use cases show that our framework supports their consistency rules, including advanced features such as conditional mapping and language set roles.

Another important concept in a multi-language modelling environment is the navigation of model elements. In this work, we propose a metamodel that covers two categories of navigation, intra-language and inter-language navigation. The metamodel allows the designer of a perspective or a modelling tool to generically capture the relevant navigation links between model elements in a set of models manipulated for a given purpose. It is done by establishing inter-language and intra-language mappings designating the relevant metaclasses and references in the metamodels of the involved languages.

We illustrated the effectiveness of our navigation metamodel by examples that involved feature models, class diagrams, and sequence diagrams, but our approach can be applied to any modelling language that is defined by a metamodel. Furthermore, we validated that our generic navigation approach covers the navigation facilities provided by current modelling tools by conducting a survey of six popular UML modelling tools.

11.2 Future Work

In this section, we present the future work of this doctoral research work as follows:

- Our approach currently focuses on how a perspective reuses/combines existing languages (e.g., class diagram, use cases, and sequence diagram languages). However, a perspective is also a language, as shown in Figure 4.7 on page 66. This concept aims to allow perspective to reuse another perspective, rather than always defining a new perspective from scratch. However, this perspective reuse hierarchy has not yet been implemented in the *PML* framework. In future work, a reuse hierarchy can be built on top of the existing *PML* infrastructure.

- In addition, as detailed in Chapter 3, a language metamodel can be configured to create different single-language perspectives. For example, a class diagram language metamodel can be configured to create a domain modelling, a design modelling, as well as a metamodelling perspective. In future work, our aim is to define a configuration language that accepts a language metamodel as input with the help of software product line (SPL) [5] technology. In this case, each feature of the SPL represents a language action, which can be *re-exposed* or *hidden*. This framework allows perspective designers to select different features, i.e., configuration, which aligns with the language actions to be *re-exposed*, and then generate the implementation of the configured single-language perspective.

- Moreover, in future work, we plan to support languages that do not have metamodels, provided that the language in question has language actions. With this approach, our framework provides a mechanism for generating or manually implementing a metamodel for the concerned language based on its language actions. This feature aims to broaden the scope of the languages that are supported by this work.

- Furthermore, currently, our approach focuses on graphical modelling languages. In future work, we plan to investigate how to apply the *PML* framework to support textual languages, which are also models. Most of the textual languages may not have metamodels. Hence, this feature builds on top of the support for non-metamodel languages as mentioned in the previous paragraph.

- Additionally, in this doctoral research work, we cover four cardinalities, i.e., 1, 0..1, 1..*, and 1..*. In future work, our aim is to improve the multiplicity constraints so that perspective designers can specify any cardinalities of their choice. For example, 1..2, 0..5, 2..5, etc, will be supported.

- Furthermore, we plan to automatically detect the effects of language actions to further streamline the specification of perspectives. With this automated detection of language action effects, perspective designers will not be required to manually specify the effects of language actions during the definition of a perspective.

- Additionally, a perspective designer can specify *LEMs* that cannot be handled with our framework. Examples of such *LEMs* include complex mappings that can introduce ambiguous circular dependencies. In future work, our aim is to detect such ambiguous *LEMs* specifications during the definition of perspectives.

- Also, we aim to conduct an empirical user study to determine the effort required to specify perspectives with our DSLs. Further avenues that could be investigated to expand the support offered by *PML* to designers include assistance in identifying potentially interacting language actions/effects during the specification of a perspective.

- Although the results of the case studies (see Chapter 9) are promising, more studies with additional related languages are needed to confirm our results. All the case studies in this thesis are based on graphical modelling languages. In future, we aim to incorporate textual modelling languages that satisfy our assumptions, i.e., each language has a metamodel and language actions.

- Finally, considering the *PML* navigation, we plan to examine the navigation facilities of non-UML modelling tools to ensure that our generic navigation approach can cover them. Finally, we will carry out an empirical user study to evaluate the usability of the navigation facilities offered by our navigation bar.

Bibliography

[1] F. P. Brooks Jr, “The mythical man-month (anniversary ed.),” *Addison-Wesley Longman Publishing Co., Inc.*, 1995.

[2] R. Fairley, *Software engineering concepts.* McGraw-Hill, Inc., 1985.

[3] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, “The structure and value of modularity in software design,” *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.

[4] W. L. Hürsch and C. V. Lopes, “Separation of concerns,” *Northeastern University, Boston, Massachusetts (Technical Report NU-CCS-95-03)*, 1995.

[5] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

[6] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice.* Morgan & Claypool Publishers, 2012.

[7] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, “Multi-view approaches for software and system modelling: a systematic literature review,” *Software and Systems Modeling*, vol. 18, no. 6, pp. 3207–3233, 2019.

[8] F. G. Marinho, "A proposal for consistency checking in dynamic software product line models using OCL," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 2, p. 333, 2010.

[9] M. E. Kramer, E. Burger, and M. Langhammer, "View-centric engineering with synchronized heterogeneous models," in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, pp. 1–6, 2013.

[10] D. V. Beard and J. Q. W. II, "Navigational techniques to improve the display of large two-dimensional spaces," *Behaviour & Information Technology*, vol. 9, no. 6, pp. 451–466, 1990.

[11] J. D. Mackinlay, G. G. Robertson, and S. K. Card, "The perspective wall: Detail and context smoothly integrated," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 173–176, ACM, 1991.

[12] E. P. dos Santos, S. de Lara, W. M. Watanabe, R. P. Fortes, *et al.*, "Usability evaluation of horizontal navigation bar with drop-down menus by middle aged adults," in *Proceedings of the 29th ACM international conference on Design of communication*, pp. 145–150, ACM, 2011.

[13] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic software modeling: a practical approach to view-based development," in *Evaluation of Novel Approaches to Software Engineering*, pp. 206–219, Springer, 2009.

[14] T. Goldschmidt, S. Becker, and E. Burger, "Towards a tool-oriented taxonomy of view-based modelling," *Modellierung 2012*, pp. 59–74, 2012.

[15] E. J. Burger, "Flexible views for view-based model-driven development," in *Proceedings of the 18th international doctoral symposium on Components and architecture*, pp. 25–30, ACM, 2013.

[16] TouchCORE Website, "Touchcore v7.0.2." `http://touchcore.cs.mcgill.ca/`. Accessed: 2019-02-01.

[17] EMF Website, "Eclipse modeling framework (emf)." `https://www.eclipse.org/modeling/emf/`. Accessed: 2022-02-10.

[18] A. W. Brown, "Model driven architecture: Principles and practice," *Software and systems modeling*, vol. 3, no. 4, pp. 314–327, 2004.

[19] D. Varró, "Model transformation by example," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 410–424, Springer, 2006.

[20] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic notes in theoretical computer science*, vol. 152, pp. 125–142, 2006.

[21] T. Stahl, M. Völter, and K. Czarnecki, *Model-driven software development: technology, engineering, management.* John Wiley & Sons, Inc., 2006.

[22] A. Watson, "Omg (object management group) architecture and corba (common object request broker architecture) specification," in *IEE Colloquium on Distributed Object Management*, pp. 4–1, IET, 1994.

[23] S. Beydeda, M. Book, V. Gruhn, *et al.*, *Model-driven software development*, vol. 15. Springer, 2005.

[24] E. Seidewitz, "What models mean," *IEEE software*, vol. 20, no. 5, pp. 26–32, 2003.

[25] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels.* Pearson Education, 2008.

[26] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[27] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[28] M. Fowler, *Domain-specific languages.* Pearson Education, 2010.

[29] A. V. Deursen and P. Klint, "Little languages: little maintenance?," *Journal of Software Maintenance: Research and Practice*, vol. 10, no. 2, pp. 75–92, 1998.

[30] T. Degueule, *Composition and interoperability for external domain-specific language engineering.* PhD thesis, University of Rennes 1, 2016.

[31] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, *et al.*, "The fortran automatic coding system," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pp. 188–198, ACM, 1957.

[32] J.-M. Favre, D. Gasevic, R. Lämmel, and E. Pek, "Empirical language analysis in software linguistics," in *International Conference on Software Language Engineering*, pp. 316–326, Springer, 2010.

[33] D. Spinellis, "Notable design patterns for domain-specific languages," *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[34] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[35] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, *et al.*, "Concern-oriented language development (cold): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, vol. 54, pp. 139–155, 2018.

[36] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, (GBR), p. 129–131, John Wiley and Sons Ltd., 2003.

[37] I. Poernomo, "The meta-object facility typed," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1845–1849, ACM, 2006.

[38] J. Overbeek, "Meta object facility (mof): investigation of the state of the art," Master's thesis, University of Twente, 2006.

[39] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 1–11, ACM, 2006.

[40] Eclipse Website, "Emf tutorial." `http://eclipsesource.com/blogs/tutorials/emf-tutorial/`. Accessed: 2022-03-31.

[41] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers, "Bridging the ms/dsl tools and the eclipse modeling framework," in *Proceedings of the International Workshop on Software Factories at OOPSLA*, vol. 5, pp. 1–19, 2005.

[42] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[43] J. Bézivin, "In search of a basic principle for model driven engineering," *Novatica Journal, Special Issue*, vol. 5, no. 2, pp. 21–24, 2004.

[44] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 604–621, Springer, 2013.

[45] O. Alam, *Concern oriented reuse: a software reuse paradigm.* PhD thesis, McGill University, Montreal, Canada, 2016.

[46] W. Al Abed, V. Bonnet, M. Schöttle, O. Alam, and J. Kienzle, "TouchRAM: A multitouch-enabled tool for aspect-oriented software design," in *5th International Conference on Software Language Engineering - SLE 2012*, no. 7745 in LNCS, pp. 275 – 285, Springer, October 2012.

[47] Y. Jin and A. Olsson, "Design and implementation for report layout merging," *Bachelor of Science Thesis, University of Gothenburg*, 2012.

[48] H. Ali, G. Mussbacher, and J. Kienzle, "Towards modular combination and reuse of languages with perspectives," in *View-oriented Software Engineering (VoSE) 2019 Workshop, Proceedings of 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 387–394, IEEE, 2019.

[49] H. Ali, "Multi-language systems based on perspectives to promote modularity, reusability, and consistency," in *MODELS 2020 Doctoral Symposium, Proceedings of*

the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–6, 2020.

[50] StarUML Website, "Staruml." `http://staruml.io/`. Accessed: 2019-04-22.

[51] ArgoUML Website, "Argouml - free, opensource uml engineering tool." `http://argouml.tigris.org/index.html`. Accessed: 2019-03-29.

[52] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[53] Xtend Website. `https://www.eclipse.org/xtend`. Accessed: 2022-03-19.

[54] A. Strohmeier, T. Baar, and S. Sendall, "Applying fondue to specify a drink vending machine," *Electron. Notes Theor. Comput. Sci.*, vol. 102, pp. 155–173, 2004.

[55] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method.* Englewood Cliffs: Prentice-Hall, 1994.

[56] D. Amyot and G. Mussbacher, "User requirements notation: the first ten years, the next ten years," *J. Softw.*, vol. 6, no. 5, pp. 747–768, 2011.

[57] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[58] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Addison Wesley, Boston*, vol. 5, 2001.

[59] MagicDraw. `https://www.nomagic.com/products/magicdraw`. Accessed: 2019-04-01.

[60] Visual Paradigm Website, "Visual paradigm - ideal modeling & diagramming tool for agile team collaboration." `https://www.visual-paradigm.com/`. Accessed: 2019-04-29.

[61] J. Meier, H. Klare, C. Tunjic, C. Atkinson, E. Burger, R. Reussner, and A. Winter, "Single underlying models for projectional, multi-view environments," in *7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, pp. 119–130, INSTICC, SciTePress, 2019.

[62] A. A. Shah, A. A. Kerzhner, D. Schaefer, and C. J. Paredis, "Multi-view modeling to support embedded systems engineering in sysml," in *Graph transformations and model-driven engineering*, pp. 580–601, Springer, 2010.

[63] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A meta-language for modular and reusable development of dsls," in *Proceedings of the 2015 ACM SIGPLAN Intl. Conference on Software Language Engineering*, pp. 25–36, ACM, 2015.

[64] J. Lin, A. Poudel, W. Yu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Enhancing automated software traceability by transfer learning from open-world data," *arXiv preprint arXiv:2207.01084*, 2022.

[65] Radio Technical Commission for Aeronautics (RTCA), "Software considerations in airborne systems and equipment certification," *Document No. RTCA/DO-178B*, 1992.

[66] Bel V, Canadian Nuclear Safety Commission, and others, "Licensing of safety critical software for nuclear reactors: Common position of international nuclear regulators and authorised technical support organisations," 2022.

[67] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mäder, "Traceability fundamentals," in *Software and systems traceability*, pp. 3–22, Springer, 2012.

[68] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[69] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 3–14, IEEE, 2017.

[70] Á. Hegedüs, Á. Horváth, I. Ráth, R. R. Starr, and D. Varró, "Query-driven soft traceability links for models," *Software & Systems Modeling*, vol. 15, no. 3, pp. 733–756, 2016.

[71] M. E. Kramer, "A generative approach to change-driven consistency in multi-view modeling," in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 129–134, ACM, 2015.

[72] M. E. Kramer, M. Langhammer, D. Messinger, S. Seifermann, and E. Burger, "Change-driven consistency for component code, architectural models, and contracts," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, pp. 21–26, ACM, 2015.

[73] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 01, pp. 31–57, 1992.

[74] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, "A feature-based survey of model view approaches," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1931–1952, 2019.

[75] K. Marussy, O. Semeráth, and D. Varró, "Incremental view model synchronization using partial models," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 323–333, 2018.

[76] H. Bruneliere, F. M. de Kerchove, G. Daniel, S. Madani, D. Kolovos, and J. Cabot, "Scalable model views over heterogeneous modeling technologies and resources," *Software and Systems Modeling*, vol. 19, no. 4, pp. 827–851, 2020.

[77] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.

[78] J.-M. Favre and T. Nguyen, "Towards a megamodel to model software evolution through transformations," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 59–74, 2005.

[79] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray, "Globalizing modeling languages," *Computer*, vol. 47, no. 6, pp. 68–71, 2014.

[80] J. Deantoni, "Modeling the behavioral semantics of heterogeneous languages and their coordination," in *2016 Architecture-Centric Virtual Integration (ACVI)*, pp. 12–18, IEEE, 2016.

[81] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000.

[82] D. Kolovos, R. Paige, and F. Polack, "Detecting and repairing inconsistencies across heterogeneous models," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 356–364, IEEE, 2008.

[83] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Jtl: a bidirectional and change propagating transformation language," in *Intl. Conf. on Software Language Engineering*, pp. 183–202, Springer, 2010.

[84] R.-H. Pfeiffer and A. Wasowski, "Texmo: A multi-language development environment," in *European Conference on Modelling Foundations and Applications*, pp. 178–193, Springer, 2012.

[85] H. König and Z. Diskin, "Efficient consistency checking of interrelated models," in *Modelling Foundations and Applications* (A. Anjorin and H. Espinoza, eds.), (Cham), pp. 161–178, Springer International Publishing, 2017.

[86] E. Leblebici, A. Anjorin, and A. Schürr, "Inter-model consistency checking using triple graph grammars and linear optimization techniques," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 191–207, Springer, 2017.

[87] H. L. Bodlaender, J. S. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Z. Tuza, "Rankings of graphs," in *International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 292–304, Springer, 1994.

[88] A. Burdusel, S. Zschaler, and S. John, "Automatic generation of atomic consistency preserving search operators for search-based model engineering," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 106–116, IEEE, 2019.

[89] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[90] A. Burrell and A. C. Sodan, "Web interface navigation design: which style of navigation-link menus do users prefer?," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pp. 42–42, IEEE, 2006.

[91] M. Kitajima, M. H. Blackmon, and P. G. Polson, "A comprehension-based model of web navigation and its application to web usability analysis," in *People and computers XIV—Usability or else!*, pp. 357–373, Springer, 2000.

[92] Eclipse Website, "Mylyn." `https://www.eclipse.org/mylyn/`. Accessed: February 2, 2019-02-02.

[93] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, pp. 3–es, 2007.

Appendix A

Generic Templates

In this appendix, we present the generic templates used by *PML* to generate the implementation of a given a perspective. Section A.1 introduces the update template, Section A.2 the delete templates, and Section A.3 the create templates. Brief definitions of all methods and the helper functions in the templates are shown in Tables A.1 and A.2, respectively.

A.1 Update Template

The update template aims to ensure that mapped model elements are always consistent. Hence, when a model is modified, the update is propagated to both directly and indirectly mapped model elements. For example, in the context of the perspective action, *update an operation*, a direct mapping is the mapping between an *operation* in a class diagram language and a *sequence diagram.* However, if the *sequence diagram* is also mapped to a *responsibility* in a use case map language, then an indirect mapping exists between the operation and the responsibility. When the *operation* is modified (e.g., change of name), the modification is propagated to the mapped *sequence diagram* and then recursively to the corresponding *responsibility* in the use case map language. The generic *update* template shown in Algorithm 2 is the same across all possible *LEM* mappings.

Algorithm 2: redefinedUpdateElement(element, attribute, value, p4, p5, ...)

```
1 updatedAttributes = new ArrayList<EObject>()
2 Map<EObject, List<EObject>> before = getExistingElements(secondaryEffects)
3 // call language action to update model element
4 update(element, attribute, value, p4, p5, ...)
5 Map<EObject, List<EObject>> after = getExistingElements(secondaryEffects)
6 // Call recursive action to handle the update of other mapped elements
7 updateOtherMappedElements(element, attribute, value, updatedAttributes, p4, p5,
  ...)
8 // handle secondary effects
9 handleSecondaryEffects(secondaryEffects, before, after, updatedAttributes, p4, p5,
  ...)
```

Algorithm 3: updateOtherMappedElements(element, attribute, value, updatedAttributes, p4, p5, ...)

```
1 updatedAttributes.add(attribute)
2 for mapping : getMappings(element) do
3     other = getOther(mapping, element)
4     nestedMapping = getNestedMapping(mapping, attribute)
5     otherAttribute = getOtherAttribute(nestedMapping, other)
6     // perspective recursive call
7     if !updatedAttributes.contains(otherAttribute) then
8         UPDATE(other, otherAttribute, value, p4, p5, ...)
9         updateOtherMappedElements(other, otherAttribute, value,
          updatedAttributes, p4, p5, ...)
```

Method	**Definition**
canCreateOrUseElement(...)	This method optionally creates a corresponding element and then establishes a *MEM* between the element and the original element. Alternatively, the method can use a corresponding existing element to establish the *MEM* with the original element
create(...)	This is the language action that creates the element in question
CREATE(...)	This is a facade action that calls the corresponding *create* language actions to create an element as a result of creating the original element.
createAtLeastOneElement(...)	This method creates at least one corresponding element and then establishes *MEMs* between the elements and the original element
createElement(...)	This method creates a corresponding element and then establishes a *MEM* between the element and the original element.
createFacadeOther(...)	This facade action calls the corresponding *createOtherRequiredElements* method to propagate the *create* secondary effects
createOrUseNonMappedElement(...)	This method creates a corresponding element and then establishes a *MEM* between the element and the original element. Alternatively, the method uses an existing corresponding element, which is not yet mapped (i.e., *MEM*) to establish the mapping with the original element
createOtherRequiredElements(...)	This method propagates the effects of creating an element to keep the concerned models consistent
delete(...)	This is the language action that deletes the element in question
DELETE(...)	This is a facade action that calls the corresponding *delete* language actions to delete each mapped element with the originally deleted element
deleteFacadeOther(...)	This facade action calls the corresponding *deleteOtherMappedElements* method to propagate the *delete* secondary effects
deleteOtherMappedElements(...)	This method propagates the effects of deleting an element to all the other mapped elements
handleSecondaryEffects(...)	This method propagates the secondary effects of creating/deleting/updating an element to keep the concerned models consistent
redefinedCreateElement(...)	This is a *create redefined* perspective action
redefinedDeleteElement(...)	This is a *delete redefined* perspective action
redefinedUpdateElement(...)	This is an *update redefined* perspective action
update(...)	This is the language action that updates the element in question
UPDATE(...)	This is a facade action that calls corresponding *update* language actions to update each mapped element with the originally updated element
updateFacadeOther(...)	This facade action calls the corresponding *updateOtherMappedElements* method to propagate the *update* secondary effects
updateOtherMappedElements(...)	This method propagates the effects of updating an element to all the other mapped elements

Helper Method	**Definition**
askNumberOfMappingsAtLeastOne()	Asks a user to provide the number of mappings (at-least one) that need to be established between the new element and the other elements, either existing or to be created
createMapping(type, element, other)	Establishes *MEM* between the two elements (*element* and *other*) based on the *type*
diff(...)	Retrieves the new elements (instances of the *affectedLanguageeLement*) due to the secondary effects of the action in question. Basically, the method finds the difference (i.e., diff) between the existing element of the *affectedLanguageElement* before and after the execution of the action
findCorrespondingElement()	Finds an existing element which can be mapped with the new element
getAffectedElement(parameterEffect)	Returns either the updated or deleted element based on the *parameterrEffect* value.
getCreateType(type)	Retrieves the create type based on the *LEM* (i.e., *type*). The create type determines if one or more other elements need to created because of creating the current element.
getExistingElements(secondaryEffects)	Retrieves all the existing model elements for all the affected (secondary effects) language elements (metaclasses) as well as the elements of the metaclass in question
getMappings(element)	Retrieves all the *MEMs* of the *element*, irrespective of the mapping type.
getMappings(mappingType, other)	Retrieves all the *MEMs*, each references the *other* element and the type of the mapping is *mappingType*
getMappingTypes(element)	Similar to getMappings(element), this method retrieves all the *LEMs* with a mapping-end that references the language element of the *element*.
getNestedMapping(mapping, attribute)	Retrieves the nested *MEM* of a *mapping* which references the *attribute* as one of the mapped element
getNewElement(languageElement, before)	Retrieves the newly created element of the *languageElement*
getOther(mapping, element)	Retrieves the other element which is mapped (*MEM*) with the *element*
getOtherAttribute(nestedMapping, other)	Similar to getOther(mapping, element), this helper method retrieves the *other* attribute which is mapped, i.e., nested *MEM*, with an attribute of the current element.
getOtherMetaClass(type, element)	Similar to getOther(mapping, element), this method retrieves the other language element which is mapped (LEM) with the *element*
isCreateMapping()	Asks the user whether to create a mapping (*MEM*) between the new element and other element, either existing or to be created

Table A.2: Definition of Helper Methods

A.1.1 Update Element

When a user initiates to modify a model element (e.g., change the name of an operation), the *redefined* perspective action, *redefinedUpdateElement* (Algorithm 2), is called. The *redefinedUpdateElement* (i.e., *Redefined Update Action* in Figure 6.1), first creates an empty list, i.e., *updatedAttributes*, at line 1, to track the attributes of model elements that have already been updated during the chain of the respective model element updates. For instance, a class (model element) in a class diagram model can have attributes such as *visibility*, *name*, and *abstract*. During the chains of updates, each updated element attribute is added to the list (*updatedAttributes*) and thus prevents further update on the same attribute during the recursion; hence, cyclic consistency relationships are properly handled and the recursion stops eventually.

At line 2, the generic *update* template initializes the map variable *before* between a language element (*key*) and its instances (*value*). The parameter, *secondaryEffects*, represents the secondary effects of the *update* action (see *PML* metamodel in Figure 4.7). This map keeps a record of the existing instances (*value*) of each affected language element (*key*) before the *update* language action is called. This set of existing model elements allows the generic template to retrieve the *new* elements after the execution of the language action in question. This is needed, because an *update* action can have a *create* action as secondary effect(s) which need to be handled to adequately maintain the consistency conditions of the perspective.

At line 4, the update template calls the original language action to update the model element with the parameters, which include *element* (the element in question), *attribute* (the attribute of the element whose value needs to be updated, e.g., *name* attribute of a `Class`), and *value* (the new value of the attribute, e.g., the new name of a class). Similar to the *before* map variable, at line 5, the template initializes another map variable (*after*) which, at this juncture, references existing instances of each affected language element, including the *newly* created element(s). The difference between the instances of each language element in both *before* and *after* variable allows the template to retrieve the newly created element(s).

The execution of this *update* action may require the concerned *perspective* action to update other model element(s), as specified in the *LEMs* of the *perspective*. Hence, the update template calls *updateOtherMappedElements* (line 7) which handles the update of both directly and indirectly mapped element with the *primary* model element (i.e., handling the *primary* effect of the update action).

Also, the execution of this update action at line 4 can automatically trigger other actions which impact additional *LEMs* and which need to be handled as the *secondary* effects of the language action. For example, when an *update operation* is called to change the *visibility* of the operation to *public*, this action may also require to change the *visibility* of the containing class of the operation to *public*. However, the perspective may require that whenever the *visibility* of a class is modified, the corresponding mapped elements with the class need to be updated accordingly. These chained effects of a language action are handled as the *secondary effects*, see Section 6.3. As *redefinedUpdateElement* redefines a language action, the template has access to all the attributes of the *redefined* language action (`LanguageAction`), including the *secondaryEffects* (see Figure 4.7). Hence, the update template calls *handleSecondaryEffects* with the *secondaryEffects* parameter at line 9 to propagate the *secondary* effects of the *update* language action. *before* and *after* are passed as parameters to allow *handleSecondaryEffects* to retrieve the *newly* created model elements.

A.1.2 Update Other Elements

In this section, we demonstrate how the generic update template recursively propagates the effects of executing an *update* language action as shown in Algorithm 3. At line 1, *updateOtherMappedElements* adds the recently updated attribute to the list of the updated attributes. Furthermore, the template ensures that the corresponding mapped elements are updated accordingly by iterating over all the *MEMs* of the element in question (lines 2-9). For each mapping, we get the corresponding mapped element (line 3), and then retrieve its nested mapping which references the updated attribute (line 4). Furthermore, the template retrieves the attribute (*otherAttribute*) of the mapped element (line 5) and then checks if the attribute has already been updated at line 7. Otherwise, the template calls an *UPDATE* method (line 8) which acts as a facade to call the actual update language action to update the corresponding model element. Finally, the template recursively calls *updateOtherMappedElements* with the corresponding mapped model element (*other*), the attribute of the corresponding mapped model element (*attribute*), and the new value (line 9). This iteration continues until all the mapped model elements (directly or indirectly) with the *primary* element have been duly updated.

A.1.3 Handle Secondary Effects

The generic template (Algorithm 4) shows the steps for handling the secondary effects of a complex language action. It is used for all three kinds of templates: update, delete, and create. For *create* secondary effects (lines 2-7), the algorithm retrieves the new element(s) for each affected language element by computing the difference between the list of elements of the affected language element in *before* and *after* map variables (lines 3-4), and then calls the corresponding *createFacadeOther* (lines 6-7), see Figure 6.1. The primary role of

Algorithm 4: handleSecondaryEffects(secondaryEffects, before, after, updatedAttributes, p4, p5, ...)

```
1  // secondary effects - create type
2  for secondaryEffect : secondaryEffects do
3      affectedLanguageElement = secondaryEffect.getLanguageElement()
4      newElements = diff(after.get(affectedLanguageElement),
         before.get(affectedLanguageElement))
5      // Calls recursive action
6      for element : newElements do
7          // call facade other
8          createFacadeOther(element, p4, p5, ...)
9  // secondary effects - update and delete types
10 for secondaryEffect : secondaryEffects do
11     affectedElement = getAffectedElement(parameterEffect)
12     if secondaryEffect == UPDATE then
13         affectedAttribute = secondaryEffect.affectedAttribute
14         // call facade other
15         updateFacadeOther(affectedElement, affectedAttribute, value,
           updatedAttributes, p4, p5, ...)
16     else if secondaryEffect == DELETE then
17         // call facade other
18         deleteFacadeOther(affectedElement, p4, p5, ...)
```

createFacadeOther is to derive the corresponding *createOtherRequiredElements* arguments from the parameters of *handleSecondaryEffects*, and then to call *createOtherRequiredElements* to propagate the required changes as shown in Figure 6.1.

For the *update* and *delete* secondary effects (lines 9-17), the template first retrieves the affected model element (line 10), i.e., the element that was deleted or updated. If the *secondary* effect is *update* (line 11), the template further retrieves the affected attribute of the element (line 12), and then calls the *updateFacadeOther* to propagate the update changes. Similar to *createFacadeOther*, the *updateFacadeOther* derives the arguments of the corresponding
updateOtherMappedElements from the parameters of *handleSecondaryEffects*. On the other hand, if the *secondary* effect is *delete* (line 15), the template calls the corresponding *deleteFacadeOther* (line 17) to propagate the effect(s) of the *secondary* delete action.

A.1.4 Generating the Perspective Actions from the Template

While the structure of the *update* generic template is the same for each perspective action that redefines a language action that updates model elements, the parameters (element, attribute, value, p4, p5, ...) (see Algorithm 2) differ from one language action to the next. Hence, the *redefined* perspective method (Algorithm 2), the *handleSecondaryEffects* method (Algorithm 4), the *createFacadeOther* (line 7 in Algorithm 4), the *updateFacadeOther* (line 14 in Algorithm 4), the *deleteFacadeOther* (line 17 in Algorithm 4), the recursive method *updateOtherMappedElements* (Algorithm 3), and the UPDATE facade method (line 8 in Algorithm 3) are generated for each update language action. Note that we choose to depict the *element*, *attribute*, and *value* parameters as the first three parameters of the methods, but they do not need to be the first three parameters and may appear anywhere in the list of parameters. The parameters, *p4, p5, ...*, represent other potential parameters that can be part of the language action parameters. This approach allows the same template to be applied across different language actions that may have a different set of parameters. Generally, the unknown parameters are in light green font color in all the generic templates to highlight variable parts of the template that are customized to a specific *redefined* perspective action. Also, some method names are highlighted with light green color to indicate that they are customized to a specific *redefined* perspective action.

The perspective designer specifies the parameters for each language action in the DSL for *PML* in addition to the actual qualified name of the language action. The qualified name and the parameters are then used in the facade methods. The methods *updateFacadeOther*, *deleteFacadeOther*, and *createFacadeOther* derive the respective parameters of *updateOtherMappedElements*, *deleteOtherMappedElements*, and *createOtherRequiredElements* from the initial parameters. However, the *UPDATE* facade method handles the parameter derivation to the corresponding language actions. At run-time, the *UPDATE* method determines which actual language action to call based on the parameter *element.* Our code generator fully generates *updateFacadeOther*, *deleteFacadeOther*, *createFacadeOther*, and the *UPDATE* facade method. How to derive the parameters has to be defined by the the perspective designer during the perspective specification.

Consider that an *operation* model element is mapped to a *sequence diagram* element and the *sequence diagram* is also mapped to a *responsibility* in a use case model language. The *redefined* update perspective action of the operation directly calls the *update operation* language action (line 4) in Algorithm 2 with the exact same parameters as the redefined perspective action. The recursive method (Algorithm 3) calls the *UPDATE* facade method with the *sequence diagram* and *responsibilty* as the *element* parameter during the first and second recursion, respectively. The possible types of the element parameter are known from the defined LEMs. During the first recursion, the *UPDATE* facade method derives the parameters of the *update sequence diagram* language action from the parameters of the *update operation* language action and then calls that action. During the second recursion, the *UPDATE* facade method derives the *update responsibility* language action parameters from the parameters of the *update operation* action and then calls that action. The *UPDATE* facade method can derive a parameter by requesting parameter values from the modeller as needed for the language action, using *default* values for the respective parameters, or using the parameters of the facade method. The derivation of parameters in *updateFacadeOther*, *deleteFacadeOther*, and *createFacadeOther* follows a similar pattern.

A.2 Delete Template

To maintain the multiplicity constraints of a *LEM* at run-time, the generic delete templates ensure that when a user deletes a model element, the corresponding mapped model elements, as well as the element mappings, are deleted accordingly. To determine whether a mapped element needs to be deleted, the template evaluates the number of instances for a mapping-end based on the multiplicity constraints in the *LEM*.

There are three types of generic *delete* templates (D1 to D3 as shown in Table 6.1) across all binary mappings:

- **JUST_DELETE** (D2): This type simply deletes the element of interest and its model element mappings (i.e., its links with other mapped elements), because the multiplicity constraints in the *LEM* allow the mapped elements to exist without the element of interest. For example, assume a *Compulsory Optional* mapping between a *Class* metaclass in a class diagram language (Compulsory mapping-end) and an *Actor* metaclass in a use case diagram language (Optional mapping-end). When a modeller decides to delete an instance of an *Actor* metaclass, the *delete* template simply deletes the actor element and does not need to delete the mapped class, because the actor is *Optional* for the class and hence the multiplicity constraint between *Class* and *Actor* is not violated.

- **DELETE_OTHERS** (D1): This type deletes the element of interest, its mappings, and other mapped model elements, because the multiplicity constraints in the *LEM* do not allow *other* mapped elements to exist without the element of interest. For example, deleting an instance of a *Class* in a *Compulsory Optional* mapping between a *Class* and an *Actor*, as shown above, requires that its mappings with the actor element as well as the mapped actor element have to be deleted to maintain the multiplicity constraint as specified in the *LEM*.

- **DELETE_SINGLE_MAPPED** (D3): This type deletes the model element of interest and its mappings. It also deletes any other mapped model element, if the removal of the mappings causes the mapped element to have no mappings left. For example, consider a *Compulsory Compulsory-Multiple* mapping between an *Operation* metaclass in a class diagram language (Compulsory mapping-end) and an *Event* metaclass in a state machine language (Compulsory-Multiple mapping-end). When a user requests to delete an *event* model element, the *delete* template deletes the *event* model element and its mappings, and also the mapped operation element if the operation has no other *MEM* left. This demonstrates that if all event occurrences mapped to the operation have been removed in the state machine, then there is no need for the operation anymore.

Algorithm 5: redefinedDeleteElement(element, p2, p3, ...)

```
1 updatedAttributes = new ArrayList<EObject>()
2 Map<EObject, List<EObject>> before = getExistingElements(secondaryEffects)
3 // call language action to delete model element
4 delete(element, p2, p3, ...)
5 Map<EObject, List<EObject>> after = getExistingElements(secondaryEffects)
6 // Call recursive action to ensure that constraints are maintained
7 deleteOtherMappedElements(element, p2, p3, ...)
8 // handle secondary effects
9 handleSecondaryEffects(secondaryEffects, before, after, updatedAttributes, p2, p3,
  ...)
```

The three generic *delete* templates can be applied across all *MEMs* as detailed in the following templates. In general, the *delete* template follows the same structure as the *update* template (i.e., the language action call, the recursive call to handle primary effects, and then handling of the secondary effects), except that there are three options in the perspective recursive operation (Algorithm 6) instead of one. As for the update template, the parts of the delete templates that are customized to a specific *redefined* action are highlighted with green color. As shown in Algorithm 5, the *redefined* perspective action, *redefinedDeleteElement* (i.e., *Redefined Delete Action* in Figure 6.1), is responsible for calling the actual *delete* language action (line 4), and then requesting to propagate both the *primary* effect (line 7) and the *secondary* effects (line 9) of the *delete* language action with *deleteOtherMappedElements* (Algorithm 6) (i.e., recursive delete operation) and *handleSecondaryEffects* (Algorithm 4), respectively. Hence, deleting of *other* model elements, deleting of *MEMs*, and the constraint validations are handled by the recursive operation *deleteOtherMappedElements*.

before and *after* maps (lines 2 and 5) are again used to determine any newly created

Algorithm 6: deleteOtherMappedElements(element, p2, p3, ...)

```
1  for mapping : getMappings(element) do
2      other = getOther(mapping, element)
3      deleteType = getDeleteType(mapping)
4      mappingType = mapping.getMappingType()
5      mapping.delete()
6      switch deleteType do
7          case JUST_DELETE do
8              // do nothing, action already covered
9          case DELETE_OTHERS do
10             DELETE(other, p2, p3, ...)
11             deleteOtherMappedElements(other, p2, p3, ...)
12         case DELETE_SINGLE_MAPPED do
13             otherMappings = getMappings(mappingType, other)
14             if otherMappings.size() == 0 then
15                 DELETE(other, p2, p3, ...)
16                 deleteOtherMappedElements(other, p2, p3, ...);
```

elements for secondary effects. An example for a secondary effect is the deletion of a *sequence diagram* in a sequence diagram language. The execution of this action can automatically delete a *lifeline* as well as a *lifeline type* in the sequence diagram. However, the *lifeline type* may have been mapped with the `Class` metaclass in the class diagram language and the constraints of this relationship need to maintained by the perspective action. The secondary effects template (Section A.1.3) handles the propagation of these secondary effects.

To effectively propagate the primary effects of the *delete* action across other models, *deleteOtherMappedElements* (Algorithm 6)

iterates over all the *MEMs* of the element in question (line 1-16) to ascertain if any of the other elements is to be deleted as well. For each mapping, the template retrieves the corresponding mapped element (line 2), the type of *delete* based on the mapping-end of the element in question (line 3), the type of mapping (i.e., *LEM*) at line 4, and then deletes the *MEM* (line 5) since the element of interest has already been deleted. If the delete type is *JUST_DELETE*, nothing needs to be done (lines 7-8) since the element and *MEM* have previously been deleted.

On the other hand, if it is the *DELETE_OTHERS* type, the template calls the *DELETE* method (line 10) which acts as a facade to call the actual delete language action, and then recursively calls the operation, *deleteOtherMappedElements*, with the corresponding mapped element and the required parameters (line 11). For the *DELETE_SINGLE_MAPPED* type (lines 12-16), the template checks whether the corresponding element has no *MEM* left, and if yes, the template calls the *DELETE* method (line 15; same as line 10). Finally, *deleteOtherMappedElements* is called recursively. This continues until all the elements, directly or indirectly mapped with the model element of interest are evaluated as described above. Cyclic consistency relationships are not a concern for the delete template, because the recursion continues only when mappings exist. However, each mapping of a deleted element is duly deleted, which prevents the recursion from considering the same mapping more than once when a cyclic relationship exists.

Again, the *redefined* perspective method (Algorithm 5), the handle secondary effects method (Algorithm 4), the *createFacadeOther* (line 7 in Algorithm 4), the *updateFacadeOther* (line 14 in Algorithm 4), the *deleteFacadeOther* (line 17 in Algorithm 4), the recursive method (Algorithm 6), and the DELETE facade method (line 10 and 15 in Algorithm 6) are generated for each delete language action, because the parameters differ for each language action. Similar to the *update* generic template, *createFacadeOther*, *deleteFacadeOther*, *updateFacadeOther*, and the *DELETE* facade method are fully generated with our code generator.

Create Type	***primary* Multiplicity**	***other* Multiplicity**	**Corresponding Element**
CAN_CREATE (**C1**)	1	0..1	Have to Use New Element
CREATE (**C2**)	1	1	
CAN_CREATE_MANY (**C3**)	1	0..*	
CREATE_AT_LEAST_ONE (**C4**)	1	1..*	
CAN_CREATE_OR_USE (**C5**)	0..*	0..1	Can Use Any Existing Element
	1..*	0..1	
CREATE_OR_USE (**C6**)	0..*	1	
	1..*	1	
CAN_CREATE_OR_USE_MANY (**C7**)	0..*	0..*	
	1..*	0..*	
CREATE_OR_USE_AT_LEAST_ONE (**C8**)	0..*	1..*	
	1..*	1..*	
CAN_CREATE_OR_USE_NON_MAPPED (**C9**)	0..1	0..1	Can Use Non-Mapped Existing Element
CREATE_OR_USE_NON_MAPPED (**C10**)	0..1	1	
CAN_CREATE_OR_USE_NON_MAPPED_MANY (**C11**)	0..1	0..*	
CREATE_OR_USE_NON_MAPPED_AT_LEAST_ONE (**C12**)	0..1	1..*	

Table A.3: Types of Create Templates

A.3 Create Template

The generic create templates handle the creation of model elements as well as the multiplicity constraints of their mappings. Proactively, these templates can prevent inconsistency by creating an element being requested by a user as well as *other* model elements required to maintain the multiplicity constraints of the *LEM*. There are twelve types of *create* templates (C1 to C12 as shown in Tables 6.1 and A.3). For each template, the element in question (i.e., *primary* element) is created, while creating or using existing corresponding elements to establish *MEMs* is based on the multiplicities of the *LEM*.

In contrast to the *delete* templates, the *create* templates depend on more than one factor: (a) how many of the other elements are required (i.e., the multiplicity of the corresponding mapping-end) and (b) can existing *other* elements be mapped to the *primary* element, as shown in Table A.3. For corresponding elements, a new element can always be created for the 12 types. However, **C9**, **C10**, **C11**, and **C12** can use existing elements which are not mapped while **C5**, **C6**, **C7**, and **C8** can use any existing element (either mapped or not mapped). Each create type is represented as a case statement in the template shown in Algorithm 8.

The generic *create* template has the same basic structure as the *update* and *delete* templates, i.e., the *redefined* perspective action (Algorithm 7) has a language action call (line 4), a recursive call for primary effects (line 8), and handling of secondary effects (line 10). Again, the parts of the create templates that are customized to a specific *redefined* action are highlighted with green color.

However, the list of *LEMs* instead of *MEMs* need to be retrieved in the recursive template (Algorithm 8), because mappings have to be created according to the *LEMs* defined for the element in question. *before* and *after* maps are again used to determine newly added model elements (lines 2 and 6), including the new *primary* element (line 5). We use this approach because some language actions may not support the *return* of the new *primary* element after executing the *create* language action. Note that the *redefinedCreateElement* (Algorithm 7) represents the *Redefined Create Action* in Figure 6.1.

To propagate the effects of the *create* language action, *createOtherRequiredElements* (Algorithm 8) iterates over all the *LEMs* of the model element (lines 1-17). For each mapping type, we get the mappings (*MEMs*) (line 2), the metaclass of the *other* element to be created (line 3), and the corresponding create type of the model element (line 4). To eventually stop the recursive operation and deal with cyclic consistency relationships, the template ensures that the model element in question has no existing mappings (*MEM*) of the type in question (lines 5-6). Furthermore, the template calls the corresponding create type algorithms (C1-C12, see Table A.3) which retrieve existing element(s) or create new element(s), and then establish the *MEM* (lines 7-17).

As for the *update* and *delete* generic templates, an analogous set of complete methods is generated for the *create* generic templates.

Out of the twelve types of generic *create* templates (C1 to C12) as shown in Table A.3,

Algorithm 7: redefinedCreateElement(p1, p2, ...)

```
1  updatedAttributes = new ArrayList<EObject>()
2  Map<EObject, List<EObject>> before = getExistingElements(secondaryEffects)
3  // call the language action to create the model element
4  create(p1, p2, ...)
5  newElement = getNewElement(languageElement, before)
6  Map<EObject, List<EObject>> after = getExistingElements(secondaryEffects)
7  // Call recursive action to ensure that constraints are maintained
8  createOtherRequiredElements(newElement, p1, p2, ...)
9  // handle secondary effects
10 handleSecondaryEffects(secondaryEffects, before, after, updatedAttributes, p1, p2,
   ...)
```

Algorithm 8: createOtherRequiredElements(element, p1, p2, ...)

```
1  for type : getMappingTypes(element) do
2      mappings = getMappings(type, element)
3      metaclass = getOtherMetaClass(type, element)
4      createType = getCreateType(type)
5      if mappings.size() != 0 then
6          break
7      switch createType do
8          case CREATE do
9              createElement(element, type, metaclass, p1, p2, ...)
10         case CREATE_AT_LEAST_ONE do
11             createAtLeastOneElement(element, type, metaclass, p1, p2, ...)
12         case CAN_CREATE_OR_USE do
13             canCreateOrUseElement(element, type, metaclass, p1, p2, ...)
14         case CREATE_OR_USE_NON_MAPPED do
15             createOrUseNonMappedElement(element, type, metaclass, p1, p2, ...)
16         case ... do
17             ...
```

we discuss four in more detail in the following sub-sections as they highlight the key issues that need to be considered for all twelve types.

A.3.1 CREATE (C2)

This *create* type automatically creates one *other* model element after creating the *primary* element, because the *primary* element requires to be mapped with the *other* element to maintain the *Compulsory Compulsory* multiplicity constraints of the concerned *LEM*. For example, creating an instance of an `InputMessage` (environment model language) based on a *Compulsory Compulsory LEM* between an *InputMessage* and an *OperationSchema* (operation model language) requires that an instance of the *InputMessage* be mapped with an instance of an *OperationSchema*. Since this kind of *LEM* is a 1-to-1 relationship, a new instance of the *OperationSchema* needs to be created, because an existing operation schema cannot be used as it must already be mapped with an input message. Thus, this *create* type ensures that the *other* element is proactively created, and then establishes a *MEM* with the *primary* element.

Algorithm 9: createElement(element, type, metaclass, p1, p2, ...)

```
1 other = CREATE(metaclass, p1, p2, ...)
2 createMapping(type, element, other)
3 createOtherRequiredElements(other, p1, p2, ...)
```

Algorithm 9 shows the execution steps for the *CREATE* type template. Line 1 creates the *other* element using a facade method as in the other generic templates, and line 2 then establishes the *MEM*. Since the creation of the new element (*other*) can violate other *LEM* constraint(s), the template recursively calls *createOtherRequiredElements* (Algorithm 8) with the *other* model element (line 3).

A.3.2 CREATE_AT_LEAST_ONE (C4)

This *create* type proactively creates at least one *other* model element (i.e., *Compulsory-Multiple* multiplicity). Type *C4* is quite similar to type *C2*, except that it creates at least one model element, while *C2* creates only one element. Consequently, line 1 of the template (Algorithm 10) asks the user to provide the number of mappings or *other* elements (at least one) which need to be mapped with the *primary* element and the template then iterates over these number of mappings (lines 2-5). For example, creating an instance of an `ActorType` (environment model language) in a *Compulsory Compulsory-Multiple LEM* between the *ActorType* (*Compulsory* mapping-end) and an *Actor* from operation model language (*Compulsory-Multiple* mapping-end) requires that at least one instance of the *Actor* is created, and each of the instances is then mapped (i.e., *MEM*) with the newly created instance of the *ActorType*.

Algorithm 10: createAtLeastOneElement(element, type, metaclass, p1, p2, ...)

```
1 int numberOfMappings = askNumberOfMappingsAtLeastOne()
2 for int count = 0; count < numberOfMappings; count++ do
3     other = CREATE(metaclass, p1, p2, ...)
4     createMapping(type, element, other)
5     createOtherRequiredElements(other, p1, p2, ...)
```

A.3.3 CAN_CREATE_OR_USE (C5)

In this *create* type, the template optionally creates one *other* model element or uses an existing element to establish a *MEM* with the primary element. The main difference between *C5* and the previously discussed *create* types is that the user is asked whether to create a *MEM* with the *primary* element (line 1 in Algorithm 11). The template proceeds only if the user agrees (lines 2-8). Furthermore, an existing model element can be used to establish a *MEM* with the *primary* element. Hence, the template queries the existing model in an attempt to retrieve a corresponding element (line 4). The matchMaker (see Section 4.3.2) is used to find such an element. If it exists, it is used to establish the *MEM*. If not, a new element is created with the facade method (lines 5-6) and subsequently used for the *MEM*.

Algorithm 11: canCreateOrUseElement(element, type, metaclass, p1, p2, ...)

```
1 boolean isCreateMapping = isCreateMapping()
2 if isCreateMapping then
3     // Check if a corresponding element exist, either mapped or not
4     other = findCorrespondingElement()
5     if other == null then
6         other = CREATE(metaclass, p1, p2, ...)
7     createMapping(type, element, other)
8     createOtherRequiredElements(other, p1, p2, ...)
```

Algorithm 12: createOrUseNonMappedElement(element, type, metaclass, p1, p2, ...)

```
1 other = findCorrespondingElement()
2 if other == null || getMappings(type, other).size() > 0 then
3     other = CREATE(metaclass, p1, p2, ...)
4 createMapping(type, element, other)
5 createOtherRequiredElements(other, p1, p2, ...)
```

A.3.4 CREATE_OR_USE_NON_MAPPED (C10)

This *create* type highlights the final issue that needs to be considered for these templates. As for type *C5*, type *C10* also attempts to find a corresponding element, and if it is not possible a new element is created and used to establish the *MEM*. However, the corresponding element cannot be mapped, i.e., the element cannot participate in a *MEM* of the type of the *LEM* in question, which is checked in line 2 of the template (Algorithm 12).

Appendix B

Definition of Perspective DSL Grammar

This appendix presents the complete definition of the perspective DSL grammar, which defines the concepts, attributes, as well as the concrete syntax of the DSL. Details of the grammar definition are shown below.

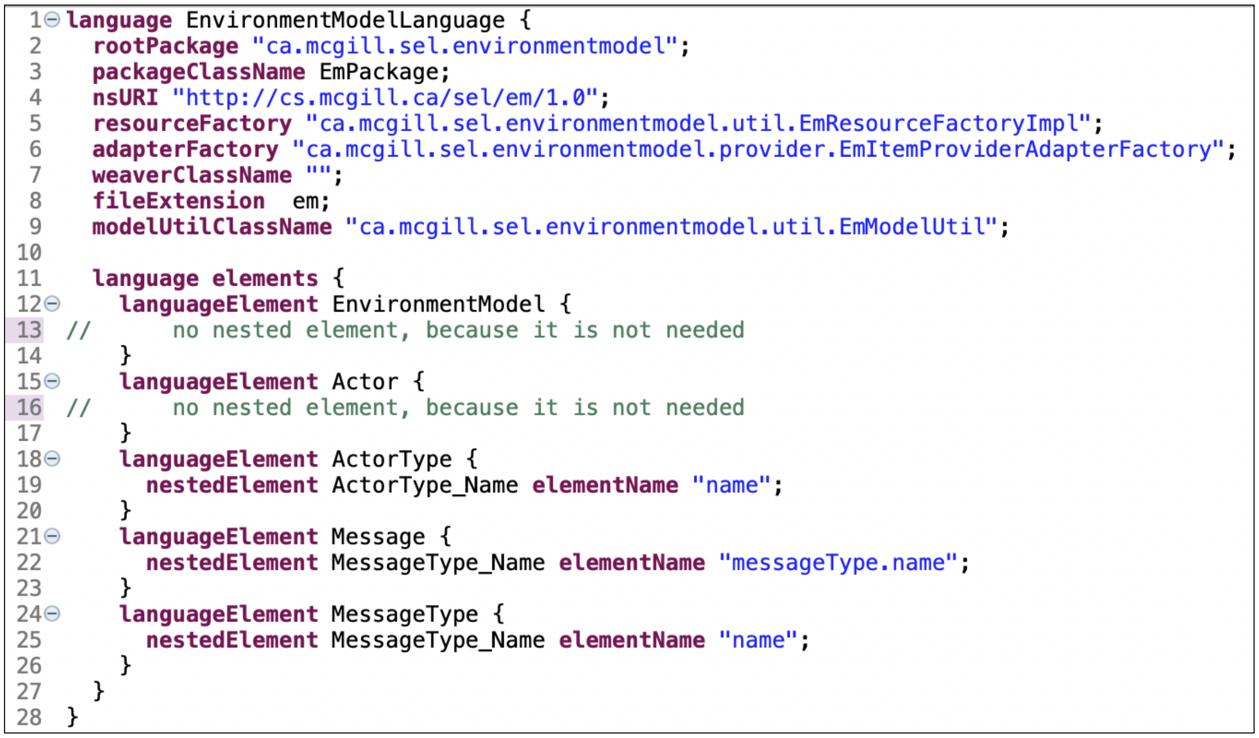
```
grammar ca.mcgill.sel.perspectivedsl.ca.mcgill.sel.PerspectiveDSL
with org.eclipse.xtext.common.Terminals
generate perspectiveDSL "http://www.mcgill.ca/sel/perspectivedsl/ca/
   mcgill/sel/PerspectiveDSL"

PerspectiveModel:
    (perspectives += Perspective)*
;
Perspective:
    'perspective' '{'
      'name'':' name = STRING';'
        # to dictate the model shown by default
        ('default'':' isDefault = [RoleName]';')?

        # language models and the perspective.
        ('savePerspective'':' savePerspective = STRING';')?
        ('saveModel'':' saveModel = STRING';')?

        # current scene, and current role name
        # when a redefined perspective action is called.
        'currentPerspective' ':' currentPerspective = STRING';'
        'currentRoleName' ':' currentRoleName = STRING';'

        ('model' 'factory' 'facade' 'calls' '{'
        (modelFacades += FacadeCall)*
      '}')?

      # define a set of role names for a given perspective
      'role' 'names' '{'
        (roleNames += RoleName)*
      '}'
      # Outlines the model cardinality for each role name
      'model' 'cardinalities' '{'
        (modelCardinalities += ModelCardinality)*
      '}'
    # Adding languages in a perspective
        'languages' '{'
            (languages += Language)*
        '}'
        ('mappings' '{'
            (mappings += LanguageElementMapping)*
        '}')?
    '}'
;
Language:
    'existing' 'language' name = ID '{'
      "roleName" roleName = [RoleName]';'
      'modelPackage' modelPackage = ID';'
        "rootPackage" rootPackage = STRING';'
        (otherPackages += OtherPackage)*

        ('actions' '{'
            (actions += PerspectiveAction)*
        '}')?

        ('intraLanguage' 'mappings' '{'
          (mappings += IntraLanguageMapping)*
        '}')?
    '}'
;
LanguageElementMapping:
    (biDirectional ?= 'bi-directional')? (uniDirectional ?= 'uni-
   directional')? 'mapping' name = ID '{'

      ('active' ':' active = BooleanType';')?
      ('default' ':' isDefault = BooleanType';')?

      (fromOrigin ?= 'origin')? (fromDestination ?= 'destination')?
   'fromMappingEnd' fromMappingEndName = ID '{'
        'modelPackage' ':' fromModelPackage = ID';'
      'isRootElement' ':' fromIsRootElement = BooleanType';'
        'cardinality' ':' fromCardinality = Cardinality';'
        'roleName' ':' fromRoleName = [RoleName]';'
        'languageElementName' ':' fromLanguageElementName = ID';'
    '}'
    (toOrigin ?= 'origin')? (toDestination ?= 'destination')?'
  toMappingEnd' toMappingEndName = ID '{'
      'modelPackage' ':' toModelPackage = ID';'
      'isRootElement' ':' toIsRootElement = BooleanType';'
        'cardinality' ':' toCardinality = Cardinality';'
        'roleName' ':' toRoleName = [RoleName]';'
        'languageElementName' ':' toLanguageElementName = ID';'
    '}'
        ('nested' 'mappings' '{'
            (nestedMappings += NestedMapping)*
        '}')?
        # constraints
    (constraints += Constraint)*
    '}'
;
NestedMapping:
    'nested' 'mapping' name = ID '{'
      'matchMaker' ':' matchMaker = BooleanType';'
        'fromElement' ':' fromElementName = STRING 'from'
  fromRoleName = [RoleName]';'
        'toElement' ':' toElementName = STRING 'from' toRoleName = [
  RoleName]';'
    '}'
;
PerspectiveAction:
   RedefinedCreateAction | RedefinedDeleteAction | HiddenAction |
   CreateMapping
;
RedefinedCreateAction:
    'redefined' 'create' 'action' name = ID '{'
      ('rootElement' ':' rootElement = BooleanType';')?
      'ownerType' ':' ownerType = ID';'
      'otherTypeAndParameters' ':' otherTypeParameters = STRING';'
      'methodCall' ':' methodCall = STRING';'
      'methodParameters' ':' methodParameter = STRING';'
      'languageElementName' ':' languageElementName = ID';'
      ('doNotGenerateMain' ':' doNotGenerate = BooleanType';')?

      # constraint condition for root model elements
    (constraints += Constraint)*
      createFacadeAction = CreateFacadeAction
      ('secondaryEffects' '{'
        ('create' 'effects' '{'
          (createEffects += CreateEffect)*
        '}')?
        ('delete' 'effects' '{'
          (deleteEffects += DeleteEffect)*
        '}')?
      '}')?
    '}'
;
RedefinedDeleteAction:
    'redefined' 'delete' 'action' name = ID '{'
      (rootElement ?= 'rootElement'';')?
      'methodCall' ':' methodCall = STRING';'
      'languageElementName' ':' languageElementName = ID';'
      ('doNotGenerateMain' ':' doNotGenerate = BooleanType';')?

      # constraints for root model elements
    (constraints += Constraint)*
      deleteFacadeAction = DeleteFacadeAction
      ('secondaryEffects' '{'
        ('create' 'effects' '{'
          (createEffects += CreateEffect)*
        '}')?
        ('delete' 'effects' '{'
          (deleteEffects += DeleteEffect)*
        '}')?
      '}')?
    '}'
;
HiddenAction:
  'hidden' 'action' name = ID';'
;
CreateMapping:
  'create' 'mapping' 'action' name = ID';'
;
CreateFacadeAction:
    'facadeAction' 'create' name = ID '{'
      'facade' 'calls' '{'
        (facadeCalls += FacadeCall)*
      '}'
    '}'
;
DeleteFacadeAction:
'facadeAction' 'create' name = ID '{'
    'facade' 'calls' '{'
      (facadeCalls += FacadeCall)+
    '}'
  '}'
;
OtherPackage:
  "otherPackage" otherPackage = STRING';'
;
FacadeCall:
  'modelPackage' ':' modelPackage = ID';'
  'languageElementName' ':' languageElementName = ID';'
  # constraints
  (constraints += Constraint)*
  (mappings += ParameterMapping)*
  'methodCall' ':' methodCall = STRING';'
;
ParameterMapping:
  'derivedParameter' mapping = STRING';'
;
```

```
CreateEffect:
  'languageElementName' ':' languageElementName = ID';'
  (mappings += ParameterMapping)*
  'methodCall' ':' methodCall = STRING';'
;
DeleteEffect:
  # The deleted element or elements
  ('deletedElement' ':' deletedElement = STRING';')?
  ('deletedElements' ':' deletedElements = STRING';')?
  # The object representation of the deleted element
  'languageElementName' ':' languageElementName = ID';'
  (mappings += ParameterMapping)*
  'methodCall' ':' methodCall = STRING';'
;
IntraLanguageMapping:
  'mapping' name = ID '{'
    'active' ':' active = BooleanType';'
    'closure' ':' closure = BooleanType';'
    'reuse' ':' reuse = BooleanType';'
    'increaseDepth' ':' increaseDepth = BooleanType';'
    'changeModel' ':' changeModel = BooleanType';'
    'from' ':' from = STRING';'
    'hops' '{'
      (hops += Hop)*
    '}'
  '}'
;
Hop:
  'hop' ':' hop = STRING';'
;
Constraint:
  'constraint' 'condition' '{'
    'attributeName' ':' attributeName = ID';'
    'value' ':' value = STRING';'
  '}'
;
ModelCardinality:
  'roleName' roleName = [RoleName]';'
  'numberOfModel' numberOfModel = Cardinality';'
;
RoleName:
  'roleName' ':' name = ID';'
;
enum Cardinality:
    COMPULSORY = '1' | OPTIONAL='0..1' | COMPULSORY_MULTIPLE='1..*'
   | OPTIONAL_MULTIPLE='0..*'
;
enum BooleanType:
    FALSE = 'false' | TRUE = 'true'
;
```

Appendix C

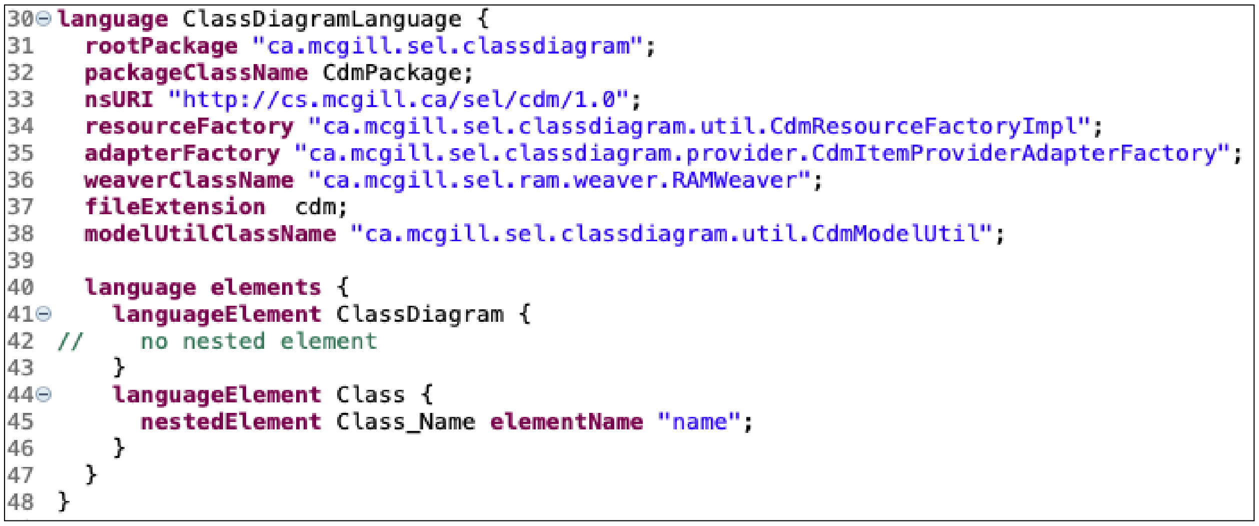
Fondue Requirement Perspective Language Registration Models

This appendix presents the complete specification for the language registrations used in the Fondue Requirement Perspective, which is shown in the following pages.



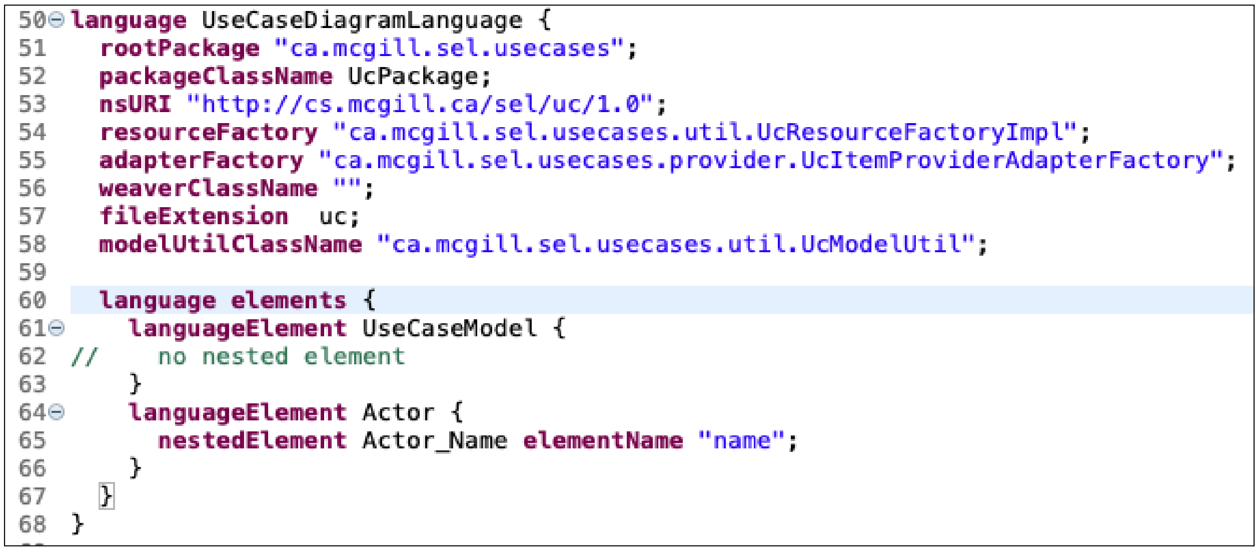
```
language EnvironmentModelLanguage {
  rootPackage "ca.mcgill.sel.environmentmodel";
  packageClassName EmPackage;
  nsURI "http://cs.mcgill.ca/sel/em/1.0";
  resourceFactory "ca.mcgill.sel.environmentmodel.util.EmResourceFactoryImpl";
  adapterFactory "ca.mcgill.sel.environmentmodel.provider.EmItemProviderAdapterFactory";
  weaverClassName "";
  fileExtension  em;
  modelUtilClassName "ca.mcgill.sel.environmentmodel.util.EmModelUtil";

  language elements {
    languageElement EnvironmentModel {
//      no nested element, because it is not needed
    }
    languageElement Actor {
//      no nested element, because it is not needed
    }
    languageElement ActorType {
      nestedElement ActorType_Name elementName "name";
    }
    languageElement Message {
      nestedElement MessageType_Name elementName "messageType.name";
    }
    languageElement MessageType {
      nestedElement MessageType_Name elementName "name";
    }
  }
}
```

Figure C.1: Environment Model Language DSL Model



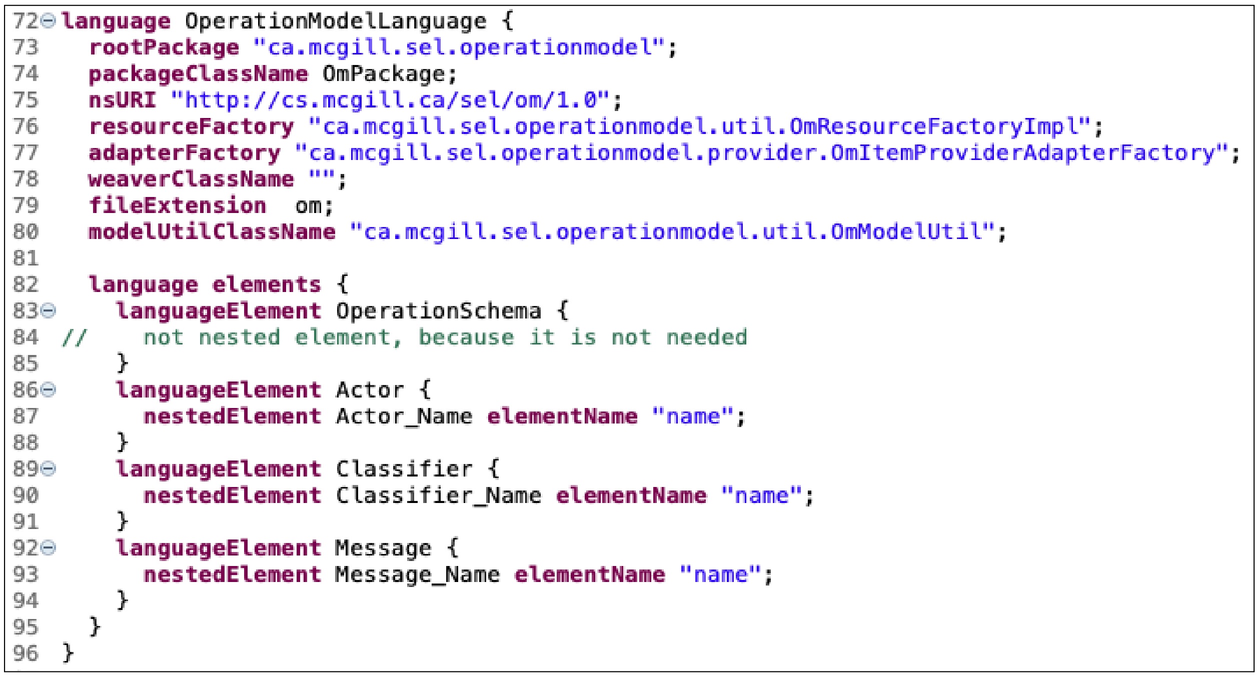
```
language ClassDiagramLanguage {
  rootPackage "ca.mcgill.sel.classdiagram";
  packageClassName CdmPackage;
  nsURI "http://cs.mcgill.ca/sel/cdm/1.0";
  resourceFactory "ca.mcgill.sel.classdiagram.util.CdmResourceFactoryImpl";
  adapterFactory "ca.mcgill.sel.classdiagram.provider.CdmItemProviderAdapterFactory";
  weaverClassName "ca.mcgill.sel.ram.weaver.RAMWeaver";
  fileExtension  cdm;
  modelUtilClassName "ca.mcgill.sel.classdiagram.util.CdmModelUtil";

  language elements {
    languageElement ClassDiagram {
//    no nested element
    }
    languageElement Class {
      nestedElement Class_Name elementName "name";
    }
  }
}
```

Figure C.2: Class Diagram Language DSL Model



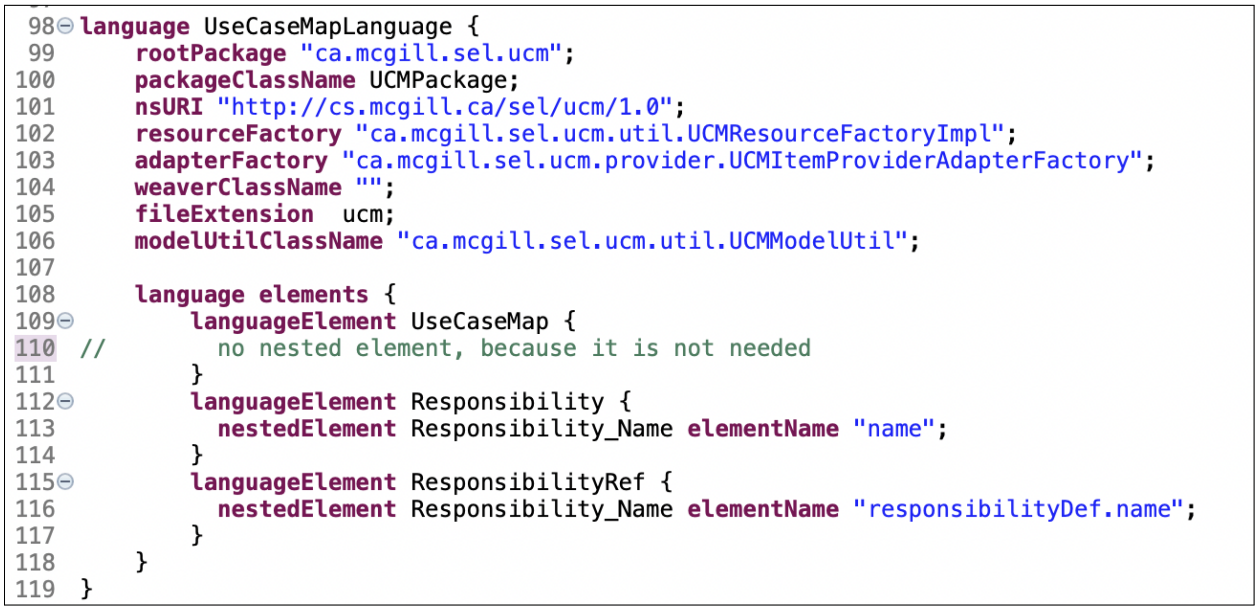
```
language UseCaseDiagramLanguage {
  rootPackage "ca.mcgill.sel.usecases";
  packageClassName UcPackage;
  nsURI "http://cs.mcgill.ca/sel/uc/1.0";
  resourceFactory "ca.mcgill.sel.usecases.util.UcResourceFactoryImpl";
  adapterFactory "ca.mcgill.sel.usecases.provider.UcItemProviderAdapterFactory";
  weaverClassName "";
  fileExtension  uc;
  modelUtilClassName "ca.mcgill.sel.usecases.util.UcModelUtil";

  language elements {
    languageElement UseCaseModel {
//     no nested element
    }
    languageElement Actor {
      nestedElement Actor_Name elementName "name";
    }
  }
}
```

Figure C.3: Use Case Diagram Language DSL Model



```
language OperationModelLanguage {
  rootPackage "ca.mcgill.sel.operationmodel";
  packageClassName OmPackage;
  nsURI "http://cs.mcgill.ca/sel/om/1.0";
  resourceFactory "ca.mcgill.sel.operationmodel.util.OmResourceFactoryImpl";
  adapterFactory "ca.mcgill.sel.operationmodel.provider.OmItemProviderAdapterFactory";
  weaverClassName "";
  fileExtension  om;
  modelUtilClassName "ca.mcgill.sel.operationmodel.util.OmModelUtil";

  language elements {
    languageElement OperationSchema {
//     not nested element, because it is not needed
    }
    languageElement Actor {
      nestedElement Actor_Name elementName "name";
    }
    languageElement Classifier {
      nestedElement Classifier_Name elementName "name";
    }
    languageElement Message {
      nestedElement Message_Name elementName "name";
    }
  }
}
```

Figure C.4: Operation Model Language DSL Model



```
language UseCaseMapLanguage {
    rootPackage "ca.mcgill.sel.ucm";
    packageClassName UCMPackage;
    nsURI "http://cs.mcgill.ca/sel/ucm/1.0";
    resourceFactory "ca.mcgill.sel.ucm.util.UCMResourceFactoryImpl";
    adapterFactory "ca.mcgill.sel.ucm.provider.UCMItemProviderAdapterFactory";
    weaverClassName "";
    fileExtension  ucm;
    modelUtilClassName "ca.mcgill.sel.ucm.util.UCMModelUtil";

    language elements {
        languageElement UseCaseMap {
//          no nested element, because it is not needed
        }
        languageElement Responsibility {
          nestedElement Responsibility_Name elementName "name";
        }
        languageElement ResponsibilityRef {
          nestedElement Responsibility_Name elementName "responsibilityDef.name";
        }
    }
}
```

Figure C.5: Use Case Map Language DSL Model

Appendix D

Complete Specification of the Fondue Requirement Perspective

This appendix presents the complete specification of the Fondue Requirement Perspective, which is detailed below.

```
perspective {

  name : "Fondue Requirement";
    default : Domain_Model;

    savePerspective: "BasePerspectiveController.saveModel(scene)";
    saveModel : "BasicActionsUtils.saveModel(EcoreUtil.getRootContainer(
   newElement), null)";

    currentPerspective : "NavigationBar.getInstance().
   getCurrentPerspective()";
  currentRoleName : "NavigationBar.getInstance().getCurrentLanguageRole()"
   ;

  model factory facade calls {
#      creating an operation schema (root model) requires to create a
   corresponding
#    message type in environment model language
    modelPackage : EmPackage;
    languageElementName : MessageType;
      methodCall : "
   FondueRequirementRedefinedOperationSchemaLanguageAction.
   createOtherElementsForOperationSchema(perspective, mappingType, scene,
   currentRoleName, currentModel, null, name)";
```

```

#   Also, creating an operation schema (root model) requires to create a
    corresponding responsibilities in use case map language
    modelPackage : UCMPackage;
    languageElementName : Responsibility;
      methodCall : "
    FondueRequirementRedefinedOperationSchemaLanguageAction.
    createOtherElementsForOperationSchema(perspective, mappingType, scene,
    currentRoleName, currentModel, null, name)";
    }

    role names {
      roleName : Domain_Model;
      roleName : UseCase_Model;
      roleName : Communication_Model;
      roleName : Operation_Model;
      roleName : Scenario_Model;
    }

     model cardinalities {
      roleName Domain_Model;
      numberOfModel 1;

      roleName UseCase_Model;
      numberOfModel 1;

      roleName Communication_Model;
      numberOfModel 1;

      roleName Operation_Model;
      numberOfModel 0..*;

      roleName Scenario_Model;
      numberOfModel 0..*;
    }

    languages {
      existing language ClassDiagramLanguage {
        roleName Domain_Model;
      modelPackage CdmPackage;
            rootPackage "ca.mcgill.sel.classdiagram";
            otherPackage "ca.mcgill.sel.classdiagram.language.controller.*
    ";
            otherPackage "ca.mcgill.sel.classdiagram.language.controller.
    impl.*";
            otherPackage "ca.mcgill.sel.usecases.*";
            otherPackage "ca.mcgill.sel.environmentmodel.*";
            otherPackage "ca.mcgill.sel.ucm.*";
            otherPackage "ca.mcgill.sel.operationmodel.*";
            otherPackage "ca.mcgill.sel.classdiagram.Classifier";
            otherPackage "ca.mcgill.sel.classdiagram.language.controller.
    impl.ControllerFactory";
            otherPackage "ca.mcgill.sel.ram.ui.components.navigationbar.
    NavigationBar";
```

```
          otherPackage "ca.mcgill.sel.perspective.fonduerequirement.
    UseCaseDiagramLanguageFacadeAction";

          actions {
            hidden action createOperation;
            hidden action createImplementationClass;
            hidden action editAssociationDirection;
            hidden action editVisibility;

      # redefined create class action, because creating a class may
    require to create other elements.
      redefined create action createNewClass {
        ownerType : ClassDiagram;
          otherTypeAndParameters : "String name, boolean dataType,
    boolean isInterface, float x, float y";
          methodCall :  "ControllerFactory.INSTANCE.
    getClassDiagramController().createNewClass((ClassDiagram) owner, name,
    dataType, isInterface, x, y)";
              # owner is not included here, because its value changes
    during the effects
              # propagation of the language action.
              # owner is dynamically handled in the generated recursive
    method
          methodParameters : "name, dataType, isInterface, x, y";
          languageElementName : Class;

          facadeAction create createOtherElementsForClass {
            facade calls {
              #   creating an actor type in the environment model
    because a class was created.
            modelPackage: EmPackage;
              languageElementName : ActorType;
              # Get the owner of the actor type to be created, since it
    is required in the createNewActor() method parameters.
              derivedParameter "EObject otherOwner = getOwner(
    perspective, scene, owner, otherRoleName)";
            derivedParameter "EnvironmentModel actorTypeOwner = (
    EnvironmentModel) otherOwner";
              methodCall : "EnvironmentModelLanguageFacadeAction.
    createActorType(perspective, scene, otherRoleName, actorTypeOwner, name
    )";

              # creating an actor in the use case diagram because a
    class was created.
              modelPackage: UcPackage;
              languageElementName : Actor;
              derivedParameter "EObject otherOwner = getOwner(
    perspective, scene, owner, otherRoleName)";
            derivedParameter "UseCaseModel actorOwner = (UseCaseModel)
    otherOwner";
              methodCall : "UseCaseDiagramLanguageFacadeAction.
    createNewActor(perspective, scene, otherRoleName, actorOwner, name, x,
    y)";

```

```
       # creating an actor in the operation model due creating an actor type.
       # Note that the actor type was created because of creating a  class,
       # see above facade call
        modelPackage: OmPackage;
        languageElementName : Actor;
        derivedParameter "EObject otherOwner = getOwner(perspective, scene, owner, otherRoleName)";
       derivedParameter "UseCaseMap actorOwner = (UseCaseMap)otherOwner";
        methodCall : "UseCaseMapLanguageFacadeAction.createActor(perspective, scene, otherRoleName, actorOwner, name)";

       # creating a classifier in the operation model because a class was created.
        modelPackage: OmPackage;
        languageElementName : Classifier;
        derivedParameter "EObject otherOwner = getOwner(perspective, scene, owner, otherRoleName)";
       derivedParameter "UseCaseMap classifierOwner = (UseCaseMap)otherOwner";
        methodCall : "UseCaseMapLanguageFacadeAction.createClass(perspective, scene, otherRoleName, classifierOwner, name)";
       }
      }
    }

    redefined delete action removeClassifier {
      methodCall :  "ControllerFactory.INSTANCE.getClassDiagramController()
      .removeClassifier((Classifier) currentElement)";
      languageElementName : Classifier;
      facadeAction delete deleteModelElement {
       facade calls {
       modelPackage: UcPackage;
        languageElementName : Actor;
        methodCall : "UseCaseDiagramLanguageFacadeAction.deleteActor(perspective, scene, otherRoleName, otherElement)";

       modelPackage : EmPackage;
        languageElementName : ActorType;
        methodCall : "EnvironmentModelLanguageFacadeAction.deleteActorType(perspective, scene, otherRoleName, otherElement)";

       modelPackage : OmPackage;
        languageElementName : Actor;
        methodCall : "OperationSchemaLanguageFacadeAction.deleteActor(perspective, scene, otherRoleName, otherElement)";

       modelPackage : OmPackage;
        languageElementName : Classifier;
        methodCall : "OperationSchemaLanguageFacadeAction.deleteClass(perspective, scene, otherRoleName, otherElement)";
```

```
            }
          }
        }
          }
      }
      existing language UseCaseDiagramLanguage {
        roleName UseCase_Model;
        modelPackage UcPackage;
          rootPackage "ca.mcgill.sel.usecases";
          otherPackage "ca.mcgill.sel.usecases.language.controller.impl
.*";
          otherPackage "ca.mcgill.sel.classdiagram.*";
          otherPackage "ca.mcgill.sel.ucm.*";
          otherPackage "ca.mcgill.sel.operationmodel.*";
          otherPackage "ca.mcgill.sel.environmentmodel.*";
          otherPackage "ca.mcgill.sel.usecases.Actor";
          otherPackage "ca.mcgill.sel.ram.ui.components.navigationbar.
NavigationBar";
          actions {
            redefined create action createNewActor {
              ownerType : UseCaseModel;
          otherTypeAndParameters : "String name, float x, float y";
          methodCall :  "UseCaseControllerFactory.INSTANCE.
getUseCaseDiagramController().createNewActor((UseCaseModel) owner, name
, x, y)";
          methodParameters : "name, x, y";
          languageElementName : Actor;

          facadeAction create createOtherElementsForActor {
            facade calls {
              modelPackage : CdmPackage;
              languageElementName : Class;
              derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
            derivedParameter "ClassDiagram otherOwner = (ClassDiagram) o
";
            derivedParameter "boolean dataType = false";
            derivedParameter "boolean isInterface = false";
              methodCall : "ClassDiagramLanguageFacadeAction.
createNewClass(perspective, scene, otherRoleName, otherOwner, name,
dataType, isInterface, x, y)";

            modelPackage : EmPackage;
              languageElementName : ActorType;
              derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
            derivedParameter "EnvironmentModel otherOwner = (
EnvironmentModel) o";
              methodCall : "EnvironmentModelLanguageFacadeAction.
createActorType(perspective, scene, otherRoleName, otherOwner, name)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
            derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
              methodCall : "OperationSchemaLanguageFacadeAction.createClass(perspective, scene, otherRoleName, otherOwner, name)";

            modelPackage : OmPackage;
              languageElementName : Actor;
              derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
            derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
              methodCall : "OperationSchemaLanguageFacadeAction.createActor(perspective, scene, otherRoleName, otherOwner, name)";
            }
          }
      }
      redefined delete action deleteActor {
          methodCall :  "UseCaseControllerFactory.INSTANCE.getUseCaseDiagramController().removeActor((Actor) currentElement)";
          languageElementName : Actor;

          facadeAction delete deleteOtherElements {
            facade calls {
            modelPackage: CdmPackage;
              languageElementName : Class;
              methodCall : "ClassDiagramLanguageFacadeAction.removeClassifier(perspective, scene, otherRoleName, otherElement)";

            modelPackage : EmPackage;
              languageElementName : ActorType;
              methodCall : "EnvironmentModelLanguageFacadeAction.deleteActorType(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Actor;
              methodCall : "OperationSchemaLanguageFacadeAction.deleteActor(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              methodCall : "OperationSchemaLanguageFacadeAction.deleteClass(perspective, scene, otherRoleName, otherElement)";
            }
          }
        }
          }
      }
      existing language UseCaseMapLanguage {
        roleName Scenario_Model;
        modelPackage UCMPackage;
          rootPackage "ca.mcgill.sel.ucm";
```

```
         otherPackage "ca.mcgill.sel.ucm.language.controller";
         otherPackage "ca.mcgill.sel.environmentmodel";
         otherPackage "ca.mcgill.sel.operationmodel";
         otherPackage "ca.mcgill.sel.classdiagram";
         otherPackage "ca.mcgill.sel.usecases";
         otherPackage "ca.mcgill.sel.ram.ui.perspective.controller";
         otherPackage "ca.mcgill.sel.ucm.language.controller.
ControllerFactory";

         actions {
           redefined create action createResponsibilityRef {
             ownerType : UseCaseMap;
             otherTypeAndParameters : "float x, float y, String name";
         methodCall :  "ca.mcgill.sel.ucm.language.controller.
ControllerFactory.INSTANCE.getResponsibilityController().
createResponsibilityRef((UseCaseMap) owner, x, y, name)";
         methodParameters : "x, y, name";
         languageElementName : ResponsibilityRef;

         facadeAction create createOtherElementsForResponsibilityRef {
           facade calls {
             modelPackage : EmPackage;
             languageElementName : Message;
             derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
           derivedParameter "EnvironmentModel em = (EnvironmentModel) o
";
           derivedParameter "EObject otherOwner = getOwner(perspective,
 scene, owner, otherRoleName)";
           methodCall : "EnvironmentModelLanguageFacadeAction.
createMessage(perspective, scene, otherRoleName, otherOwner, em, name,
MessageDirection.INPUT)";
           }
         }
     }
     redefined create action createResponsibility {
       ownerType : UseCaseMap;
         otherTypeAndParameters : "String name";
         methodCall :  "ca.mcgill.sel.ucm.language.controller.
ControllerFactory.INSTANCE.getResponsibilityController().
createResponsibility((UseCaseMap) owner, name)";
         methodParameters : "name";
         languageElementName : Responsibility;

         facadeAction create createOtherElementsForResponsibility {
           facade calls {
             modelPackage : OmPackage;
             languageElementName : OperationSchema;
           methodCall : "ModelFactory.INSTANCE.createNewModel(
perspective, scene, otherRoleName, name)";

           modelPackage : OmPackage;
           languageElementName : MessageType;
```

```
              derivedParameter "EObject otherOwner = getOwner(
perspective, scene, owner, otherRoleName)";
              methodCall : "EnvironmentModelLanguageFacadeAction.
createMessageType(perspective, scene, otherRoleName, otherOwner, name)"
;
            }
          }
      }

      redefined delete action deleteResponsibility {
        methodCall :  "ControllerFactory.INSTANCE.
getResponsibilityController().removeResponsibility((Responsibility)
currentElement)";
          languageElementName : Responsibility;

          facadeAction delete deleteOtherElements {
            facade calls {
            modelPackage : EmPackage;
              languageElementName : Message;
              methodCall : "EnvironmentModelLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

            modelPackage : EmPackage;
              languageElementName : MessageType;
              methodCall : "EnvironmentModelLanguageFacadeAction.
deleteMessageType(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Message;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteClass(perspective, scene, otherRoleName, otherElement)";
            }
          }
        }
        redefined delete action deleteResponsibilityRef {
          methodCall :  "ca.mcgill.sel.ucm.language.controller.
ControllerFactory.INSTANCE.getResponsibilityController().
removeResponsibilityRef((ResponsibilityRef) currentElement)";
          languageElementName : ResponsibilityRef;

          facadeAction delete deleteOtherElements {
            facade calls {

            modelPackage : EmPackage;
              languageElementName : Message;
              methodCall : "EnvironmentModelLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

```

```
           modelPackage : EmPackage;
             languageElementName : MessageType;
             methodCall : "EnvironmentModelLanguageFacadeAction.
deleteMessageType(perspective, scene, otherRoleName, otherElement)";

           modelPackage : OmPackage;
             languageElementName : Actor;
             methodCall : "OperationSchemaLanguageFacadeAction.
deleteActor(perspective, scene, otherRoleName, otherElement)";

           modelPackage : OmPackage;
             languageElementName : Message;
             methodCall : "OperationSchemaLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

           modelPackage : OmPackage;
             languageElementName : Classifier;
             methodCall : "OperationSchemaLanguageFacadeAction.
deleteClass(perspective, scene, otherRoleName, otherElement)";

           }
         }
       }
     }
   }

     existing language EnvironmentModelLanguage {
       roleName Communication_Model;
       modelPackage EmPackage;
         rootPackage "ca.mcgill.sel.environmentmodel";
         otherPackage "ca.mcgill.sel.environmentmodel.language.
controller.impl.*";
         otherPackage "ca.mcgill.sel.classdiagram.*";
         otherPackage "ca.mcgill.sel.ucm.*";
         otherPackage "ca.mcgill.sel.operationmodel.*";
         otherPackage "ca.mcgill.sel.usecases.*";
         otherPackage "ca.mcgill.sel.ram.ui.perspective.controller.*";
         otherPackage "ca.mcgill.sel.environmentmodel.Actor";
         otherPackage "ca.mcgill.sel.environmentmodel.Message";
         otherPackage "ca.mcgill.sel.ram.ui.components.navigationbar.
NavigationBar";

         actions {

           redefined create action createActorType {
             ownerType : EnvironmentModel;
         otherTypeAndParameters : "String name";
         methodCall :  "EnvironmentModelControllerFactory.INSTANCE.
getEnvironmentModelController().createActorType((EnvironmentModel)
owner, name)";
         methodParameters : "name";
         languageElementName : ActorType;

```

```
            facadeAction create createOtherElementsForActorType {
              facade calls {
                modelPackage : CdmPackage;
                languageElementName : Class;
                derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
              derivedParameter "ClassDiagram otherOwner = (ClassDiagram) o";
              derivedParameter "boolean dataType = false";
              derivedParameter "boolean isInterface = false";
              derivedParameter "float x = 0";
              derivedParameter "float y = 0";
                methodCall : "ClassDiagramLanguageFacadeAction.createNewClass(perspective, scene, otherRoleName, otherOwner, name, dataType, isInterface, x, y)";

              modelPackage : OmPackage;
                languageElementName : Actor;
                derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
              derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
              methodCall : "OperationSchemaLanguageFacadeAction.createActor(perspective, scene, otherRoleName, otherOwner, name)";

              modelPackage : UcPackage;
                languageElementName : Actor;
                derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
              derivedParameter "UseCaseModel otherOwner = (UseCaseModel) o";
              methodCall : "UseCaseDiagramLanguageFacadeAction.createNewActor(perspective, scene, otherRoleName, otherOwner, name, 0, 0)";

              modelPackage : OmPackage;
                languageElementName : Classifier;
                derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
              derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
              methodCall : "OperationSchemaLanguageFacadeAction.createClass(perspective, scene, otherRoleName, otherOwner, name)";
              }
            }
        }

        redefined create action createActor {
          ownerType : EnvironmentModel;
            otherTypeAndParameters : "String actorTypeName, String name, float x, float y";
            methodCall :  "EnvironmentModelControllerFactory.INSTANCE.getEnvironmentModelController().createActor((EnvironmentModel) owner,
  actorTypeName, name, x, y)";
            methodParameters : "actorTypeName, name, x, y";
            languageElementName : Actor;

            facadeAction create createOtherElementsForActorType {
              facade calls {

              }
            }
            secondaryEffects {
              create effects {
                languageElementName : ActorType;
                methodCall : "
  FondueRequirementRedefinedEnvironmentModelLanguageAction.
  createOtherElementsForActorType(perspective, scene, currentRole,
  newElement, newElement.eContainer(), name)";
              }
            }
        }
        redefined create action createMessage {
          ownerType : Actor;
            otherTypeAndParameters : "EnvironmentModel em, String name,
  MessageDirection messageDirection";
            methodCall :  "EnvironmentModelControllerFactory.INSTANCE.
  getEnvironmentModelController().createMessage((Actor) owner, em, name,
  messageDirection)";
            methodParameters : "em, name, messageDirection";
            languageElementName : Message;

            facadeAction create createOtherElementsForMessage {
              facade calls {
                modelPackage : UCMPackage;
                languageElementName : ResponsibilityRef;
                derivedParameter "EObject otherOwner =
  PerspectiveControllerFactory.INSTANCE.getBasePerspectiveController().
  getRootElement(scene, UCMPackage.eINSTANCE.getUseCaseMap())";
              derivedParameter "float x = 0";
              derivedParameter "float y = 0";
                methodCall : "UseCaseMapLanguageFacadeAction.
  createResponsibilityRef(perspective, scene, otherRoleName, otherOwner,
  x, y, name)";
              }
            }
            secondaryEffects {
              create effects {
                languageElementName : MessageType;
                methodCall : "
  FondueRequirementRedefinedEnvironmentModelLanguageAction.
  createOtherElementsForMessageType(perspective, scene, currentRole,
  newElement, newElement.eContainer(), name, messageDirection)";
              }
            }
        }
```

```
    redefined create action createMessageType {
      ownerType : EnvironmentModel;
        otherTypeAndParameters : "String name, MessageDirection messageDirection";
        methodCall :  "EnvironmentModelControllerFactory.INSTANCE.getEnvironmentModelController().createMessageType((EnvironmentModel) owner, name)";
        methodParameters : "name, messageDirection";
        languageElementName : MessageType;

        facadeAction create createOtherElementsForMessageType {
          facade calls {
            modelPackage : OmPackage;
            languageElementName : Message;
            derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
          derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
            methodCall : "OperationSchemaLanguageFacadeAction.createMessage(perspective, scene, otherRoleName, otherOwner, null, name, true)";

          modelPackage : OmPackage;
          languageElementName : OperationSchema;
            methodCall : "ModelFactory.INSTANCE.createNewModel(perspective, scene, otherRoleName, name, false)";
        }
        }
    }

    redefined delete action deleteActorType {
        methodCall :  "EnvironmentModelControllerFactory.INSTANCE.getEnvironmentModelController().removeActorType((ActorType) currentElement)";
        languageElementName : ActorType;

        facadeAction delete deleteOtherElements {
          facade calls {
          modelPackage: CdmPackage;
          languageElementName : Class;
            methodCall : "ClassDiagramLanguageFacadeAction.removeClassifier(perspective, scene, otherRoleName, otherElement)";

          modelPackage: UcPackage;
            languageElementName : Actor;
            methodCall : "UseCaseDiagramLanguageFacadeAction.deleteActor(perspective, scene, otherRoleName, otherElement)";

          modelPackage : OmPackage;
            languageElementName : Actor;
            methodCall : "OperationSchemaLanguageFacadeAction.deleteActor(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteClass(perspective, scene, otherRoleName, otherElement)";
            }
          }
        }
        redefined delete action deleteMessage {
          methodCall :  "EnvironmentModelControllerFactory.INSTANCE.
getActorController().removeMessage((Message) currentElement)";
          languageElementName : Message;

          facadeAction delete deleteOtherElements {
            facade calls {

            modelPackage : UCMPackage;
              languageElementName : Responsibility;
              methodCall : "UseCaseMapLanguageFacadeAction.
deleteResponsibility(perspective, scene, otherRoleName, otherElement)";

            modelPackage : UCMPackage;
              languageElementName : ResponsibilityRef;
              methodCall : "UseCaseMapLanguageFacadeAction.
deleteResponsibilityRef(perspective, scene, otherRoleName, otherElement
)";

            modelPackage : OmPackage;
              languageElementName : Message;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

            }
          }

        }

        redefined delete action deleteMessageType {
          methodCall :  "EnvironmentModelControllerFactory.INSTANCE.
getEnvironmentModelController().removeMessageType((MessageType)
currentElement)";
          languageElementName : MessageType;

          facadeAction delete deleteOtherElements {
            facade calls {

            modelPackage : OmPackage;
              languageElementName : Message;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteMessage(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteClass(perspective, scene, otherRoleName, otherElement)";
            }
          }
        }
      }
 }

      existing language OperationSchemaLanguage {
        roleName Operation_Model;
    modelPackage OmPackage;
          rootPackage "ca.mcgill.sel.operationmodel";
          otherPackage "ca.mcgill.sel.operationmodel.language.controller
.*";
          otherPackage "ca.mcgill.sel.classdiagram.*";
          otherPackage "ca.mcgill.sel.environmentmodel.*";
          otherPackage "ca.mcgill.sel.ucm.*";
          otherPackage "ca.mcgill.sel.usecases.*";
          otherPackage "ca.mcgill.sel.ram.ui.perspective.controller.*";
          otherPackage "ca.mcgill.sel.operationmodel.Actor";
          otherPackage "ca.mcgill.sel.operationmodel.Message";
          otherPackage "ca.mcgill.sel.operationmodel.Classifier";
          otherPackage "ca.mcgill.sel.ram.ui.components.navigationbar.
NavigationBar";

          actions {
            # This action is needed because of its create other elements
.
            # Model factory handles the creation of all root model
elements
            redefined create action createOperationSchema {
              ownerType : OperationSchema;
          otherTypeAndParameters : "String name";
          methodCall :  "String doNotCreate = null";
          methodParameters : "name";
          languageElementName : OperationSchema;

          facadeAction create createOtherElementsForOperationSchema {
            facade calls {

            modelPackage : UCMPackage;
              languageElementName : Responsibility;
              derivedParameter "EObject ucm =
PerspectiveControllerFactory.INSTANCE.getBasePerspectiveController().
getRootElement(scene, UCMPackage .eINSTANCE.getUseCaseMap())";
              methodCall : "UseCaseMapLanguageFacadeAction.
createResponsibility(perspective, scene, otherRoleName, ucm, name)";

            modelPackage : EmPackage;
            languageElementName : MessageType;
              derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
              derivedParameter "EnvironmentModel em = (EnvironmentModel)
 o";
```

```
             methodCall : "EnvironmentModelLanguageFacadeAction.createMessageType(perspective, scene, otherRoleName, em, name, MessageDirection.INPUT)";
           }
         }
     }
     redefined create action createActor {
       ownerType : OperationSchema;
         otherTypeAndParameters : "String name";
         methodCall :  "OperationModelControllerFactory.INSTANCE.getOperationModelController().createActor((OperationSchema) owner, name)";
         methodParameters : "name";
         languageElementName : Actor;

         facadeAction create createOtherElementsForActor {
           facade calls {

             modelPackage : EmPackage;
           languageElementName : ActorType;
             derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
           derivedParameter "EnvironmentModel em = (EnvironmentModel) o";
             methodCall : "EnvironmentModelLanguageFacadeAction.createActorType(perspective, scene, otherRoleName, em, name)";

           modelPackage : CdmPackage;
           languageElementName : Class;
             derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
           derivedParameter "ClassDiagram otherOwner = (ClassDiagram) o";
           methodCall : "ClassDiagramLanguageFacadeAction.createNewClass(perspective, scene, otherRoleName, otherOwner, name, false, false, 0, 0)";

           modelPackage : OmPackage;
           languageElementName : Classifier;
             derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
           derivedParameter "OperationSchema otherOwner = (OperationSchema) o";
             methodCall : "OperationSchemaLanguageFacadeAction.createClass(perspective, scene, otherRoleName, otherOwner, name)";

           modelPackage : UcPackage;
           languageElementName : Actor;
             derivedParameter "EObject o = getOwner(perspective, scene, owner, otherRoleName)";
           derivedParameter "UseCaseModel otherOwner = (UseCaseModel) o";
```

```
              methodCall : "UseCaseDiagramLanguageFacadeAction.
    createNewActor(perspective, scene, otherRoleName, otherOwner, name, 0,
    0)";

            }
          }
      }
      redefined create action createMessage {
        ownerType : OperationSchema;
          otherTypeAndParameters : "ca.mcgill.sel.operationmodel.Actor
    actor, String name, boolean inputMessage";
          methodCall :  "OperationModelControllerFactory.INSTANCE.
    getOperationModelController().createMessage((OperationSchema) owner,
    actor, name, inputMessage)";
          methodParameters : "actor, name, inputMessage";
          languageElementName : Message;

          facadeAction create createOtherElementsForMessage {
            facade calls {

              modelPackage : EmPackage;
              languageElementName : MessageType;
              derivedParameter "EObject o = getOwner(perspective, scene,
     owner, otherRoleName)";
            derivedParameter "EnvironmentModel em = (EnvironmentModel) o
    ";
              methodCall : "EnvironmentModelLanguageFacadeAction.
    createMessageType(perspective, scene, otherRoleName, owner, name,
    MessageDirection.OUTPUT)";

            modelPackage : OmPackage;
            languageElementName : OperationSchema;
              methodCall : "ModelFactory.INSTANCE.createNewModel(
    perspective, scene, otherRoleName, name, false)";
            }
          }
      }

      redefined create action createClass {
        ownerType : OperationSchema;
          otherTypeAndParameters : "String name";
          methodCall :  "OperationModelControllerFactory.INSTANCE.
    getOperationModelController().createClass((OperationSchema) owner, name
    )";
          methodParameters : "name";
          languageElementName : Classifier;

          facadeAction create createOtherElementsForClassifier {
            facade calls {
              modelPackage : EmPackage;
              languageElementName : ActorType;
              derivedParameter "EObject o = getOwner(perspective, scene,
     owner, otherRoleName)";
```

```
           derivedParameter "EnvironmentModel em = (EnvironmentModel) o
";
             methodCall : "EnvironmentModelLanguageFacadeAction.
createActorType(perspective, scene, otherRoleName, owner, name)";

           modelPackage : CdmPackage;
             languageElementName : Class;
             derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
           derivedParameter "ClassDiagram otherOwner = (ClassDiagram) o
";
           methodCall : "ClassDiagramLanguageFacadeAction.
createNewClass(perspective, scene, otherRoleName, otherOwner, name,
false, false, 0, 0)";

           modelPackage : UcPackage;
             languageElementName : Actor;
             derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
           derivedParameter "UseCaseModel otherOwner = (UseCaseModel) o
";
           methodCall : "UseCaseDiagramLanguageFacadeAction.
createNewActor(
           perspective, scene, otherRoleName, otherOwner, name, 0, 0)";


           modelPackage : OmPackage;
             languageElementName : Actor;
             derivedParameter "EObject o = getOwner(perspective, scene,
 owner, otherRoleName)";
           derivedParameter "OperationSchema otherOwner = (
OperationSchema) o";
           methodCall : "OperationSchemaLanguageFacadeAction.
createActor(
           perspective, scene, otherRoleName, otherOwner, name)";

           }
         }
     }

     redefined delete action deleteActor {
         methodCall :  "OperationModelControllerFactory.INSTANCE.
getOperationModelController().removeActor((Actor) currentElement)";
         languageElementName : Actor;

         facadeAction delete deleteOtherElements {
           facade calls {
           modelPackage: CdmPackage;
           languageElementName : Class;
             methodCall : "ClassDiagramLanguageFacadeAction.
removeClassifier(perspective, scene, otherRoleName, otherElement)";

           modelPackage: UcPackage;
```

```
              languageElementName : Actor;
              methodCall : "UseCaseDiagramLanguageFacadeAction.
deleteActor(perspective, scene, otherRoleName, otherElement)";

            modelPackage : EmPackage;
              languageElementName : ActorType;
              methodCall : "EnvironmentModelLanguageFacadeAction.
deleteActorType(perspective, scene, otherRoleName, otherElement)";

            modelPackage : OmPackage;
              languageElementName : Classifier;
              methodCall : "OperationSchemaLanguageFacadeAction.
deleteClass(perspective, scene, otherRoleName, otherElement)";

            }
          }
        }
        redefined delete action deleteMessage {
          methodCall :  "OperationModelControllerFactory.INSTANCE.
getOperationModelController().removeMessage((Message) currentElement)";
          languageElementName : Message;

          facadeAction delete deleteOtherElements {
            facade calls {

            modelPackage : EmPackage;
              languageElementName : MessageType;
              methodCall : "EnvironmentModelLanguageFacadeAction.
deleteMessageType(perspective, scene, otherRoleName, otherElement)";
            }
          }
        }
        redefined delete action deleteClass {
          methodCall :  "OperationModelControllerFactory.INSTANCE.
getOperationModelController().removeClassifier((Classifier)
currentElement)";
          languageElementName : Classifier;

          facadeAction delete deleteOtherElements {
            facade calls {
            modelPackage: CdmPackage;
            languageElementName : Class;
              methodCall : "ClassDiagramLanguageFacadeAction.
removeClassifier(perspective, scene, otherRoleName, otherElement)";

            modelPackage: UcPackage;
              languageElementName : Actor;
              methodCall : "UseCaseDiagramLanguageFacadeAction.
deleteActor(perspective, scene, otherRoleName, otherElement)";

            modelPackage : EmPackage;
              languageElementName : ActorType;
```

```
                methodCall : "EnvironmentModelLanguageFacadeAction.
    deleteActorType(perspective, scene, otherRoleName, otherElement)";

              modelPackage : OmPackage;
                languageElementName : Actor;
                methodCall : "OperationSchemaLanguageFacadeAction.
    deleteActor(perspective, scene, otherRoleName, otherElement)";
              }
            }
          }
            }
        }
    }

    # Language Element Mappings
    mappings {
      # languageElement mappings for root model elements
      bi-directional mapping DM_UC {
        fromMappingEnd DomainModelCompulsory {
          modelPackage : CdmPackage;
          isRootElement : true;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : ClassDiagram;
        }

        toMappingEnd UseCaseCompulsory {
          modelPackage : UcPackage;
          isRootElement : true;
          cardinality : 1;
          roleName : UseCase_Model;
          languageElement : UseCaseModel;
        }

        }

        bi-directional mapping DM_EM {
          fromMappingEnd DomainModelCompulsory {
            modelPackage : CdmPackage;
          isRootElement : true;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : ClassDiagram;
        }
        toMappingEnd EnvironmentModelCompulsory {
          modelPackage : EmPackage;
          isRootElement : true;
              cardinality : 1;
              roleName : Communication_Model;
              languageElement : EnvironmentModel;
        }
        }
        bi-directional mapping DM_OM {
          fromMappingEnd DomainModelCompulsory {
            modelPackage : CdmPackage;
          isRootElement : true;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : ClassDiagram;
        }
          toMappingEnd OperationModelOptionalMultiple {
            modelPackage : OmPackage;
          isRootElement : true;
              cardinality : 0..*;
              roleName : Operation_Model;
              languageElement : OperationSchema;
        }
        }

        bi-directional mapping EM_UCM {
          fromMappingEnd EnvironmentModelCompulsory {
            modelPackage : EmPackage;
          isRootElement : true;
          cardinality : 1;
          roleName : Communication_Model;
          languageElementName : EnvironmentModel;
        }
          toMappingEnd UseCaseMapMultiple {
            modelPackage : UCMPackage;
          isRootElement : true;
              cardinality : 0..*;
              roleName : Scenario_Model;
              languageElement : UseCaseMap;
        }
        }

        bi-directional mapping UCM_OM {
          fromMappingEnd UseCaseMapMultiple {
            modelPackage : UCMPackage;
          isRootElement : true;
              cardinality : 0..*;
              roleName : Scenario_Model;
              languageElementName : UseCaseMap;
        }
        toMappingEnd OperationModelOptionalMultiple {
            modelPackage : OmPackage;
          isRootElement : true;
              cardinality : 0..*;
              roleName : Operation_Model;
              languageElement : OperationSchema;
        }
        }

        # Other language element mappings

```

```
    # R1
        bi-directional mapping DMClass_EMActorType {
          fromMappingEnd CdmClassCompulsory {
            modelPackage : CdmPackage;
          isRootElement : false;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : Class;
          }
          toMappingEnd EMActorTypeOptional {
            modelPackage : EmPackage;
            isRootElement : false;
              cardinality : 0..1;
              roleName : Communication_Model;
              languageElement : ActorType;
          }

            nested mappings {
              nested mapping ClassName_ActorName {
                matchMaker : true;
                fromElement : "name" from Domain_Model;
                    toElement : "name" from Communication_Model;
              }

            }
        }

    # R2
        bi-directional mapping EMMessageType_OMOperationSchema {

      fromMappingEnd EMMessageTypeCompulsory {
        modelPackage : EmPackage;
            isRootElement : false;
            cardinality : 1;
            roleName : Communication_Model;
            languageElementName : MessageType;
          }
          toMappingEnd OperationModelCompulsory {
            modelPackage : OmPackage;
          isRootElement : true;
              cardinality : 1;
              roleName : Operation_Model;
              languageElement : OperationSchema;
        }

            nested mappings {
              nested mapping MessageTypeName_OperationSchemaName {
                matchMaker : true;
                fromElement : "name" from Communication_Model;
                    toElement : "name" from Operation_Model;
              }
            }
            constraint condition {
```

```
          attributeName: messageDirection;
          value: "MessageDirection.INPUT";
        }
        }

    # R3
        bi-directional mapping EMMessageType_OMMessage {

          fromMappingEnd EMMessageTypeOptional {
            modelPackage : EmPackage;
            isRootElement : false;
            cardinality : 0..1;
            roleName : Communication_Model;
            languageElementName : MessageType;
          }
          toMappingEnd OMMessageCompulsoryMultiple {
            modelPackage : OmPackage;
            isRootElement : false;
              cardinality : 1..*;
              roleName : Operation_Model;
              languageElement : Message;
          }

            nested mappings {
              nested mapping MessageTypeName_MessageName {
                matchMaker : true;
                fromElement : "name" from Communication_Model;
                    toElement : "name" from Operation_Model;
              }
            }
            constraint condition {
          attributeName: messageDirection;
          value: "MessageDirection.OUTPUT";
        }
        }

    # R4
        bi-directional mapping EMActorType_OMActor {
          fromMappingEnd EMActorTypeCompulsory {
            modelPackage : EmPackage;
            isRootElement : false;
              cardinality : 1;
              roleName : Communication_Model;
              languageElementName : ActorType;
          }
          toMappingEnd OMActorMultipleCompulsory {
            modelPackage : OmPackage;
            isRootElement : false;
              cardinality : 1..*;
              roleName : Operation_Model;
              languageElement : Actor;
          }
            nested mappings {
              nested mapping ActorTypeName_ActorName {
                matchMaker : true;
                fromElement : "name" from Communication_Model;
                    toElement : "name" from Operation_Model;
              }
            }
        }

    # R5
    bi-directional mapping DMClass_OMClassifier {
          fromMappingEnd CdmClassCompulsory {
            modelPackage : CdmPackage;
          isRootElement : false;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : Class;
          }
          toMappingEnd OMClassifierMultipleOptional {
            modelPackage : OmPackage;
            isRootElement : false;
              cardinality : 0..*;
              roleName : Operation_Model;
              languageElement : Classifier;
          }

            nested mappings {
              nested mapping ClassName_ClassifierName {
                matchMaker : true;
                fromElement : "name" from Domain_Model;
                    toElement : "name" from Operation_Model;
              }
            }
        }

      # R6
        bi-directional mapping UCMResponsibility_OMOperationSchema {
          fromMappingEnd UCMResponsibilityCompulsoryMultiple {
            modelPackage : UCMPackage;
            isRootElement : false;
              cardinality : 1..*;
              roleName : Scenario_Model;
              languageElementName : Responsibility;
          }
          toMappingEnd OperationModelCompulsory {
            modelPackage : OmPackage;
          isRootElement : true;
              cardinality : 0..1;
              roleName : Operation_Model;
              languageElement : OperationSchema;
        }

            nested mappings {
              nested mapping ResponsibilityName_OperationSchemaName {
                matchMaker : true;
                fromElement : "name" from Scenario_Model;
                    toElement : "name" from Operation_Model;
              }
            }
        }

        # R7
        bi-directional mapping EMessage_UCMResponsibilityRef {
          fromMappingEnd EMMessageCompulsory {
            modelPackage : EmPackage;
            isRootElement : false;
              cardinality : 1;
              roleName : Communication_Model;
              languageElementName : Message;
          }
          toMappingEnd UCMResponsibilityRefCompulsoryMultiple {
            modelPackage : UCMPackage;
            isRootElement : false;
              cardinality : 1..*;
              roleName : Scenario_Model;
              languageElement : ResponsibilityRef;
          }
            nested mappings {
              nested mapping ActorTypeName_ActorName {
                matchMaker : true;
                fromElement : "messageType.name" from Communication_Model;
                    toElement : "responsibilityDef.name" from
   Scenario_Model;
              }
            }

        }

    # R8
        bi-directionalmapping DMClass_UCActor {
          fromMappingEnd CdmClassCompulsory {
            modelPackage : CdmPackage;
          isRootElement : false;
          cardinality : 1;
          roleName : Domain_Model;
          languageElementName : Class;
          }
          toMappingEnd UCActorOptional {
            modelPackage : UcPackage;
          isRootElement : false;
              cardinality : 0..1;
              roleName : UseCase_Model;
              languageElement : Actor;
          }

            nested mappings {
              nested mapping ClassName_ActorName {
```

```
                matchMaker : true;
                fromElement : "name" from Domain_Model;
                    toElement : "name" from UseCase_Model;
              }
            }
        }

    }
}
```