Modularity and reuse of domain-specific languages

Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen
op dinsdag 7 november 2017 om 16:00 uur

door

Ana Maria Şutîi

geboren te Bacău, Roemenië

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. B. Koren
promotor:	prof.dr. M.G.J. van den Brand
co-promotor:	dr.ir. T. Verhoeff
leden:	prof.dr. J.J. Lukkien
	Prof.Dr. B. Rumpe (RWTH)
	prof.dr. E. Scott (Royal Holloway University)
	dr. E. van Wijk (University of Minnesota)
	prof.dr. J.J. Vinju (Centrum van Wiskunde en Informatica)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod

Ana Maria Şutîi

Promotor: prof.dr. M.G.J. van den Brand
(Eindhoven University of Technology)

Copromotor: dr.ir. T. Verhoeff
(Eindhoven University of Technology)

Additional members of the core committee:

prof.dr. J.J. Lukkien (Eindhoven University of Technology)
prof.dr. J.J. Vinju (Centrum Wiskunde en Informatica)
prof.dr. B. Rumpe (RWTH Aachen University)
prof.dr. E. Scott (Royal Holloway University of London)
prof.dr. E. van Wyk (University of Minnesota)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2017-09.

Part of the work in this thesis has been carried out as part of the European Union's ARTEMIS Joint Undertaking for CRYSTAL - Critical System Engineering Acceleration - under grant agreement No. 332830.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-4379-3

Printed by the print service of Ipskamp Printing, Enschede, The Netherlands

Acknowledgements

My PhD has been an incredible journey, during which I have grown both professionally and personally. Besides the technical knowledge, the PhD made me more confident in my abilities to take on challenging tasks and it has shown me how to deal with uncertainty. It has also taught me to be more self-disciplined and resilient in stressful times. On the fun side, during these four years, I attended inspiring conferences, workshops and summer schools in various parts of the world. There, I met inspiring people and I have visited beautiful places. These people and the places I visited, all left a profound mark on me.

It is now time to express my gratitude towards people who have guided, supported, and encouraged me during the four years of my PhD. In the next paragraphs, I will mention the people that had the greatest influence on me in this period.

First and foremost, I am really grateful towards my two supervisors, Mark and Tom. The guidance, liberty, and trust they gave me was of uttermost importance for the completion of my PhD. Mark has always been the one to give me the high-level overview and to put my work into perspective. Tom has always been the one to discuss the details of my work and to show me the beauty of abstraction. Thank you both for the great, inspiring talks we had these four years. You made my PhD journey a pleasure. Both of you have showed me the joy of good mentorship and guidance. Moreover, thank you, Tom, for the tennis games we played Saturday mornings, and the small conversations in Dutch we always had after each game.

I have always cherished the moments spent with my colleagues. We enjoyed numerous drinks, barbecues, coffees, and chit-chats together. They have made life at the university a real delight. My south American colleagues helped me improve my Spanish and Portuguese, they got me into playing football, and they showed me what a good barbecue really is. Moreover, they introduced me to "farofa" (no more barbecues without it) and the famous Pernambuco "bolo de rolo". My Asian colleagues brought me numerous flavors from Asia. They showed me the Chinese "sweets", the real pistachios (not the ones you find in European shops), the amazing saffran, the smoked tea, and so many other flavors. I also had a failed attempt to learn Chinese. Last, but not least, my European friends showed me how diverse Europe itself is and how, despite the differences, we share dishes that I thought were exclusively romanian! I am especially grateful to my dutch colleagues who have shared precious tricks and tips on how to get around the dutch culture. I wholeheartedly thank my colleagues for showing me the joy of diversity. Among my

colleagues, I mention Yanja Dajsuren, Önder Babur, Dana Zhang, Ulyana Tikhonova, Felipe Ebert, Weslley Silva Torres, Neda Noroozi, Sander de Putter, Josh Mengerink, Fei Yang, Alexander Serebrenik, Anton Wijs, Luna Luo, Mahmoud Talebi, Thomas Neele, Alexander Fedotov, Mauricio Verano Merino, Kousar Aslam, Rodin Aarssen, Priyanka Karkhanis, Guilherme Amaral Avelino, Sangeeth Kochanthara, Omar Alzuhaibi, Sjoerd Cranen, Sarmen Keshishzadeh, Bogdan Vasilescu, Aminah Zawedde, Yuexu Chen, Maarten Manders, Dragan Bosnacki, Tim Willemse, Hans Zantema, Jaewon Oh, Alexander Aroyo, Miguel Botto Tobar, Loek Cleophas, Rob Faessen, Frank Peter, Arash Khabbaz Saberi, Luc Engelen, Arjan van der Meer, Maciej Gazda, Raquel Alvarez Ramirez, Ion Barosan, Ad Aerts, Harold Weffers, Maggy de Wert, Margje Mommers-Lenders, Tineke van den Bosch, and Willeke Quaedflieg.

During these four years, I got involved in various associations and clubs, that have made my PhD life much more enjoyable and fulfilling. I especially thank my improv friends, who have provided for numerous magical evenings at the International Hub in Eindhoven. They have showed me the joy of living in the moment. I express warm thanks to my Toastmasters friends as well. They have showed me the joy of sharing personal stories. Last, but not least, I express gratitude to my PhD Council friends. Organizing events for the PhD community was a very rewarding experience. They have showed me the joy of giving back to the community.

I express sincere gratitude to my friends, with whom I have spent many wonderful evenings and weekends. Besides spending relaxing times together, we had many stimulating conversations about life and work. They have showed me the joy of friendship.

I would also like to convey my gratitude to the students that I have supervised (Jia Zhang, Stef van Schuylenburg, and Nanne Wielinga) or lectured (2016-2017 PDEng trainees) during these four years. They have showed me the joy of sharing knowledge.

I am also extremely grateful for the opportunity I have been given to work at Fabien Campagne's laboratory at Weill Cornell Medicine in New York. I especially thank Mark for allowing me to do this internship at a short notice, and Fabien for guiding me through the entire period. Besides domain-specific language technology, I have learned a great deal about biology, statistics, and data analysis during this period. It has been a truly intense learning period. Moreover, I will never forget the magical evenings spent in New York and the numerous people I met there from all over the world.

I want to thank people from different companies that, during my PhD, have allowed me to present my work at their offices, and have given me valuable feedback. In no particular order, I thank Bart Theelen from Océ, Eugen Schindler from Océ, Hristina Moneva from Océ, Klemens Schindler from Sioux, Remi Bosman from Sioux, Arjan Mooij from TNO, Markus Voelter from mbeddr, Marc Hamilton from Altran, Ramon Schiffelers from ASML, and Rob Ekkel from Phillips.

I am also extremely grateful to Charise Walraven, who has designed the cover of this thesis. The cover represents an abstract representation of my work. Thank you, Charise, for putting up with me while going through several rounds of revisions.

I would also like to express my deep gratitude to the members of my reading committee, who had the patience to go through my thesis and give me valuable feedback: Erik van Wyk from University of Minnesota, Elisabeth Scott from Royal Holloway University of London, Bernard Rhumpe from RWTH Aachen University, Jurgen Vinju from Centrum Wiskunde en Informatica, and Johan Lukkien from Eindhoven University of Technology.

I am concluding this section with the people who are dearest to me. I have been blessed with what might be the most loving and the most inspiring family in the world. My parents, my sister, and my husband have always been with me, through the happiest

and saddest times. I will conclude this section with a big and warm thank you to them! They have showed me the joy of unconditional love and support.

Ana Maria Şutîi
Eindhoven, September 2017

Table of Contents

Chapter 1

Introduction

In this thesis we present MetaMod, consisting of a collection of mechanisms and meta-tools, that tackles modularity and reuse in the creation and application of domain-specific languages. The research we did with MetaMod was an exploration of ideas on better ways to create domain-specific languages, with an emphasis on the modularity and reuse qualities. A subset of these ideas lead to a collection of mechanisms for the design and implementation of DSLs, that we embodied in the implementation of our meta-tools. In the introduction of the thesis we motivate our research, we present its highlights, we discuss the research questions, we sketch an outline of the entire thesis and we make our research strategy explicit.

1.1 Background

The history of software engineering is one of raising the level of abstraction [19]. Programming has evolved from machine code with strings of zeros and ones, and assembly code with mnemonics, to higher-level programming languages with loops, conditionals, classes, traits, etc. This raising of the abstraction level has brought an incredible productivity boost in writing programs and it has also allowed more people to develop software. The productivity boost happened because this higher abstraction level has decreased the following gap. On one hand, there is the conceptual model in the mind of the software developer who wants to solve a problem. On the other hand, there is the representation of the solution for the problem in the program, solution that needs to be molded with the abstractions available in the programming language. The difference between the conceptual model and the actual solution in the programming language represents the gap aforementioned.

In most cases, the gap between the conceptual models in the mind of the software developers and the representation of the solution in higher-level programming languages is still daunting. The abstractions introduced in these higher-level programming languages are abstractions in the solution space [124] because the concepts used to create programs have a computing-focused flavor (*for* loops, *while* loops, etc.). Besides the solution-space

abstractions, one could argue that there is another kind of abstraction in these higher-level programming languages. The other kind of abstraction can be best illustrated with object-oriented programming (OOP) [103] in Java. This kind of abstraction lives in the mind of the beholder (user), not in the language itself. When programming in Java, for instance, the user understands certain predefined classes as abstractions and subsequently uses them as such, rather than as features of the programming language. Consider, for instance, class 'Animal' with methods 'eat' and 'sleep'. The user can treat the 'Animal' class as an abstraction of a real animal, and methods 'eat' and 'sleep' as an abstraction of the actions the animal can perform. Nonetheless, these abstractions are intertwined with general-purpose abstractions such as overloading, overriding, virtual functions, etc. This intertwining breaks the illusion of programming using concepts from the problem space. A similar argument holds for other mechanisms, such as software libraries. Moreover, these 'in-language' abstractions, in languages like Java, do not provide domain-specific optimizations or static error checks that DSLs can provide.

Abstractions of the problem space [124] itself would help in closing the aforementioned gap because concepts in programming languages would be concepts from the application domain itself (*train*, *station*, etc.). There are new methodologies that advocate the introduction of abstractions of the problem space in the programming languages. Two of these methodologies are model-driven engineering (MDE) and language-oriented programming (LOP).

Model-driven engineering advocates to raise the level of abstraction through the use of models (abstract representations of the real world) in software development and it uses, among others, domain-specific languages (DSLs) and code generators to achieve its goal [124]. The goal of MDE and the means to achieve it relate to language-oriented programming. LOP advocates the creation and application of domain-specific languages to express solutions in various domains [39, 51, 162]. In particular, when given a new problem, a DSL engineer[1] creates one or more DSLs (if these DSLs do not already exist) that a DSL user[2] employs to express the solution. This solution is expressed at a higher level of abstraction, using concepts from the problem domain itself. The solution can be subsequently transformed to executable code as well.

In MDE parlance, the central language aspect[3] of a DSL is the metamodel. The metamodel describes the kinds of nodes and edges possible in a graph data structure that contains all the relevant information from a program; this graph data structure is called the abstract syntax. The metamodel itself consists of *concepts* (equivalent to the nodes) and *relations* (equivalent to the edges) between these concepts. Besides the metamodel, a DSL also involves *processing units*[4], which take models conforming to the metamodel as input and produce something useful from them as output, such as executable code or visualizations. A processing unit can be related to any of the following language aspects: static semantics, interpretation, code generation (more generally, model transformations), editor, etc. Thus, a processing unit implements a language aspect. It is the metamodel and its associated processing units that the DSL engineer needs to provide for a complete definition of a DSL.

Traditionally, the creation of a DSL was a time-consuming endeavor. Fortunately, their development has been eased with the introduction of specialized environments called

[1] The DSL engineer is a software developer responsible for implementing DSLs.

[2] The DSL user is a person (ideally domain expert) using a DSL.

[3] A language aspect is a constituent part of the definition of a DSL (e.g., code generation, constraints, editor, etc.).

[4] Note that the term processing unit is a term that we introduced.

language workbenches [51]. The language workbenches ease the development of DSLs by offering meta-languages to implement the different language aspects, such as editor, code generation, constraints, type system, etc. There are also language workbenches that offer a single, powerful meta-language to develop all the language aspects of a DSL (e.g., Rascal [12]).

No matter what facilities they offer to implement the different language aspects of a DSL, language workbenches need to provide good support for modular and reusable DSLs as well, in order to fulfill the vision of MDE and LOP. That is because many DSLs have common parts. For instance, many DSLs need a form of arithmetic expressions. Thus, reusing arithmetic expressions from other DSLs would speed up the development time of the new DSLs. It would also increase the quality of the new DSLs if the reused DSLs are stable. Moreover, in LOP, a combination of DSLs is needed for the implementation of an application, further motivating the need for modularity and reuse. Additionally, the benefits of modularity and reuse are already long recognized in software development and other engineering disciplines (see Chapter 2). By extension, modularity and reuse should benefit the development of DSLs as well, DSLs that are at the core of MDE and LOP.

One type of language workbench that enables more modularity and reuse in the creation of domain-specific languages is the projectional language workbench [160]. This kind of language workbench uses projectional editing, which allows users to directly manipulate the abstract syntax of the program through the actions they perform when creating the program in the editor. This is different from the classical way where users manipulate the text of programs and the compiler parses the text into an abstract syntax. Language workbenches that employ parsing technologies for the editors exhibit difficulties in combining the textual notations of the DSLs. Note that we refer here to the difficulties that the DSL engineers encounter when they combine languages with the purpose of creating a new language. This is not the case for projectional language workbenches, because they do not need to parse text or any other notation. Circumventing the difficulty of combining languages when parsers are involved [68] enables DSL engineers to more easily mix DSLs that have different notations (not only textual). Arguably, one of the most advanced projectional language workbenches developed so far is Jetbrains MPS [65]. It was successfully used to implement a considerable amount of inter-operating DSLs [158]. The modularity of DSLs in MPS is based on mechanisms similar to object-oriented programming in Java.

Looking at modularity and reuse of DSLs in various language workbenches, one can observe two general tendencies. Many language workbenches approach modularity and reuse using only inheritance-like mechanisms, or they approach it only outside of the modeling formalism itself (not considering modularity and reuse in the modeling formalism). Besides these two tendencies, the implementation of DSLs is a particular case of software development, so the particularities of DSLs (such as the separate definition of the metamodel and that of the processing units, or the hierarchies created by the modeled domain itself) could be further leveraged in language workbenches. Based on the two tendencies and the observation, we formulate three guiding principles for MetaMod: *do not use only inheritance-like mechanisms*, *treat modularity and reuse starting from the modeling formalism* and *leverage characteristics of DSLs in the features of MetaMod.*

Crystal European Project The work carried out during the four year PhD was partially supported by the Crystal European Union project [6]. The overall goal of the project was to establish an interoperability specification and a reference technology platform for safety-critical systems. In particular, we were involved in the DSL brick,

that was responsible to provide capabilities to design domain-specific languages and automatically generate code and other artifacts from the models.

1.2 Highlights of our research

In the next paragraphs, we give the main highlights of our research. To guide us during our research, we have used the three principles formulated in the previous section. As already argued, many researchers and tool developers approach modularity and reuse in the creation of DSLs employing mechanisms similar to inheritance only (principles one), or solving it outside of the modeling formalism only (principle two). That is why, we set to explore features of modularity and reuse for the creation of DSLs in language workbenches that depart from these two tendencies and that, at the same time, leverage the particularities of DSLs during their creation (principle three).

Principle one Many of the existing formalisms for creating languages (e.g., mechanism in Jetbrains MPS [65]) approach the DSL modularity and reuse with inheritance-like mechanisms, but we think that this brings unnecessary restrictions to the modularity of DSLs. These inheritance-like mechanisms, for instance, allow very little adaptation to the metamodel of a reused DSL in a reusing context. Our mechanisms are more flexible when it comes to adapting the metamodel of a reused DSL in a new context. We give details on this in Chapter 4.

Most work on the reuse of DSLs focuses on the reuse of entire DSLs (with some adaptations); see Section 5.3. We also looked at how to reuse only the processing units of DSLs where the metamodels of these DSLs are conceptually similar, but do not necessarily have the same structure (see Chapters 7 and 8). Again, these two types of reuse use different mechanisms than inheritance-like mechanisms.

Principle two In formalisms for creating languages that do not cater for modularity (e.g., Ecore [135]), this has been dealt with outside of the modeling formalism only (e.g., Melange [35]). We will argue that modularity should start from the modeling formalism itself and go to the processing unit level. This is because, the organization of the metamodels themselves can influence modularity and reuse of processing units. That is why, we decided to create our own collection of mechanisms and meta-tools for the design and implementation of languages, MetaMod, and not adapt an existing formalism for creating languages. MetaMod is both the mechanisms for creating languages and their implementation in meta-tools. We do not call MetaMod a language workbench because we do not have a mechanism and associated meta-tools for custom editors in MetaMod (which is considered to be a constituent part of a language workbench). Apart from that, MetaMod has support for all the other parts of a language workbench. In most places in this thesis it is safe to equate our mechanisms and meta-tools with language workbenches. When that is not the case, we make it clear in the discussion.

Principle three The implementation of domain-specific languages is a particular case of software development, and thus, one can take advantage of the particularities of DSLs. The two reuse mechanisms mentioned in the previous paragraphs take advantage of the fact that metamodels and processing units are defined separately (characteristic of MetaMod). Furthermore, we exploited hierarchies created with the metamodel in the processing units (see Chapter 5).

Modularity of models of DSLs Most of our research was geared towards the modularity and reuse features that language workbenches offer for the creation of domain-specific languages. Nonetheless, the way we designed MetaMod allowed us to have the same modularity and reuse features for the models of the DSLs, as for the metamodels of the DSLs (see Chapter 4 and Chapter 6). The fact that these features are already enabled at the model level means that the DSL engineer does not need to add them to the particular metamodels of the DSLs they are developing, further easing the development effort of a DSL.

Implementation of MetaMod As for the meta-tools, we implemented them using Jetbrains MPS. The definitions and explanations of the mechanisms we introduced are accompanied by the implementation of these mechanisms in meta-tools in Jetbrains MPS. We encourage the readers to consult the implementation[5] in case of technical doubts. Furthermore, it is at the implementation level that the relation to projectional editors comes into play, because we rely on the projectional editing facility of MPS in MetaMod. That is why we do not consider modularity and reuse for scanning and parsing [141]. As a result, the default editors of MetaMod are projectional editors.

Exploration Coming up with the mechanisms for the design and implementation of DSLs was a real exploration. This can also be seen from the commit history of the implementation of MetaMod, where we have also implemented ideas that we do not mention in this thesis. Moreover, in the thesis, Chapter 7 and Chapter 8 refer to two explorations we made in previous versions of MetaMod. One can go back to these versions by checking out the right branch from the Github repository of our project. In that sense, version control systems are excellent tools for explorations and for revisiting a previous exploration.

One phenomenon we observed in our explorations is that no single feature for modularity and reuse is a panacea for all problems. Such features always require trade-offs with various qualities, and they can be often misused. Not even in software development is there an universally-accepted mechanism for modularity and reuse.

What complicates the discussion In explaining MetaMod, we often need to deal with multiple levels at which definitions occur in the programming world, which make the discussions harder to follow. To get a high-level understanding of these levels of the world, imagine one is *defining a program* (also know as the model). This is what we call the first level of the world. To write this program, you need a language (also known as the metamodel). Imagine now that we are in a setting where we can *define this language*. This is the second level of the world, and this is another kind of definition. Moreover, we do not use an existing formalism to define the language, so we need to *define the formalism* as well. This is called a meta-metamodel; it represents the third level of the world and it requires yet another kind of definition. We are going to switch often between these levels of the programming world in the thesis, that of the meta-metamodel, the metamodel and model. There are important interactions between these levels of the world, and we will touch on them in the remainder of the thesis.

We summarize the previous paragraph in Table 1.1.

The meta-tools of MetaMod play a role in both the first level and the second level; this is also the reason to call them 'meta'-tools. Using the meta-tools, one can define

[5] `https://github.com/farcasia/MetaMod`

Definition of ...	Related to the ... level of the world
Program	Model
Language	Metamodel
Formalism for creating a language	Meta-metamodel

Table 1.1: Table depicting the level of the programming world to which a certain definition relates.

metamodels, models complying to metamodels and semantic-carrying operations on the models.

What further complicates the discussion is the fact that we use one formalism to describe both metamodels and models, but we try to make things clear in Chapter 4.

1.3 Research Questions

In wishing to accomplish the vision of MDE and LOP as discussed in Section 1.1, we have set to answer a main research question, that is subsequently split into six, more focused, research questions.

The main research question is an overarching question and it is related to finding elements that both MDE and LOP need in order to accomplish their vision, more specifically, elements of modularity and reuse in the implementation of a DSL. Thus, we ask the following.

> **RQ:** *What are effective ways to achieve modular and reusable definition, implementation, and application of domain-specific languages?*

Answering this main question, (1) we need to identify the reasons and requirements for modularity and reuse of domain-specific languages, (2) we need to propose ingredients that accomplish modularity and reuse of domain-specific languages and (3) we need to see how the proposed ingredients behave in practice. These concerns are treated by the next six research questions as follows. The first concern is addressed by RQ_1, the second concern is addressed by RQ_2, RQ_3, RQ_4, and RQ_5, and the third concern is addressed by RQ_6. Among these, the second concern addresses the core of the main research question.

Before we go on to the questions that tackle the core of the main research question, we first set the context for and motivate RQ. We do so by diving deeper into modularity and reuse, by giving reasons for wanting these two qualities in the implementation of a DSL and by stating requirements for modularity and reuse of domain-specific languages. Specifically, we ask

> **RQ_1:** *What are reasons and requirements for modularity and reuse in language workbenches?*

Many works that address the problem of modularity and reuse of DSLs, do so outside of the modeling formalism itself. That is, they take existing formalisms and add operators on top of them to make up for the lack of modularity in those formalisms. Although there are good reasons to do that, such as having to support a legacy formalism, we want to address modularity from the core of a DSL implementation; this should bring more opportunities to improve modularity. As a result, we look for a collection of mechanisms and meta-tools that tackle modularity and reuse starting from the core of a DSL, the metamodel. The question that we need to answer is

> **RQ_2:** *How can we organize metamodels of the DSLs such that we facilitate modularity and reuse of DSLs?*

Although the main language aspect of a DSL is its metamodel, and we address the modularity and reuse of metamodels with RQ_1, implementing the metamodel is only a part of the DSL implementation effort. We also need support for modularity in the processing units of these DSLs, processing units that are collections of operations defined on metamodels. We consequently need to address modularity and reuse at the level of operations as well. Thus, we ask

> **RQ_3:** *How can we organize processing units of the DSLs and the operations in the processing units such that we facilitate modularity and reuse of DSLs?*

The previous research questions addressed reuse of entire DSLs, that is, of both metamodels and operations. Those research questions do not concern the situation where metamodels of two DSLs, although conceptually similar, have different structures. Reusing the operations of one of these DSLs in the implementation of the operations of the other DSL would be a further use case for reuse with DSLs. This leads us to asking the following question.

> **RQ_4:** *How can we facilitate reuse of operations despite structural differences among domain-specific languages?*

The previous questions were specifically focused on the implementation of a DSL. Looking at various DSLs, one can notice that there are recurring elements between these DSLs. For instance, the use of arithmetic expressions, or logical expressions in DSLs is a prevalent practice. These types of recurring elements, that come in the form of reusing existing DSLs, are tackled with RQ_2, RQ_3 and RQ_4. Nonetheless, there are also recurring elements that can be addressed differently. For instance, many DSLs need the capability to define models at separate places and to combine them, or to create small variations of model fragments. DSL engineers thus need to add mechanisms to modularize their models in the DSL itself. To overcome this, and to further ease the development of DSLs, we ask

> **RQ_5:** *What modularity and reuse mechanisms can be applied to models, irrespective of the DSL?*

We finally look at how the mechanisms and meta-tools for the design and implementation of DSLs, that we introduced while exploring the preceding research questions, behave in practice. Specifically, we ask

> **RQ_6:** *How can modularity and reuse features of language workbenches be evaluated?*

These research questions are addressed in the remainder of the thesis. What research questions are tackled by each chapter is discussed in the next section.

1.4 Outline

Following is the outline for our thesis on MetaMod, a collection of mechanisms and meta-tools targeting modularity and reuse in the creation of domain-specific languages. The thesis is split into three main parts. The first part, comprising Chapters 1 through

3, establishes the research questions, and sketches the context for our research and the requirements for language workbenches. The second part, comprising Chapters 4 through 6, depicts the main mechanisms that we devised to achieve modularity and reuse of domain-specific languages. Finally, in the third part, comprising Chapter 9 and Chapter 10, we evaluate the practicality of the introduced techniques and we revisit our research questions.

Chapter 2: Setting the context In this chapter, we review the main technologies and the terminology we use throughout the thesis. Although some parts of it could be skipped by those familiar with the topic, some of our terminology deviates, for good reasons, from the more common alternatives. Thus, this chapter should be at least skimmed through.

Chapter 3: Language workbench requirements for modularity and reuse In this chapter, we sketch criteria that a language workbench needs to fulfill in order to be fit for creating modular and reusable domain-specific languages. The chapter starts with a description of two DSL qualities: *the modularity and the reusability qualities*. We first glance at modularity and reuse in other sciences and engineering disciplines, and then we look at modularity and reuse in general software development. We also give arguments for modularity and reuse of DSLs in this first part. We then go on to describe other DSL qualities that interact with the two main qualities, that of modularity and reuse. The choice of some qualities requires trade-offs with other qualities (doing better on one will deteriorate the other), or it is aligned with other qualities (doing better on one will improve the other as well). We then show how modularity and reuse in the creation of domain-specific languages is related to the expression problem [168] from software development; the expression problem in software development defines criteria for the extensibility of software. Finally, we define language workbench requirements by rephrasing the criteria for the expression problem. This chapter offers an answer to research question RQ_1 and is partly based on the following publication.

> [139] E. Schindler, and K. Schindler, and F. Tomassetti and A.M. Şutîi. Language Workbench Challenge 2016: the JetBrains Meta Programming System. *Language Workbench Challenge*, 2016.

Chapter 4: MetaMod In this chapter, we define the meta-metamodel of MetaMod and the other meta-languages implemented in MetaMod, and we provide a complete example of a DSL built with MetaMod. This chapter offers a complete overview of the meta-metamodel of MetaMod, both in terms of our mechanisms and the meta-tools that support these mechanisms. It gives an overview of the features introduced for the benefit of both the metamodel and the model level. These features also include features that relate to modularity and reuse. As a result, this chapter addresses research questions RQ_2 and RQ_5. This chapter is partly based on the following two publications.

> [139] A.M. Şutîi, T. Verhoeff and M.G.J. van den Brand. Modular multi-level metamodeling with MetaMod. *Companion Proceedings of the 15th International Conference on Modularity*, 2016.
>
> [137] A.M. Şutîi. MetaMod: a modeling formalism with modularity at its core. *Companion Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015.

Chapter 5: Features of MetaMod In this chapter, we analyze the features of MetaMod from two perspectives: that of modularity and reuse, and that of fulfilling the language workbench requirements. This offers an answer to research questions RQ_2 and RQ_3. The first part, related to modularity and reuse, finishes with a discussion on implementing the same DSL with both Jetbrains MPS and with MetaMod, which contributes to answering research question RQ_6 too. At the end of the chapter, we also make an extensive comparison with existing language workbenches concerning modularity and reuse. This chapter is partly based on the following publication.

> [140] A.M. Şutîi, T. Verhoeff and M.G.J. van den Brand. Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod. *Computer Languages, Systems & Structures*, 2017.

Chapter 6: Modularity of value models Here, we describe the modularity mechanisms that MetaMod is able to provide for models, mechanisms that are the same as for the metamodels. MetaMod makes these mechanisms available in any model, no matter what DSL they conform to. Because these modularity mechanisms do not need to be created in the DSL itself, the effort of creating the DSL decreases. The chapter exclusively addresses RQ_5.

Chapter 7: Reuse mappings In this chapter, we exclusively address RQ_4. We present a mechanism that achieves reuse of operations when the metamodel of the reusing DSL and the corresponding part of the reused DSL metamodel differ. The mechanism we present here has some limitations, part of which we mitigate with the mechanism presented in the next chapter.

Chapter 8: Delegated operations In this chapter, we also exclusively address RQ_4. We present another mechanism of reusing operations despite of structural differences among DSLs. This new mechanism is more flexible and allows more structural differences among the reused and the reusing DSL than the mechanism presented in the previous chapter.

Chapter 9: Evaluation In this chapter, we reflect on some DSLs that we developed using MetaMod: the expression DSL, the Kaja DSL, the shapes DSL, the bootstrapping of a subset of MetaMod, and a group of smaller DSLs made of the state machine, Petri net, and graph DSLs. These DSLs range from small sizes to larger sizes and from imperative to declarative. We highlight what elements related to modularity and reuse were eased with MetaMod and what major design decisions we made with these DSLs. Thus, this chapter addresses research question RQ_6. This chapter is partly based on the following publication.

> [140] A.M. Şutîi, T. Verhoeff and M.G.J. van den Brand. Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod. *Computer Languages, Systems & Structures*, 2017.

Chapter 10: Conclusions In this chapter, we revisit the research questions and summarize the findings for each of them.

1.5 Research strategy

To set the right expectations, in this section, we report on the research strategy we used in our work. We categorize our research strategy as suggested by Shaw [128]. The research strategy is made of three elements: what kind of research questions do we investigate, what kind of result do we produce, and what kind of validation do we do? We are going to investigate each of these elements separately in the next paragraphs.

Firstly, the research questions we ask are related to a *method or means of development*. In particular, we ask questions about how can one better define, develop and use modular and reusable domain-specific languages. This translates into ways to organize metamodels, processing units and their operations, and models.

Secondly, the kind of result we produce is a *technique accompanied by a tool*. In particular, the technique we create with MetaMod is arguably a better way of defining and implementing modular and reusable domain-specific languages. The meta-tools of MetaMod embody this technique and is publicly available on a Github repository[6].

Thirdly, the kind of validation we produce is based on *examples*. We use a breadth of examples for validation, from toy examples (the shapes and route DSL examples) to examples that are slices of reality (the bootstrapping example and the expression DSL) and standard examples (state machines, Petri nets, graphs). We reflect on these implementations and their design, using the modularity and reuse lenses. Moreover, we have two DSLs that are reimplementations of DSLs from Jetbrains MPS, and we can thus make comparisons between MetaMod and MPS more directly.

[6] `https://github.com/farcasia/MetaMod`

Chapter 2

Setting the Context

In this thesis we make a series of explorations, during which we touch upon several methodologies and technologies related to domain-specific languages. This chapter briefly explains the terminology for these related points. Specifically, these points consist of two methodologies that employ domain-specific languages to accomplish their goals: model-driven engineering and language-oriented programming; a more detailed explanation of domain-specific languages; and a technology used for building DSLs, that of language workbenches. We go through these points because it is important that one understands the ideas behind them for the rest of the thesis. Furthermore, in this chapter we highlight our preferred terminology when alternatives are presented, or we even introduce our own terminology. For this chapter, and all other chapters of the thesis, we assume that the reader is fairly familiar with object-oriented programming concepts.

2.1 Model-driven engineering

This section defines one methodology that we use in the rest of our thesis, that of model-driven engineering. We start with a visual story that goes through most elements of MDE, and we then explain the formal ingredients that constitute MDE. We finish this section with a presentation of the most common standards and frameworks for MDE.

We start with a description of how building a modular house would work using MDE. Consider a construction company creating houses. Assume that this company also has a technology to build houses from prefabricated pieces (see right-hand side of Figure 2.2[1]). They have 5 types of prefabricated pieces available: wall with door, wall with window, simple vertical wall, floor block, and ceiling block (see Figure 2.1). All these prefabricated pieces have square sides and they all have the same edge length. There are also connections possible among these pieces, in the form of edge to edge placements, resulting in either 180° angles or 90° angles among the pieces. The types of prefabricated pieces and the

[1]The container houses image is published at `https://www.flickr.com/photos/javic/3195578220/in/gallery-sdscad1-72157623532972207/` by Javier Carcamo under license CC BY-NC-SA (`https://creativecommons.org/licenses/by-nc-sa/2.0/`).

possible connections among them form a metamodel in MDE. The metamodel describes the valid constructs that can be used in models and the relationships among them. Using these types of prefabricated pieces, we build a model. Consider that we choose six of such pieces: a floor, a ceiling, one window wall, one door wall and two vertical walls. Connected in the right way, these give rise to a box in the form of a house (see left-hand side of Figure 2.2). This model is a representation of a real house (see multiple such houses on the right-hand side of Figure 2.2). Now that we have a model of the house, we want to transform this model in meaningful ways. This process is called model transformation in MDE. In this case, we wish to transform the model of the house into different other models, e.g., a 2D plan (to be shown to the municipality for approval), or a model showing the estimated costs. The model can be used in many other phases of the development of a house.

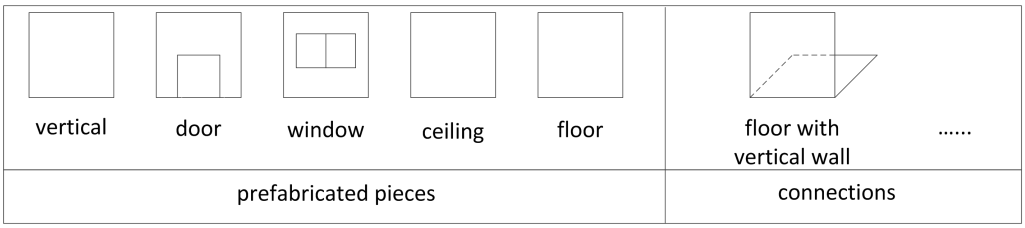


Figure 2.1: Metamodel elements and connections for creating houses out of prefabricated components.

Model-driven engineering [23] raises the level of abstraction in programming languages and brings them closer to the domain of operation. This should result in reducing accidental complexity in software development (complexity introduced by the development tools, platforms etc.), leaving software engineers to deal mostly with essential complexity (complexity related to the problem domain itself) [81]. MDE [124] eases software development through the manipulation of models of the problem domain in the development process. Moreover, these models form the basis for model transformations. The model transformations play a central role in MDE. A model transformation defines how a model conforming to metamodel MM_a transforms to a model conforming to metamodel

Figure 2.2: Model of a house formed of prefabricated pieces on the left-hand side, and houses made of prefabricated pieces in the real world on the right-hand side.

MM_b. The definition of the model transformation is made on the metamodels, and the application is made on the models. There are multiple ways to define model transformations, from using general-purpose programming languages to using specialized languages, e.g. ATL [69], ETL [80], and QVT [111]. Usually, sequences of transformation reduce the level of abstraction until getting to executable code. There are many types of model transformations [99]. Among these types, we mention refactoring, refinement, language migration, optimization and code generation.

2.1.1 Four-level metamodel hierarchy

One of the most important technical points in MDE is the classical four-level architecture for metamodeling and its related terms. This four-level architecture governs many of the MDE tools. It was proposed by the Object Management Group (OMG) [114]. The metamodeling architecture consists of the following four levels: meta-metamodel, metamodel, model and object. Next, we describe these levels and other closely-related terms.

A meta-metamodel defines a meta-language to express metamodels and it lives at level *M3*. The meta-metamodel determines the constructs available to create metamodels. Usually, the meta-metamodel is self-defined; that means its structure is described with the constructs of the meta-metamodel itself.

A metamodel is an instance of a meta-metamodel and it lives at level *M2*. At the metamodel level, a language for specifying models is defined; the abstract syntax of this language is defined. In the modular house example, the metamodel would be the types of prefabricated pieces and the possible connections among them.

A model is an instance of a metamodel, and it lives at level *M1*. The model is a simplified representation of a part of the world, named the system [99]. This is the user specification level, where users model such systems [114]. In the modular house example, the model would be the representation on the left-hand side of Figure 2.2.

The objects of the model are run-time instances of model elements defined in the model. They live at level *M0*. In the modular house example, the object level would be the actual house constructed in reality and captured in the picture on the right-hand side of Figure 2.2 (the picture actually depicts more than one instance of such a house).

For a better understanding of the different levels of modeling, we will make a couple of parallels to other technological spaces [86, 155]. We make parallels to general-purpose programming languages in general, to Java programming, to database specifications and to the grammarware world. These parallels are captured in Table 2.1.

Style	In ...	express a ...	defining ...
MDE	Meta-metamodel (M3)	Metamodel (M2)	Models (M1)
Programming	Programming language	Data type definition	Data type values
Java programming	Java	Class definition	Objects
Databases	SQL	Database schema	Database content
Grammarware	BNF	Grammar	Text (in language defined by grammar)

Table 2.1: Parallels between the levels in MDE and programming languages [155].

Note that a metamodel is also a model (a model of models), so there are places in the

thesis where we say model and we refer to both kinds of models, metamodels and models alike. That will be clear from the context.

There are three running terminologies when talking about a lower level in terms of its direct upper level. In this paragraph, one can replace level (either lower or upper) with any of the four levels (object, model, metamodel, or meta-metamodel). The lower level is "an instance of" upper level, the lower level "conforms to" the upper level, or the lower level is "a value of type" the upper level (see Figure 2.3). The relationships among the levels also imply relationships among the elements of those respective levels. The relationships mentioned earlier are called instantiation, conformance and typing. Some authors [16,49] suggest to use the conformance terminology, in order to distinguish it from classical instantiation in OOP. In this thesis, we often call instantiation the relationship between level *M2* and level *M3*, and we call conformance the relationship between level *M1* and level *M2*. We explain this choice in Chapter 4.1. There are also cases when we talk in terms of typing, and we specify the cases when we do that in the paragraph on "Alternative terminology".

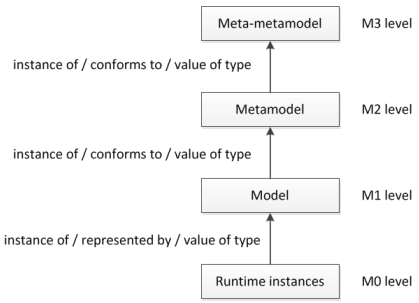


Figure 2.3: The four levels of the OMG metamodeling infrastructure and the three equivalent relationships between two adjacent levels.

To make the discussions in the following chapters easier to follow, we introduce some notations. We denote a model M conforming to a metamodel MM with $M :: MM$, and an element in model M, *MElement*, conforming to an element in metamodel MM, *MMElement*, with $MElement :: MMElement$. When we define a metamodel element, we write $MMElement :: \_$ because this is the level where users of a formalism (DSL engineers) start defining elements. We say that *MMElement* does not conform to any element. Actually, *MMElement* does conform to a general "'Concept" element from the meta-metamodel, but for simplicity and conciseness, we ignore this at the moment (see Chaper 4).

Constraints The metamodel itself imposes constraints (also called well-formedness rules) on models that conform to it. These constraints are imposed by the type of concepts in the metamodel and the type of relations that can exist between them in the metamodel. Only types of these concepts and relations can exist in valid models. Not all well-formedness rules can be captured with the metamodels though. Particular well-formedness rules can only be captured with user-defined constraints (also called validation rules). The most common form of an user-defined constraint is an invariant. The invariant is a property of the model that needs to hold at all times. For instance, consider a metamodel for modeling student courses. The metamodel can capture a rule saying that a course has at least one professor teaching it, but rules such as the end

date of a course cannot be before the start date of the course, cannot be captured in the metamodel itself (because metamodels ultimately only prescribe how model elements can be related). This kind of rule is captured with an invariant that needs to hold on the models. One example of a formalism used to express invariants in the MDE world is OCL [113].

Alternative terminology From all levels, we mostly use the metamodel and the model levels. We alternatively distinguish between metamodels and models using the following terms: type models and value models. A type model is a metamodel of other models, which we call value models. When using the terms *type model* and *value model*, there is no need to refer to the different models with the 'meta' keyword prepended to them. We have introduced this alternative terminology because it is closer to the classical terminology in general purpose-programming. We use this alternative terminology more often when we talk about a specific value model, and when we refer to its type model. On the other hand, when we do not need to refer to specific value models and we talk about a DSL, for instance, we refer to its underlying metamodel.

Given this alternative terminology, instead of $MElement :: MMElement$, we can also say that *MElement* is of type *MMElement*.

Multilevel metamodeling Atkinson et al. [8] show the limitations of classical fixed metamodeling levels and the unsatisfactory workarounds of these limitations, when it comes to modeling certain domains where more than two classification levels are involved (the four-level architecture has only the metamodel and model levels as classification levels). In such cases, the multiple levels need to be 'squeezed' or 'folded' inside either the metamodel or the model levels. To solve such problems, they introduce a multilevel metamodeling environment where an unrestricted number of levels is possible instead of only the metamodel level and the model level. This is achieved by allowing one model to play simultaneously two roles, that of a value model, and also that of a type model. Besides this uniformity, they also introduce the concept of deep instantiation, where properties defined at a certain level can be instantiated at more than one level below.

2.1.2 Major modeling standards and frameworks

In this part we discuss the major modeling standards and frameworks that have obtained industrial and academic adoption. In the following chapters, we often refer to these modeling standards and frameworks so that we show where does MetaMod stand among existing developments.

2.1.2.1 MOF / EMOF

Meta-Object Facility (MOF) [112] provides the basis for metamodel definitions for OMG's family of languages. Besides the capabilities for metamodel definitions, it also adds capabilities for model management. Essential MOF (EMOF) [112] is a subset of MOF that closely corresponds to the facilities found in OOP languages. Key modeling elements in EMOF are classes with properties and operations, associations among these classes and data types. Another important element in EMOF is the superclass relationship that exists between two classes and that states that the subclass inherits all properties, associations and operations of the superclass. An instance of a class is called an object in EMOF and an instance of an association is called a link in EMOF.

For modularity reasons, MOF introduced packages. Packages in MOF fulfill two goals, that of partitioning and that of extending metamodels. The partitioning goal is achieved with package import. Package import makes imported elements from the imported package directly visible in the importing package. One can specialize classes from the imported package or may use classes from the imported package as targets in new associations. The extending goal is achieved with package merge. After package merge, some classes (those with the same name as in the merged package) in the merging package get the features of the classes in the merged package.

2.1.2.2 UML

Unified Modeling Language (UML) [114] is used in the design and specification of software systems, with a focus on software systems written in an object-oriented programming language. UML is architecturally aligned with MOF in that MOF is the meta-metamodel of UML (see Table 2.2). UML advocates that the definition of a complex software system involves many views, and each of these views has a corresponding model diagram in UML (e.g. class diagrams, sequence diagrams, state machine diagrams or use case diagrams). The class diagram in particular, is of interest for the discussions in this thesis.

The most important elements of the UML class diagram are the following: the classes, the associations among these classes, the data types, and the generalization relationship between classes. As for modularity elements, UML introduced packages, which are used to group elements and to provide a namespace for the grouped elements (the package is a subtype of namespace). There are two types of relationships that involve packages: the package merge relationship, that defines how the content of one package is extended with the content of another package, and the package import relationship, that allows the use of unqualified names to refer to package members from other namespaces. These are similar to package import and merge from MOF.

The UML infrastructure document [114] also specifies how instances of UML models are represented. The instance of a class in UML is called an object, and an instance of an association in UML is called a link.

2.1.2.3 EMF

The Eclipse Modeling Framework (EMF) [135] is a modeling framework and code generation facility for building Java applications from model definitions. EMF is built within Eclipse [42], a generic framework for tool integration and a Java development environment. EMF is an open-source project with wide adoption in industry and academia, and it closely resembles EMOF.

The meta-metamodel of EMF is called Ecore. Ecore has its roots in MOF and UML, and was designed to map to Java implementations. An Ecore model is the primary source of information for the EMF code generator. The kernel of Ecore contains *EClass*, that models classes themselves, *EAttribute*, that models the fields of a class, *EDataType*, that models simple types and *EReference*, that models one end of an association between classes.

One of the modularity elements in Ecore is the package. A package groups related classes and data types. Packages can also have sub-packages. Another modularity element in Ecore is the proxy. A proxy is a cross-model reference in Ecore models, which means that separate Ecore models can reference objects between each other.

2.1.2.4 Levels for UML and EMF

Table 2.2 shows what elements play a role in the top three levels of the OMG metamodeling architecture in the case of UML and EMF.

Style	In ...	express a ...	defining a set of ...
MDE	Meta-metamodel (M3)	Metamodel (M2)	Models (M1)
UML	MOF	UML	UML models
EMF	Ecore	Ecore	Ecore models

Table 2.2: Elements of three metamodeling levels in the case of UML and EMF.

2.2 Language-oriented programming

In this section we discuss another methodology concerned with increasing the productivity and quality of software systems, language-oriented programming (LOP). In the end of the section, we also explain the connection between MDE and LOP.

Language-oriented programming [162] is an approach for software development that organizes software systems in a middle-out way. It starts from designing a domain-specific language [52] for the task at hand, instead of, or in addition to using general purpose programming languages for the task. It then continues in two outward directions, by developing code generation, interpretation or translation from the DSL (direction down) and by developing programs in the DSL (direction up) [162]; see Figure 2.4. LOP thus advocates the creation and application of domain-specific languages to express solutions in various domains [39, 51, 162]. In particular, when given a new problem, the DSL engineer creates one or more DSLs (if these DSLs do not already exist) that the DSL users employ to express the solution. This solution is expressed at a higher level of abstraction, using terms from the domain itself via DSLs. The solution can be subsequently transformed to executable code as well. This approach allows DSL users to concentrate on the problem domain, and it increases the productivity and quality of software development [162]. However, for the vision of LOP to be achieved, it should be easy to create and reuse DSLs and tools for DSLs.

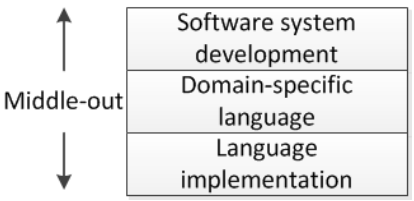


Figure 2.4: LOP and middle-out development [162].

Note that there are slight variations on the ideas of LOP in the form of extensible languages, e.g. LISP, F#, Scala, and Groovy [62], or in the form of Unix's little languages, e.g. AWK [3] and make [50].

Besides the main domain, in a software system, there are often other secondary domains playing an important role, such as deployment, user interfaces, paying systems, security, etc. Using DSLs for each of these domains requires that DSLs can be composed such that

the complete application can be written using DSLs. Thus, language composition plays an essential role in LOP. This is reinforced by Erdweg et al. [44], that have identified five types of language composition in LOP: language extension, language restriction, language unification, self-extension and extension composition.

Both LOP and MDE want to achieve the same goal, to increase the productivity and quality of software development. Moreover, both of them make use of DSLs to achieve this goal (see next section for the relation among MDE and DSLs). Where they differ is in terminology and where they put their focus on. In LOP, the language (metamodel) and its composition with other languages are at the center, while in MDE, the model and model transformations are at the center. We use both perspectives in the thesis.

2.3 Domain-specific languages

In this section, we present domain-specific languages in more depth. To get a better understanding of DSLs, we make comparisons to common technologies, we categorize DSLs and related aspects, and we present benefits and challenges of DSLs.

Domain-specific languages are computer programming languages of limited expressiveness, focused on a particular domain [52]. DSLs allow concise, understandable and transparent expression with the goal of improving productivity and quality.

The definition of a domain-specific language is ambiguous because it relies on the term "domain", which is itself ambiguous. Domain-specificity is not a clear-cut property, but rather a gradual one. For a better understanding of DSLs, Voelter et al. [157] show common differences between domain-specific languages and general-purpose languages, e.g. GPLs are always Turing-complete, while DSLs are often not; GPLs have a large and complex domain, while DSLs have smaller and well-defined domains; GPLs have a lifespan of years to decades, while DSLs have a lifespan of months to years; and the evolution of GPLs is often slow and standardized, while the evolution of DSLs is fast-paced.

The Sapir-Whorf hypothesis [164] states that the principle of linguistic relativity holds that the structure of a language affects its speakers' world view or cognition. If we translate this to the programming world, the programming language that one uses affects the way the programmer thinks about a solution. Besides languages, tools (languages can also be considered tools here) can also affect the programmers' cognition; "The tools we use have a profound (and devious!) influence on our thinking habits, and therefore, on our thinking abilities." [107]. Thus, DSLs can shape the form of the software solutions. A well-designed DSL increases the chances of a well-designed software system. Users need languages that allow them to program using concepts from their domain (that would ultimately get translated into executable code). This would close the gap between the conceptual model that programmers have about the application and the actual implementation [38].

To better understand DSLs, we compare them to a more well known technique for encoding domain-specific knowledge, that of software libraries. Software libraries offer a collection of functions that solve domain-specific tasks. The difference is that software libraries are dependent on one general-purpose language, do not have a dedicated concrete syntax, and do not offer static error checking, static optimizations, or relevant IDE support [157].

Several authors, among which Kurtev et al. [87] observed the convergence of MDE and DSLs, in that MDE is more and more related to DSL engineering. We adopt this perspective and we even claim that DSLs are a means to accomplishing the vision of MDE. In line with this perspective, the central *language aspect* of a DSL is a *metamodel* (descri-

bing the structure of the DSL through concepts and relations among these concepts), that is complemented by auxiliary language aspects, e.g., editor, interpreter, code generator, model transformations, and constraints. The metamodel is the central language aspect because all auxiliary language aspects use various techniques that in the end boil down to operations that make use of the metamodel (they navigate and query the metamodel). Figure 2.5 illustrates this idea. To make a parallel to the grammar-based world, the metamodel is the grammar of the DSL, with the difference that the metamodel does not prescribe a concrete syntax and it contains more semantic information. Moreover, in MDE terminology, the program (which need not be executable) written using the DSL, is called a *model expressed in the DSL*.

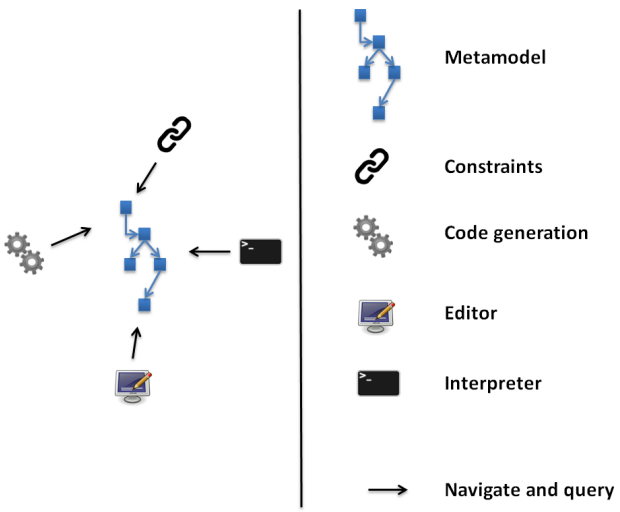


Figure 2.5: Language aspects and the central role of the metamodel aspect (icons from Clip Art library in Microsoft Powerpoint 2010).

External versus internal DSLs There are two main types of DSLs: external and internal DSLs.

Internal DSLs are created inside a *host language*. Interoperability among internal DSLs is more easily achieved (because they are using resources of the same host language), although external languages designed on the same platform (such as in MPS) can also interoperate with ease. The concrete syntax for these DSLs is usually restricted to what the host language allows. Although the DSL can rely on the tools provided for the host language, these are not custom made for the domain of the DSL.

External DSLs usually need to go through the entire specter of language creation, e.g. scanning and parsing (these two are eliminated with projectional editors), and creation of the compiler and interpreter. Moreover, integrated development environments (IDEs) are also considered part of the programming experience, and it thus becomes important to provide IDE support for new languages. Modern language workbenches offer facilities to build IDE support for the new DSLs; for instance, DSLs built with Xtext are supported in Eclipse, and DSLs built with Jetbrains MPS are supported in IntelliJ IDEA. Although the DSL engineer has the freedom to define all these elements with an external DSL, this comes at higher costs than those of implementing an internal DSL. It is external DSLs

that we focus on in this thesis, and thus, our following discussions refer to external DSLs.

Roles There are also two main roles played by developers interacting with DSLs. There are the DSL engineers and the DSL users. The DSL engineers are language engineers and they implement the language aspects in collaboration with the domain experts. The DSL users are the users writing models of the DSL and they should ideally be domain experts. In some organizations, there is also a third role, that of DSL integrators. The DSL integrators take the models from the DSL users and process these models further through model transformations provided by the DSL engineers. In the remainder of the thesis, we will only refer to DSL engineers and DSL users.

Providing meaning to DSLs The most common ways to provide meaning to DSLs are the interpretative way (navigate the models and execute code) and the code generation way, or, more generally, the model transformation way (transform the models to an implementation in a different language). These are similar to general-purpose programming language techniques, interpretation, and compilation. Both the interpretative and the generative ways are encoded in the processing units. Moreover, the processing units are also statically type-checked (see Section 5.2.2).

Definition and application of processing units To get a better understanding of the processing units and their interplay with the metamodel and model levels, one should consider the levels where the definitions and applications of the processing units occur. The definition of a processing unit occurs at the metamodel level. The application of a processing unit, on the other hand, happens at the model level. That means that the operations in a processing unit are written generically on metamodel types, and at the model level, these operations are applied to instances of the types.

Benefits There are many benefits associated to using DSLs [157]. The size of the DSL model the DSL user needs to write is presumably smaller than that of the generated code, thus increasing productivity [145]. Moreover, because of the abstractions used by the DSL, more meaningful validation and verification can be performed on a DSL model. DSLs also increase the quality of the model because they only contain the necessary constructs and repetitive work is automated in the generator. Furthermore, models should be more long lasting than the code, because although the code might become obsolete (consider a change of platforms, for instance), models are still relevant. Models can also be used as a designing and communication tool, and they can involve the domain experts directly. From these benefits, one can notice the first-class nature of models.

Another benefit of domain-specific languages is that they protect the investment in the domain [31], in that they do not depend on the language. Think, for instance, of domain-specific libraries that need a different implementation for each general purpose language, while in theory, a DSL can be combined with other DSLs.

Challenges There are also challenges associated to using DSLs [157]. The first and foremost is the effort of building a DSL (we refer only to external DSLs from now on) and it requires language engineering skills from the software developers. Moreover, other challenges might come from the fact that the persons building the DSLs and those using them are different, and that DSLs evolve and need to be maintained. Furthermore, there is the danger that people keep using the DSLs just because they invested effort and time

in them, and they are not as easily willing to change to an alternative. One other critic that we hear often about DSLs developed in house for various projects is that they require new employees to learn yet another language. Our view on this is that when the language is designed well, this should not be a big concern. When someone starts working in a new place, one first needs to get acquainted with the domain itself, and a well designed DSL can be part of the process to get acquainted with the domain. Learning how to write programs in the DSL is learning how to speak the language of the domain.

2.4 Language workbenches - Jetbrains MPS

In discussions of how to implement domain-specific languages, one can not ignore language workbenches. In this section, we define language workbenches, and we showcase its characteristics with the help of Jetbrains MPS.

A *language workbench* is an environment that provides complete support for the DSL engineers to define a DSL (to implement all language aspects) with one or several meta-languages; it is also an environment for the DSL users to apply the DSLs. A meta-language is a language that contains dedicated constructs for implementing a particular language aspect of a DSL (constraints, code generation, interpretation, etc.). A meta-tool, on the other hand, is a tool that embodies a meta-language and that takes care of auxiliary concerns as well, such as persisting the meta-language and its models, provide infrastructure for code generation, etc. Thus, a language workbench can be viewed as a collection of meta-languages and meta-tools, that together allow DSL engineers to define DSLs and DSL users to apply the DSLs.

A particular type of language workbench is a *projectional* language workbench, which makes use of projectional editors. In a projectional editor, the user is always directly manipulating the representation of the projected data (object graph). This is in contrast to parser-based editors, where the user is manipulating text that is first parsed; the parser outputs the representation of the data (if the text is a member of the language). Projectional editors are not a new technology. They have been around since at least the 1980s, with Incremental Programming Environment [98] and the Gandalf project [109]. These editors had usability problems, especially when typing arithmetic expressions [160]. For instance, introducing $1 + 2$ meant introducing first the + sign, and then filling in the left-hand side and the right-hand side arguments. Newer projectional editors, such as those in Jetbrains MPS [65] and Intentional Domain Workbench [63], have succeeded in bypassing these problems [160].

To get a better understanding of language workbenches, we present the definition given by the person who coined this term, Martin Fowler [51]. He proposed the following characteristics to language workbenches:

1. *Users can freely define new languages which are fully integrated with each other.* This first point gets us to the ease of creating DSLs and to the composition of DSLs.

2. *The primary source of information is a persistent abstract representation.*

3. *Language designers define a DSL in three main parts: schema, editor(s), and generator(s).* The schema represents the metamodel in our terminology.

4. *Language users manipulate a DSL through a projectional editor.* Although this characteristic exists in the original definition, it is not a defining characteristic

of language workbenches [157]. There are language workbenches that use parser technology.

5. *A language workbench can persist incomplete or contradictory information in its abstract representation.* This is easy with parser-based technology, but more challenging with projectional editing.

We illustrate the characteristics of a language workbench on Jetbrains MPS. We often use MPS in the thesis as a comparison, so this description also serves as a reference. Note that each language workbench usually uses its own terminology for the DSL terms introduced in this chapter. Such tool-specific terminology will be marked with a superscript in the case of MPS.

Jetbrains MPS [65] is a projectional language workbench that was designed from the beginning to accomplish the vision of language-oriented programming [38]. In this thesis, we mostly refer to the latest available version of MPS at the time of writing, version 3.4. When this is not the case, we make it explicit.

In MPS, a DSL definition resides in a *language module*^{MPS} and a DSL program resides in a *model*^{MPS}. In a language module, there are several language aspects that a DSL engineer needs to implement. The basic language aspects are structure (the metamodel), constraints (the static semantics), editor (the concrete syntax), generator (the model transformations), behavior (common operations for the concept), and type system. There are also more advanced language aspects, such as data flow or intentions (they smooth out the user experience). Most of these language aspects need to be implemented per concept. Thus, a DSL definition in MPS boils down to implementing the necessary language aspects (defining operations) for each concept. One can notice that the three main parts of defining a DSL exist in MPS (as stated by *characteristic number three*), but these main parts are complemented by many other language aspects.

As it is the case with many other language workbenches, each language aspect can be created using a DSL designed specifically for that aspect in MPS. For instance, Figure 2.6 shows a model written in the DSL for editors. The editor consists of cells that contain information from the model or constant information.

```
<default> editor for concept Canvas
  node cell layout:
    [-
    Painting { name }
    (/ % shapes % /)
      /empty cell: <default>
    -]

  inspected cell layout:
    <choose cell model>
```

Figure 2.6: The editor for concept *Canvas* that contains a vertical collection consisting of a line of cells formed of constant cell *Painting* and a cell with the name property of the canvas, and of a list of editors for the contained shapes on the next line.

All the features of MPS described so far relate to *characteristic number one* of language workbenches stating that one can freely define new languages.

Moreover, each concept in MPS is persisted in an XML-based format, or a custom format defined by the DSL engineer. That is why the models can not be rendered in any other editor, unless those editors know how to interpret the format. In MPS, the

models can be projected on screen using one of the available projections defined for them. These features relate to *characteristic number two and four*, on persisting the abstract representation of the models and projecting that abstract representation to the users.

The next paragraphs are related to *characteristic number one* and the integration of languages.

There are two main ways to reuse DSLs in MPS. The default one is where one *uses*^{MPS} DSL B in DSL A. This means that one can incorporate concepts that exist in DSL B in DSL A as they are. The definition of concepts in B can not be modified. Thus, this represents a reuse by reference. The more advanced reuse mechanism is the one where DSL A *extends*^{MPS} DSL B. This means that one can extend any concept that exists in DSL B in DSL A. This results in the creation of a new concept that inherits all the properties and behavior of the extended concept. This also allows the customization of most inherited language aspects in the extending concept. The extension mechanism is also the most used mechanism in mbeddr [158]. That is mostly because one does not reuse a DSL as it is, but usually needs to make conservative modifications that cater to the needs of the reusing context.

Moreover, different language aspects have different extension mechanisms. We are briefly going to introduce the most common ones. In MPS, the metamodel of a language is captured in a language aspect called structure. The extension of concepts in MPS is similar to extension in Java. The extending concept inherits all the properties, children and references of the extended concept (these represent the relations between concepts), with the possibility to specialize the type of the references. The constraints can also be overridden in MPS, with the exception that one can not call the constraints implementation of the parent concept, as one can do with the keyword 'super' in Java. In the behavior aspect, on the other hand, only the *virtual methods*^{MPS} can be overridden. Editors can be overridden as well, totally or partially through the *editor components*^{MPS}.

Two language aspects that have special extension mechanisms are the generator and the type system. A *generator*^{MPS} is implemented with a set of *generator rules*^{MPS}, that are subsequently applied on the models of the DSLs. When combining languages, one can create a generation plan with the help of priorities among each two generators. For instance, when combining language A with generator *GenA* and language B with generator *GenB*, one can specify whether the generator rules of *GenA* should be applied before, at the same time, or after the generator rules of *GenB*. The type system, on the other hand, uses declarative rules. One defines a typing equation per concept, which is subsequently used by the type solver engine. A language extension can simply add new typing equations; in cases of conflict, children concepts can override parent typing equations.

There are also so-called *extension points*^{MPS} in MPS. This mechanism allows one to write an abstract class as an extension point that can be implemented by different extending languages, and one of these extending languages' implementations can be chosen by the DSL engineer to take effect.

Because of its projectional nature, MPS is not as good at persisting incomplete or incorrect information in its abstract representation, as required by *characteristic number five*. Nonetheless, it can do that to a certain extent, as can be seen in Schindler et al. [123]. A node, for example an *if* statement, must have a complete skeleton. It is possible to leave content out, such as the guard and body in an *if* statement. The result can be considered syntactically incorrect since the guard is missing from the *if* statement. However, the construction is still structurally sound since it is a valid tree node, albeit with some gaps to be filled in. Figure 2.7 shows an example of a function with omitted name containing

an *if* statement missing the guard.

```
public void <no name>() {
  if (<condition>) {
    <no statements>
  }
}
```

Figure 2.7: An incomplete function with missing name, containing an *if* statement missing a guard.

Chapter 3

Language workbench requirements for modularity and reuse

The overall quality of a software system can be viewed as a trade-off between different goals that the software system needs to accomplish when completed. It is this view that we adopt on the overall quality of a DSL. The overall quality of a DSL is a combination of various qualities that each serve one goal only. Most DSL qualities we discuss are a paraphrasing of qualities defined for software systems. At times, there are subtle differences between qualities for software systems and those for DSLs. In this chapter, we first describe the two DSL qualities that are the theme of this thesis, modularity and reuse. Then, we briefly review other DSL qualities and how they are affected by the two main ones. Finally, based on the two main DSL qualities and inspired by criteria for extending software systems in general-purpose programming languages, we describe language workbench requirements for the creation of modular and reusable DSLs.

3.1 Modularity

The first DSL quality we tackle is modularity. We first discuss modularity in general terms, and then we move on to discuss it in the context of software development in particular. Finally, we consider modularity of DSLs and also our goals with modularity of DSLs. Thus, we also partially answer research question RQ_1 in this section.

> **RQ_1:** *What are reasons and requirements for modularity and reuse in language workbenches?*

The desire that systems possess modularity is often motivated by complexity. There is no consensus among scientists on a definition of complexity, but it is usually characterized by an involvement of many parts, aspects, details, and notions requiring deep studying or examination to grasp fully [9]. Thus, the kind of complexity we refer to is complexity as perceived by humans [167]. In a famous study in cognitive psychology, Miller [106] argued that the number of objects that an average human can hold in working memory is seven

plus or minus two. In a later study, this number was deemed even smaller by Cowan [29]. He argued that the number of chunks that a young adult can hold in working memory is around four.

Humans have the same cognitive limitations when studying the elements and the relationships between these elements in models. This problem is exacerbated by the larger and more complex models [81] that stem from applying MDE to larger and more complex systems and domains. In order to understand complex models, humans need mechanisms to organize those models and the mechanisms could come from modularity. The definition of modularity itself is linked to complexity by some authors. For example, Garud et al. [54] define modularity as an efficient strategy to organize complex products and processes.

Other definitions of modularity rely on the definition of its building blocks, the modules (or units, pieces, subsystems, components) [55]. A definition from software design considers modularity as 'tools for the user to build large programs out of pieces' [22] with the purpose to encapsulate what can change, or the 'building of a complex product or process from smaller subsystems that can be designed independently yet function together as a whole' [10].

Some authors refrain from giving an explicit definition of modularity and instead look at it from multiple viewpoints, giving criteria, rules, and principles for modularity [103].

In dealing with complexity, modularity has proven to be a good ally for engineers in all kinds of fields. The alleged benefits of modularity from the point of view of product design (not necessarily software products) are numerous [55]. Modularity promotes interchangeability (welcoming experimentation); economies of scale due to the use of components across product families; ease of product updating and maintenance; ease of design and testing due to decoupling, parallel development or comprehensibility; and manageable complexity [11, 55]. Furthermore, regarding the other main quality targeted in this thesis, modularity also encourages code reuse. This is not automatic; one can have modularity without actually having practically reusable components.

There are also downsides to modularity, especially when over-modularizing. Often, it is not clear how to achieve the right modularization, and it is essential that one avoids not only under-modularization (that leads to difficult maintenance), but also over-modularization (that leads to many relationships to manage and hinders understandability) [97]. Thus, paradoxically, modularity itself, if done wrong, can lead to an increase in complexity. Moreover, modularization often requires more thought up-front during the design and has to be planned systematically. Modularization also increases total size, because there is overhead in interfaces and coupling.

Next, we look at modularity in software development. At this level, modularity is usually captured in the formalisms, the programming languages or the tools.

Modularity in the software development literature Modularity in computer programming languages has a rich history. We look at three influential studies on this subject in the literature.

At first, modular programming was associated with an assembly of programs from subroutines [77, 167]. Yourdon et al. [167] talk about cohesion and coupling in modular modules. Coupling means the degree of interdependence between modules and cohesion means the degree to which elements of a module belong to that module. Interestingly, coupling and cohesion are interrelated; even more, the two measures are correlated (as one increases, the other decreases). The authors define good systems as systems that have low coupling and high cohesion. Meyer [103] notes that the discussion of module

modularity around coupling and cohesion in this case is limited by the scope of structured programming and its focus on subroutines.

Modularity is also related to the concept of 'separation of concerns', that was defined by Dijkstra [36], and it underlines the fact that "one tries to deal with the difficulties, the obligations, the desires, and the constraints one by one". Concerns are captured in software through language and tool mechanisms (classes and extension of classes in OOP, for instance) [142].

Later, Meyer [103] brought a set of criteria, rules, and principles that guarantee modules to be self-contained and organized in a stable architecture. He considers a design methodology to be modular if it respects the following five criteria: decomposability, composability, understandability, continuity, and protection. Decomposability refers to the property of being able to decompose a software problem into a number of smaller problems connected by a simple structure and that can be extended in the future independently of each other. Composability refers to the property of being able to compose software elements in order to produce new systems. Understandability means that a module can be understood independently, without having to inspect other modules or by having to inspect only a few of them. Continuity refers to the fact that a small change in the specification of a software triggers changes in a small number of modules. Protection refers to the fact that an abnormal condition in a module at runtime propagates to a small number of other modules. From these criteria, five rules follow: direct mapping (structure in software maps onto the structure in the domain), few interfaces (low coupling), small interfaces (there is not much information communicated among two coupled modules), explicit interfaces, and information hiding. Furthermore, five principles follow from these rules: linguistic modular units (units correspond to syntactic units in the language used), self-documentation (information about the module is part of the module itself), uniform access (all services of a unit should be accessible through a uniform notation that does not reveal implementation details), open-closed (modules are opened for extension, but closed for modification), and single choice (when the software system supports a list of variants, one and only one module knows about the exhaustive list of variants). Note that "closed for modification" in the previous sentence means that the reusing entity can not modify the original source of the reused unit from the reusing context.

Why modularity of DSLs in particular? Coming back to DSLs, we first state the reasons for modularity of DSLs. DSLs are usually designed to solve a clearly defined task, and it is thus essential to be able to combine several DSLs to capture all the information of an application [84]. A big, monolithic DSL encompassing all these separate DSLs would clearly deteriorate the overall quality of the DSL, it would be less maintainable and it would be harder to evolve. For instance, mbeddr [158] is an example of a non-trivial collection of integrated languages, that consists of 81 different languages. Creating one monolithic language encompassing these 81 languages would probably be unmanageable.

Modular language implementations facilitate language evolution because modular implementations ease the process of changing a DSL implementation and its related tools [12]. This happens in the evolution process of a DSL. One difference between DSLs and GPLs is that DSLs evolve at a faster pace [157]. This happens because the process of getting a DSL in good shape can be an exploratory process. Moreover, the user is most of the times accessible locally, so implementers get feedback from the users immediately and they can change the DSLs accordingly.

The need for extensibility in general programming languages (adding new features to a programming language) also offers an insight into why modularity of DSLs is

important [165]. There are many reasons one would want to extend a language: security, static checking, language design, optimization, style, or teaching [110].

The goal for modularity A paper from 1968 [28] emphasizes the fact that identification of modules, contents, and interconnections has to be consciously designed with explicit system goals in mind. More recently, Don Batory [13] says that the most important aspect of modularity is its goal.

In most cases, the main goal for modularity in our explorations is the reuse of the modularized DSL units[1], but we often also have understandability as a goal. Note that modularity of DSL units means modularity of both the metamodel and of the processing units.

3.2 Reuse

The second DSL quality that we cover is reuse. We discuss it in general terms and in the context of software development, in particular. In the end, we state why we need reuse of DSLs, thus contributing to research question RQ_1.

> **RQ_1:** *What are reasons and requirements for modularity and reuse in language workbenches?*

Before we start the discussions, note that modularity and reuse are closely linked. There is an interplay between the two. Firstly, modularity helps reuse. One monolithic component is clearly less reusable than smaller components. Furthermore, Meyer [103] considers that modularity encompasses extendibility and reuse. Secondly, most of the times, reuse also implies adaptability of the reused artifact to some degree, which leads to extendibility, which further leads to modularity [103].

Software reuse is interwoven with the birth of software engineering. The NATO software engineering conference in 1968 is considered by many the birthplace of the software engineering discipline [85]. Software reuse was proposed at this conference as a means to solve the software crisis, the problem of building large software systems in a controlled and reliable way. Software reuse is a way of building software that entails reusing existing artifacts instead of building everything from scratch or copy-and-pasting artifacts. There are multiple ways to abstract, select, specialize, and integrate the reused artifacts [85].

There are a handful of expected benefits [103] associated to reuse. One might expect improvements in speed of time-to-market (one has less software to develop), decreased maintenance effort (someone else is responsible for the evolution of the reused component), reliability (if reused components come from a reputed source), efficiency (if components are developed by experts in the field), consistency (style of the component influences the style of the developed software), and investment (components are a way of preserving the know-how).

However, reuse comes with its own challenges. Reuse adds extra dependencies to the software system, which could result in maintainability issues [118]. Moreover, reuse implies the use of abstractions that could lead to more mental complexity. For instance, the templating mechanism in C++, that enables generic programming in C++, is a form

[1] A DSL unit is a DSL (metamodel and processing units); we use the word unit in association with DSL when we want to emphasize the fact that the DSL can be part of other DSLs.

of reusing algorithms for different types. Although a powerful form of reuse, templates are also more complex and, thus, harder to grasp by programmers. In addition, developing reusable artifacts is harder than developing non-reusable artifacts. It is often the case that a non-reusable artifact evolves (is refactored) into a reusable artifact.

Reuse in the software development literature We now explore reuse in general software development. We are not exhaustive in our exploration of the reuse techniques in software, but we touch on the most well known ones.

Probably one of the oldest forms of software reuse is copy and paste. This is an opportunistic type of reuse, because although it gives good results in terms of time to market, it can create maintainability problems (bugs have to be solved in multiple places, for instance). There are a multitude of works highlighting problems related to code duplication [41].

When it comes to software reuse, constructs of the programming language itself can promote reuse. One can think of constructs such as functions. They take an input through the *in* parameters, make some computation and produce an output through the *out* parameters. Calling this function more than once is equivalent to reusing the computation more than once.

There are also more systematic ways to reuse software. A systematic form of reuse consists, for instance, of software libraries. A software library allows users to reuse functions from the library. On the same line are software frameworks [133]. A software framework allows users to reuse a set of classes and to override methods of those classes. Yet another form of reuse are design patterns [53]. They are more abstract, in the sense that design patterns are not directly transformable to code. Design patterns are a description of how to solve a commonly occurring problem in software design.

More advanced forms of reuse were facilitated by OOP via classes, fields, methods, generics, etc.; explicit design is needed to ensure that some of these artifacts are reusable. A class incorporates a set of fields and methods that can be reused by any object instantiating the class. Moreover, the inheritance mechanism in OOP itself is a reuse mechanism, because of the reuse and adaptation of fields and methods of the super-class in the sub-class. The organization of classes into packages and jar files is again another way of reuse [58].

Two other forms of software reuse are generic programming and code generators. Generic programming allows reuse through the use of generic types in the definition of an algorithm, types that are to be instantiated later via parameters [31]. Thus, one algorithm can be reused with different specific types. A code generator, on the other hand, is a program that takes a specification of a piece of software and generates its implementation [31].

DSLs themselves are a form of reuse. A DSL captures domain expert knowledge in the metamodels and the generators. This knowledge is reused every time a DSL user creates models of the DSL and generates the executable code.

Why reuse of DSLs in particular? Although DSLs themselves are a form of reuse (as argued in the previous paragraph), our main interest in this thesis is the reuse of the DSL implementation itself, the metamodel and the processing units, at the level of the DSL engineer. Tangentially, we also touch on the reuse of value models at the level of the DSL user.

As discussed in Section 2.2, to accomplish the vision of LOP we need mechanisms to more easily create DSLs. One way of creating DSLs easier, is by reusing parts of other

DSLs in the development of the current DSL.

Given that implementing DSLs is a particular form of software development, by extension, all benefits and challenges from software reuse are valid for DSL implementations as well (this argument also holds for the modularity of DSLs).

3.3 Other qualities

In this section, we discuss a few other DSL qualities that are not part of the main qualities that we target in the thesis, modularity and reuse, but that are influenced by these two. The influences can be both positive (an increase in one quality leads to an increase in the other), or negative (an increase in one quality leads to a decrease in the other quality). Most of the following qualities are paraphrasing of software qualities as discussed by Meyer [103], except for the last two qualities, expressiveness and learning curve. We argue that these two qualities are particularly important for DSLs.

3.3.1 Correctness and robustness

Correctness is the ability of a DSL to function according to its specification. Specification in this case encompasses two elements: the domain specific tasks that the DSL should encode and the semantics of these tasks. The specification thus answers these two questions: what are the tasks that the DSL should be able to express and what is the meaning of each of the constructs encoding a task? If a DSL does not do what is supposed to do, then none of the other qualities matter. Thus, being correct comes first. On the other hand, enforcing correctness is hard, because the specification can be unclear; deciding on the most essential collection of tasks that the DSL should handle can be challenging. One could try correct-by-construction techniques, or rely on testing and debugging facilities [103].

Robustness is the ability of a DSL to handle abnormal conditions. The specification tells how a DSL is supposed to behave under normal conditions, that is, when the DSL users introduce correct DSL models. Besides that, it should also give appropriate error messages in case of abnormal conditions. Robustness thus complements correctness [103].

The correctness and robustness of the DSL is linked to the DSL models, because it is with the DSL models that the DSL can be tested: can we express a particular solution in the DSL and do we obtain the expected results from the DSL? The DSL engineers can enforce correctness and robustness by adding type systems, extra semantic checking rules, or data flow analysis checks to their DSLs. For instance, Jetbrains MPS allows DSL engineers to create type systems, checking rules and data flow analysis for their DSLs [65].

Correctness and robustness are affected by modularity and reuse as well. In order to obtain a correct and robust DSL, one needs to pay attention to the semantics of the composition of different DSL units. Moreover, when we reuse DSL units from other sources, those DSL units need to be correct and robust themselves, to ensure that the reusing DSL unit is correct and robust.

3.3.2 Performance

Performance is the ability of a DSL to be efficient in terms of processor time used, memory footprint and bandwidth used in communication devices [103]. For DSLs, in particular, this translates to how efficient is the generation of code from the processing units and how efficient is the generated code itself.

Both modularity and reuse affect performance. The effects can be both negative and positive. Looking at the efficiency of generating code, for instance, managing the combination of separate DSL units is more involved than managing one unit that contains everything. On the other hand, one can benefit from modularity through generating code separately for each processing unit and through making incremental generation (only generate the code of those processing units that are modified, or the processing units with modifications in the associated metamodels since the last generation).

3.3.3 Understandability and usability

Understandability is the ability of a DSL to be grasped by DSL engineers with an acceptable amount of effort. Understandability is inevitably linked to humans. Humans are the ones that benefit or are hindered by a hardly understandable DSL. This quality relates to the human cognitive limitation discussed in Section 3.1. How easy is it for a human to understand the metamodels, and the processing units?

Modularity, in particular, should have a positive impact on understandability. Dividing the information into multiple units and combinations that are coherent and cohesive helps humans better grasp the units and their combination.

Usability, on the other hand, is the ease with which a DSL engineer can manipulate the different DSL aspects. Usability is influenced by the maturity of the language workbench.

3.3.4 Expressiveness

The degree to which abstractions in a DSL are intuitive for and related to the domain of operation defines expressiveness. Having all the abstractions to express solutions in a particular domain, but not more, defines an expressive and focused DSL.

If a DSL is not expressive enough, users are reluctant to use it. Making a DSL expressive enough can be an iterative process. DSL engineers can receive feedback from the DSL users when it comes to adding missing functionality to their DSLs (or even remove unused functionality). They could even observe how DSL users make use of their DSLs, and they could abstract common patterns of usage into new abstractions in the DSL. There is also another side to expressiveness. Although abstractions in a language allow expressing a certain solution, the abstractions could be too complicated for the majority of the DSL users, thus restricting the number of users. As a result, expressiveness could influence understandability and usability in a negative way.

3.3.5 Learning curve

The amount of time one needs to get acquainted to a DSL is an important aspect for DSLs in terms of adoption. Because of the small domain of operation of a DSL, it is essential that the DSL is easy to get started with and easy to get productive with.

Tools play an important role in this quality aspect. Tools could offer helping pointers for DSL users getting acquainted with the DSL: sample models written in the DSLs, tool-tips, common usage scenarios, etc.

Understandability, usability and expressiveness have an influence on the learning curve. The more understandable, usable and expressive a DSL, the easier it is to become productive with it. Moreover, performance also influences the learning curve. If a user needs to wait long for processing results, that can lead to a significant increase in the learning time. Furthermore, modularity also influences the learning curve because people

can approach understanding of sample models by first looking at smaller units and then at the combinations.

3.3.6 Discussion on DSL qualities

Some qualities described in this chapter are governed by contradicting forces. One needs to make compromises and prioritize the qualities that their DSL needs to posses. All the qualities can affect each other both positively and negatively.

3.4 Language workbench requirements for modularity and reuse

The qualities that we mainly focused on during this research are modularity and reuse of DSLs, mostly on the DSL metamodel and processing unit levels. Based on these qualities and based on criteria for such qualities in software development, we explored requirements that language workbenches need in order to facilitate the creation of modular and reusable DSLs. Thus, we contribute to research question RQ_1 in this section.

> **RQ_1:** *What are reasons and requirements for modularity and reuse in language workbenches?*

Both modularity and reuse are strongly linked to extensibility. The original software and the extended software are two separate units (modularity), while the extension reuses the original software (reusability). Extensibility, on the other hand, is linked to software evolution. Software evolves over time, which makes it a prerequisite for software to be extendible. This is hard because of the so-called *expression problem* that manifests when extending software systems [168]. The expression problem occurs when the software developer is not able to extend datatypes with new data variants and, at the same time, to add operations to the datatypes. A solution to the expression problem needs to fulfill certain criteria, such as non-duplication of the original code or strong static-typing of the solution. A variant introduced by Zenger et al. [168] called the extended expression problem, introduces one more criterion that says that independent extensions should work together, because this last criterion would allow programmers to make extensions in a non-linear fashion. As a result of the limitations of the mainstream programming languages regarding the expression problem, developers often need to implement complex designs that anticipate various dimensions of variability [60].

In software systems, the kind of solutions that a programming language offers to the expression problem are seen as an indicator of the expressive power of a language. That is why we have adapted the extended expression problem criteria to language workbenches.

The requirements for language workbenches that we are about to formulate are a paraphrasing of the extended expression problem criteria in terms of domain-specific languages. Because the development and application (which involves creation of models, and transformations, code generation, etc.) of domain-specific languages can be seen as a special case of software development, the criteria for a solution to the extended expression problem in language workbenches is also a special case of the original criteria.

Below, we present five requirements for language workbenches that focus exclusively on modularity and reuse of DSLs. Language workbenches are tightly linked to the meta-languages used to implement the DSLs; so, most of the requirements will reference

the meta-languages. We now state these five requirements, and then we will discuss each requirement in more detail.

1. When reusing a DSL unit, the meta-language allows a DSL engineer to add new elements (concepts and relations) and new processing operations on reused concepts to the reused version of that DSL unit.

2. The meta-language offers strong static type checking of the DSL implementations: no processing operation is applied on elements that it cannot handle.

3. The meta-language should be such that reusing a DSL unit does not require to modify or to duplicate the reused DSL unit.

4. The meta-language should be such that if you have a type-checked DSL unit for which code was generated, then reusing that DSL unit should not require type checking it again, and regenerating the code.

5. The meta-language allows the combination of independently developed DSL extensions.

The *first requirement* states that extensibility should be possible for both metamodel elements and operations on reused concepts. This requirement caters, on one hand, for the extensibility of the metamodel, and, on the other hand, for the extensibility of the processing units. The additions proposed by this requirement are useful not only for the extensibility of a DSL unit, but also for the general reuse of a DSL unit, where one needs to adapt the reused unit to the reusing context. One might notice that we have made a slight addition to the extended expression problem that was defined for extensible software. We consider the addition of both concepts and relations as new elements. In the definition of the extended expression problem, they only consider the addition of new data variants. In MetaMod, the equivalent of a data variant is a concept which is a subtype of another concept. The addition of a new data variant would mean the addition of a new concept that is a subtype of a reused concept. We decided to introduce also relations involving a reused concept (either as a source or as a target concept) so that the augmentation of the reused concepts is possible. It is often the case that the reused concepts need to adapt to fit in the new context, that of the reusing DSL (see example with *CContainer* in Section 5.1).

The next three requirements state that the additions suggested in the first requirement should satisfy some properties. Not any solution to extending the metamodel and the processing units qualifies as a good solution.

The *second requirement* states that DSL implementations should be statically type-safe. This mainly translates to enforcing that, at compile time, an operation of a processing unit is called on arguments that satisfy the type requirements of the operation definition. This offers more protection against errors to the DSL engineers and, thus, increases their confidence in the implementation.

The *third requirement* states that the reuse of a DSL unit should happen without the modification of the reused DSL unit or without its duplication. A solution that wouldn't satisfy this requirement, would be a very invasive solution. Modifying the original DSL unit for the purpose of the reusing DSL unit is a bad practice, because this might break the intentions of the original DSL unit and it might break other DSL units that depend on the original DSL unit. Duplicating the reused DSL unit in the reusing DSL unit, on the other hand, brings maintenance problems, because errors need to be fixed in multiple

places. Moreover, the code base increases because one has multiple copies of the same DSL unit.

The *fourth requirement* states that the reused DSL units should not need re-type checking or regeneration of code when reused in other DSL unit. That is, the reused DSL units and their generated code remain as they are from the point of view of the type checker and code generator, even when used in a new context. This decreases the generation time and the type-checking time as well (both are important for the DSL engineers when it comes to ease of development). We made a slight modification to this requirement, because we talk about generation of code, and not about compilation. That is because the DSLs need to generate code from the processing units first, before the generated code is compiled. Generation is accompanied, in the end, by compilation, because the generated code needs to be compiled.

The *fifth requirement* states that DSL extensions that were defined independently should be composable. This requirement ensures that DSL engineers can work separately on different extensions, in a non-linear fashion [168], and that they can combine the extensions in one extension. Combining two extensions can require some glue code; this is not forbidden by the criterion.

3.5 Conclusions

In this chapter, we analyzed, firstly, the two main qualities that are the focus of our thesis, modularity and reuse, in the realm of software engineering. We motivated the need for modularity and reuse in DSLs, and we mentioned how these two qualities translate to DSLs. Furthermore, we analyzed other qualities for DSLs and how they are influenced (positively or negatively) by the two main ones. Our contribution here is that, although these qualities were initially defined for software applications, we have modified them for DSLs (with slight differences at times). Moreover, based on these qualities and the expression problem from software engineering, we have defined requirements for language workbenches in regard to modularity and reuse of DSLs. Again, our contribution consists in paraphrasing the criteria for the expression problem in software engineering for language workbenches (with slight additions and modifications).

Chapter 4

MetaMod

There are two main reasons why we defined our own mechanisms and meta-tools for the design and implementation of DSLs in the form of MetaMod. First and foremost, we wanted to introduce elements of modularity and reuse starting from the core of a DSL, the metamodel, and ending with the processing units. We did not want to tackle the issue only outside of the modeling formalism itself. Secondly, to experiment with our ideas on modularity and reuse, we needed a simple metamodeling language. A full-scale metamodeling language, like MOF or even EMOF [112], contains redundant modeling elements [153], and an exploration of modularity mechanisms in such a context would be cluttered by the extra modeling elements. Thus, we had modularity, reuse and simplicity in our mind while designing MetaMod. These goals have guided many of the decisions we made in the construction of MetaMod. Simplicity, in particular, motivated the design of a minimalistic and uniform meta-metamodel. The mechanisms developed for MetaMod are captured in abstractions of meta-languages; these meta-languages are part of the meta-tools. In this chapter we introduce all meta-languages of MetaMod and, in the end, we showcase all the meta-languages on an example DSL.

4.1 Meta-metamodel

In this section we define the core component of MetaMod, the meta-metamodel (defined in Section 2.1.1). This component is described first, because all other components of MetaMod depend on it. Moreover, through describing this component, we give an answer to research questions RQ_2 and RQ_5.

> **RQ_2:** *How can we organize metamodels of the DSLs such that we facilitate modularity and reuse of DSLs?*
>
> **RQ_5:** *What modularity and reuse mechanisms can be applied to models, irrespective of the DSL?*

MetaMod has a multilevel nature, because of two reasons. Firstly, the meta-metamodel defines the structure for both type models and value models, viz. an instance of the

meta-metamodel can be either a type model or a value model. Secondly, the conformance relationship between a value model and a type model is captured in the meta-metamodel (this will be detailed in Section 4.1.1.1). Note the terminology we use; we say that type models and value models are *instances* of the meta-metamodel, while we say that a value model *conforms* to a type model. We chose this terminology to make clear the relationships between the meta-metamodel and the type or value models (instantiation), versus the relationships between the type and value models (conformance).

All elements of the meta-metamodel of MetaMod are captured in the diagram in Figure 4.1. The diagram is represented in the visual syntax of MetaMod; we give full details of the visual syntax in Section 4.1.5.2. At this point, the reader is not expected to understand the diagram in its entirety. We will now explain what the reader needs to understand from this diagram. Firstly, note that the definition of MetaMod in Figure 4.1 is circular, because it uses a subset of MetaMod to define MetaMod. It is like defining Lisp [166] by presenting a Lisp interpreter written in Lisp. Such definition is circular, but it is still considered useful [120]. Secondly, our implementation of MetaMod in MPS serves as the formal definition. But since this may be even less accessible to the reader, it suffices for now to have an understanding of Figure 4.1 based on MOF [112] (described in Section 2.1.2.1). The way in which the semantics of MetaMod diverges from MOF are not important at the moment. Thus, in MOF terms,

- the rectangles are classes (we call them concepts),
- the filled arrow lines are associations in MOF (we call them relations),
- and the open arrow lines are superclass relationships in MOF (we call them subtype relationships).

A relation in MetaMod has a direction so that one can distinguish the two ends of the relation: the source, represented by the straight end, and the target, represented by the filled arrow end. The name in the label of a relation represents the name of the relation. The cardinalities of the relation are represented at the left-hand side of the named label for the source and at the right-hand side of the named label for the target. At this point, one can notice that there exists a *subtype_of* relationship in the diagram, and that in the legend, the open arrow line itself is denoted as a *subtype_of* relationship. Again, that is because we describe MetaMod using MetaMod. For the moment, one can interpret visual constructs in the diagram as MOF terms (e.g., subtype relationship with superclass relationship). With the explanations in each of the next sections, MetaMod diagrams should become clearer.

All the elements in Figure 4.1 are explained in the next sections, where the definitions are given in an incremental fashion starting with the core elements, extending it with groups and then with fragment abstractions and applications. In the end, we present the elements that are introduced for implementation purposes, and that are not part of this diagram.

The definitions we present in this thesis are not mathematical in nature. The main reason for this is that mathematically rigorous definitions are less accessible than we desire. Note that the focus of this thesis is not on formal semantics of languages and related proofs. Therefore, we have chosen to employ natural language descriptions, examples, procedures, and algorithms to create the definitions. Do keep in mind that these definitions are also accompanied by the Jetbrains MPS implementation of MetaMod.

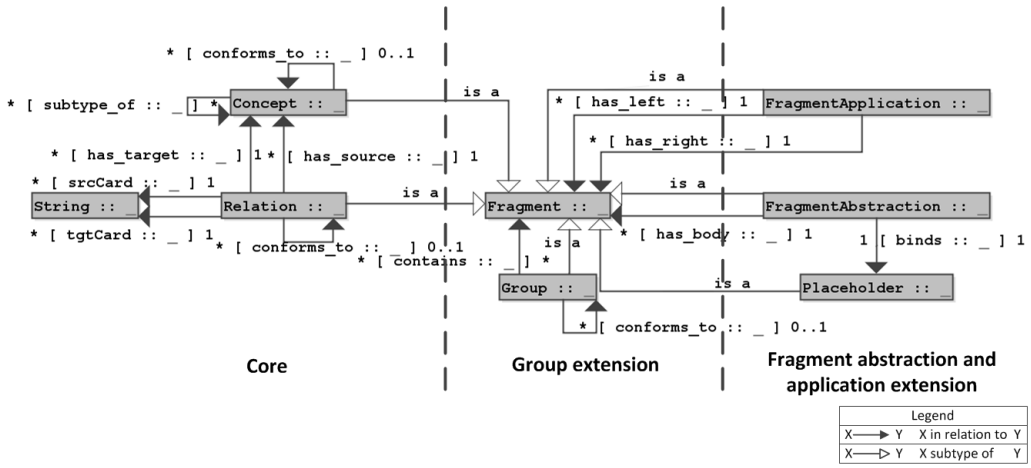


Figure 4.1: The meta-metamodel of MetaMod. This figure highlights the three parts of the meta-metamodel.

4.1.1 Core

The core of MetaMod contains the basic elements that can model a domain. More specifically, it is formed by concept and relation (see Figure 4.2).

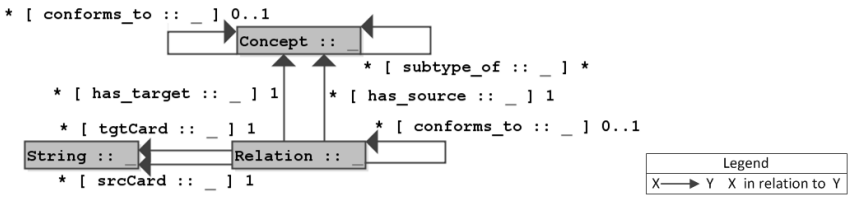


Figure 4.2: The core of the meta-metamodel of MetaMod.

Definition 1 *A **concept** denotes an entity of a model defined in MetaMod.*

Example 1 *For instance, if we define a type model to describe the railway infrastructure in a country, we use concepts such as* train station*,* rail segment*,* switch*,* road crossing*, etc.; we refer to such concepts as type concepts.*

Example 2 *If, on the other hand, we discuss value concepts in the railway infrastructure example, we use concepts such as* train station Amsterdam*,* train station Vienna*, etc.; we refer to such concepts as value concepts.*

A parallel can be drawn between type concepts in MetaMod and *classes* in MOF [112], and between value concepts in MetaMod and *elements* in MOF. One difference is that type concepts (unlike classes) do not have attributes or operation signatures. Attributes can be modeled with relations, so they can be dropped from a minimalistic meta-metamodel; remember that the minimalistic characteristic was motivated by the simplicity goal. Operations, on the other hand, are modeled separately from the type models for modularity

reasons. This way, the same type model can be used with different operations. Another difference is that concepts do not own any relations (this is unlike in MOF, where classes own associations). We made this decision for modularity reasons and we explain it in later sections. One last difference is that concepts do not have properties such as abstract, in order to preserve the minimalistic nature of MetaMod.

Definition 2 *The* ***relation***[1] *element is a relationship between a source concept and a target concept, with a source and target cardinality.*

Example 3 *If we get back to the railway infrastructure DSL, there is a relation called* direct_connection *between one* train station *and another* train station*; we refer to such relations as type relations. There can be multiple direct connections from one train station, so the cardinality of the* direct_connection *relation will be zero-to-many at the source. There can also be multiple direct connections to one train station, so the cardinality of the* direct_connection *relation will be zero-to-many at the target.*

Example 4 *If, on the other hand, we discuss about value relations in the railway infrastructure example, we say that there is a relation of type direct connection between* train station Amsterdam Central *and* train station Amsterdam South*; we refer to such relations as value relations.*

A parallel can be drawn between type relations in MetaMod and associations in MOF, and between value relations in MetaMod and links in MOF. For simplicity reasons, we drop the properties of the associations, such as ordered and unique; the ordered property can be modeled with extra relations in the type model, and the unique property can be achieved with constraints. On the meta-metamodel level, we do not define the relation element as part of the concept element, for modularity reasons (explained in detail in later sections). This is unlike associations that are part of a class and links that are part of an element.

Concepts and relations, together with the "has_source" and "has_target" relationships, form the bare minimum for modeling. All the other elements (subtyping, groups, fragment abstractions, etc.) are introduced for modularity or convenience reasons.

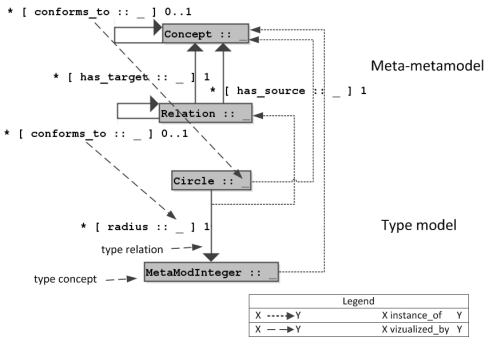
Now that we have introduced concept and relation, we can also give a better explanation of the meta-metamodel having as instances both type models and value models. We explain this with the help of a few figures. Figure 4.3a shows that a type relation and two type concepts are instances of elements *Relation* and *Concept* from the meta-metamodel. Similarly, Figure 4.3b shows that a value relation and two value concepts are instances of elements *Relation* and *Concept*. The explicit conformance relationships between type model elements and value model elements is then represented in Figure 4.4.

4.1.1.1 Conformance relationships (*conforms_to* relation)

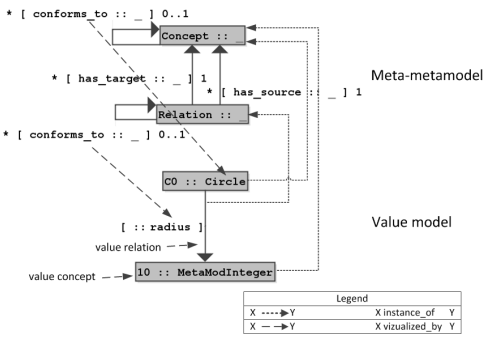
In this section we define the conformance relationship between a value model and a type model. The conformance relationship dictates the structural constraints of a value model. This relationship is defined in terms of concept conformance and relation conformance (see Figure 4.2 and the *conforms_to* relationships of *Relation* and of *Concept*).

In Figure 4.5 we depict separately two levels, the type model and the value model levels, that are encoded together in the meta-metamodel in Figure 4.1. Figure 4.5 makes

[1] To avoid confusion, relation is used when referring to the technical term introduced in the definition of MetaMod. A relationship is a generic association, that is not necessarily related to MetaMod elements.



(a) Visual representation of type model instances of the meta-metamodel.



(b) Visual representation of value model instances of the meta-metamodel.

Figure 4.3: Visual representation of instances of the meta-metamodel. All elements in the instances comply with the structure prescribed by the meta-metamodel.

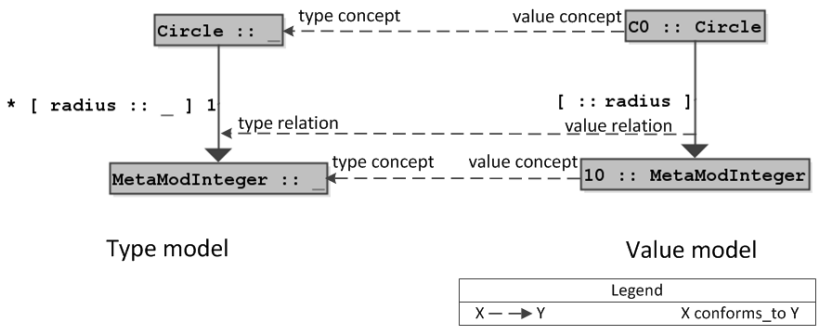


Figure 4.4: The conformance relationship between type model elements and value model elements represented explicitly.

it clear that the conformance relationships go from one level to the other. The relationship between these two levels with actual instances was shown in Figure 4.4. Note that at the value model level the *subtype_of* is used only if the value model also represents a type model. That is because the *subtype_of* relationship gets its meaning at the instance level.

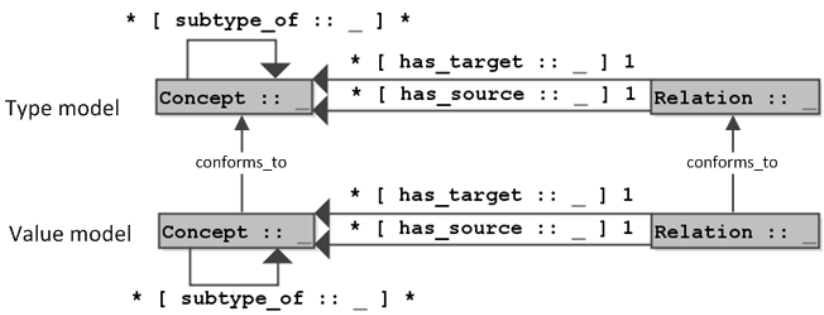


Figure 4.5: Two model levels and the conformance relationships among them.

The presence of the *conforms_to* association in the meta-metamodel and the fact that elements of the meta-metamodel describe both value models and type models, give rise to a multilevel modeling environment. Moreover, the conformance relationship can also give meaning to type models, by saying that the semantics of a type model can be expressed as the collection of the value models that conform to it.

In the following definitions, we assume that there exist conformance links among value concepts and type concepts, and between value relations and type relations. This is the default case for modeling tools, because when one introduces a concept or a relation in a value model, she also says what concept it conforms to, or what relation it conforms to, respectively. Moreover, we denote all super-concepts of concept X in zero or more steps by $allSuperConcepts(X)$. Zero steps means concept X, one step means all concepts in zero steps plus all direct super concepts of X, two steps means all concepts in one step and all direct super-concepts of the direct super-concepts of X, and so on. In a similar manner we denote all the sub-concepts of concept X in zero or more steps by $allSubConcepts(X)$. Finally, we also call a *direct relation* of a concept X a relation that has concept X as a source or a target explicitly in the model, and not through subtype relationships.

Definition 3 *A value model A conforms to a type model B if and only if all the concepts in model A conform to concepts in model B and all the relations in model A conform to relations in model B.*

Definition 3 is represented in Figure 4.6.

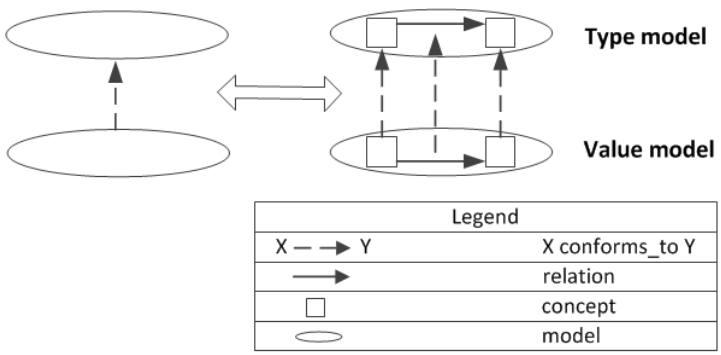


Figure 4.6: Visual representation of the definition of conformance between a value model and a type model, from Definition 3.

We represent conformance relationships with the *conforms_to* operator, '::'. For instance, concept CV conforms to concept CT is denoted as $CV :: CT$, and relation RV conforms to relation RT is denoted as $RV :: RT$. Once more, the definition in textual form might be hard to follow, so we also represent it visually in Figure 4.7. The textual definition of concept conformance is as follows:

Definition 4 *A concept CV conforms to a concept CT if and only if:*

1. *for all direct relations $RV :: RT$ where CV is a source concept in the value model, there exist direct relations RT with one of $allSuperConcepts(CT)$ as a source concept in the type model;*

2. *for all direct relations* $RV :: RT$ *where* CV *is a target concept in the value model, there exist direct relations* RT *with one of* $allSuperConcepts(CT)$ *as a target concept in the type model;*

3. *for all direct relations* RT *where one of* $allSuperConcepts(CT)$ *is a source concept with lower bound cardinality greater than zero in the type model, there exists a number at least equal to the lower bound cardinality, but not greater than the upper bound cardinality of relations* $RV :: RT$ *with* CV *as a source in the value model.*

4. *for all direct relations* RT *where one of* $allSuperConcepts(CT)$ *is a target concept with cardinality greater than zero in the type model, there exists a number at least equal to the lower bound cardinality, but not greater than the upper bound cardinality of relations* $RV :: RT$ *with* CV *as a target in the value model.*

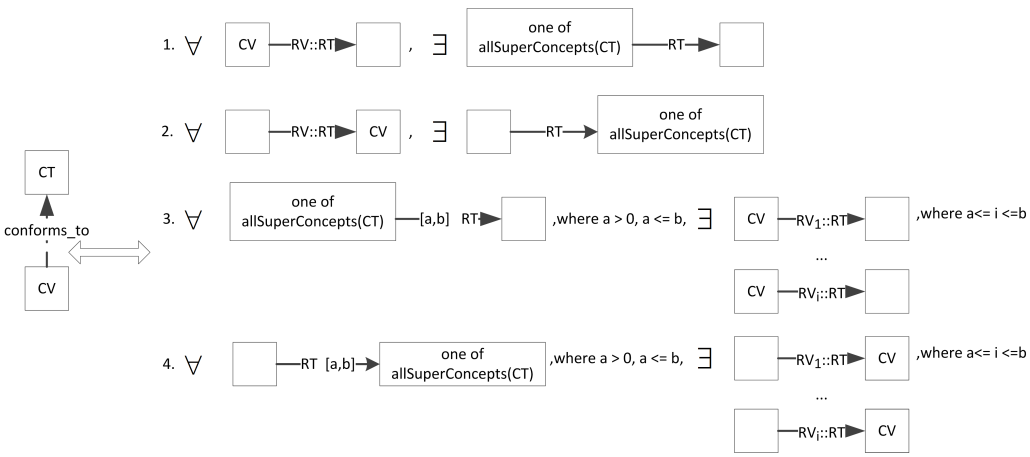


Figure 4.7: Visual representation of the definition of conformance between two concepts, from Definition 4.

The definition of relation conformance is represented visually in Figure 4.8. The textual definition is as follows:

Definition 5 *A relation* RV *conforms to a relation* RT *if and only if:*

1. *the source concept of* RV *conforms to one of* $allSubConcepts(\text{source concept of } RT)$*;*

2. *the target concept of* RV *conforms to one of* $allSubConcepts(\text{target concept of } RT)$*.*

Notice that the definition of conformance takes into account the subtype relationship. This way, the meaning of the subtype relationship can be inferred from the definition of conformance. Moreover, the definition of conformance takes into account cardinalities, and, again, the meaning of cardinalities can be inferred from this definition.

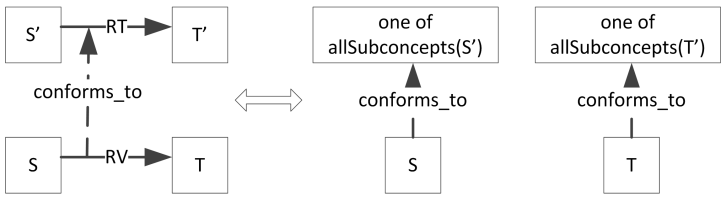


Figure 4.8: Visual representation of the definition of conformance between two relations, from Definition 5.

4.1.1.2 Subtype relationships (*subtype_of* relation)

In this section we define the subtype relationship, relationship that plays an important role in the type models. The subtype relationship is a relationship between a concept S (the source concept) and a concept T (the target concept). We call source concept S, the sub-concept, and the target concept T, the super-concept. The subtype relationship is introduced for reusability reasons, because this relationship allows the use of value concepts of type S in relations where value concepts of type T are expected. Thus, the subtype relationship gets its meaning and use at the instance level. Moreover, as we will show in Chapter 5, we leverage the subtype relationship in the implementation of the processing units.

The subtype relationship is similar to the superclass relationship in MOF. We have multiple inheritance in MetaMod, but we do not allow cyclic dependencies between concepts. This decision is in line with our simplicity goal.

Initially, we designed MetaMod without the subtyping relationship. We soon realized that the effort to create metamodels was daunting and error-prone (because we had to repeat relations that would only need to be defined for the super-concept with subtyping) and that we would not be able to create polymorphic operations based on concepts from the type models. These two concerns were serious enough to surpass the simplicity goal.

We chose to have only concepts and relations in the core because of the desire to design a minimalistic meta-metamodel that can still be as expressive as MOF (although not as accurate because of the absence of properties such as unique, ordered, abstract, etc.).

4.1.2 Group extension

At this point, with the elements from the core, we have concepts and relations that can be part of type or value models (or both) in MetaMod. With these elements, we can not split models, or we can not reuse other models. In the discussion in Chapter 3, we argued why we want a mechanism for splitting and reusing models.

Our work was an exploration of mechanisms for modularity and reuse. In this exploration, the central and simplest mechanism for modularity that we devised for MetaMod is a form of grouping. If we look at programming languages, we see that grouping is an essential aspect and it occurs often. One can think of object-oriented programming languages where there are many types of groupings: groupings of classes into packages, groupings of methods and fields into classes, groupings of statements into methods, groupings of variable declarations into parameter lists and so on.

We explain the group mechanism as an extension to the core of the meta-metamodel. The group extension and its additions to MetaMod are defined in Figure 4.9 and they are

explained in the next sections.

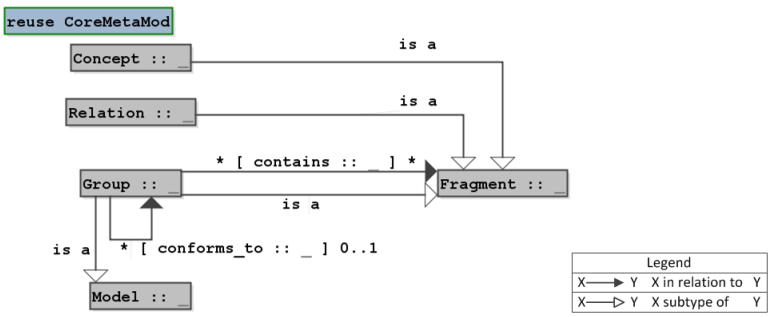


Figure 4.9: The group extension of the core meta-metamodel of MetaMod.

4.1.2.1 Model elements

Definition 6 *The **group** is the root modeling element of a model of MetaMod that contains other modeling elements (concepts, relations and other groups) as a unit.*

The contents of a group are captured with the *contains* relation from Figure 4.9. An extra constraint on this relation, and that cannot be captured in the diagram, is that a group cannot contain itself. Moreover, the group actually represents a type model or a value model. The notion of a group has been introduced for organizational purposes. We could have made this relationship part of the meta-metamodel (see Figure 4.10), but we decided not to do that because of the simplicity goal.

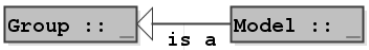


Figure 4.10: A value model or a type model is a group.

Properties One important property of groups regarding modularity is that *groups can share concepts*. That means the definition of a concept can reside in different groups. By the definition of a concept we mean the set of all relations where the concept is a source of. This helps in the definition of a complex concept (involved in many relations) and it also helps in the definition of a system with many parts, where the same concept is involved in more than one part of the system. For instance, in the Ecore metamodel description [135], concept *EClass* appears both in the kernel group and in the classifiers group. That is because *EClass* is a core concept in Ecore and it thus needs to be part of the kernel group, but it only gets a complete definition in the classifiers group.

Another important property of groups for modularity is that *groups can share relations*. There are cases where the same relation naturally lies in two groups. For instance, in the Ecore metamodel description [135], there are relations that appear both in the kernel group, where they discuss the central elements of Ecore, and also in other groups. For instance, classes have attributes in Ecore, and so there is a relation between the class element and the attribute element in Ecore. This relation appears both in the kernel group of Ecore, because classes with attributes are at the core, and in the structural features group, because attributes are structural features.

Moreover, *groups can reuse other groups*. That means that modeling elements inside of the reused groups are made available in the reusing group. Cyclic reuse among groups is not permitted. This design decision is in line with the simplicity goal by keeping modeling constructs easy to reason about.

The fact that groups that reuse other groups with sharing of concepts and relations is a natural way of describing domains, is proved by how various structures are described in official documents and books. In these documents, one can notice that groups are described in separation and they are also described in combination with other groups, with which they share concepts and relations. Among these descriptions, we mention the one of the Ecore metamodel in the Eclipse modeling framework book [135], the one of UML in the standard UML infrastructure and superstructure documents [114, 115], or the one of MOF in the core specification document [112]. Our meta-modeling facility with its modularity mechanisms can capture such descriptions the same as these documents (groups in separation and groups in combination with sharing of elements). Moreover, we follow the same structure when we describe MetaMod itself. We describe it in terms of three groups: the core group, the group extension, and the fragment abstraction extension.

Parallel If we draw a parallel with MOF, groups can be compared with packages. The difference is that groups can share relations and concepts. This sharing can, nonetheless, be obtained in MOF using two types of operations, package import and package merge [112].

Fragment Finally, the last element of the group extension is the fragment.

Definition 7 *The **fragment** is a unifying model element from which all modeling elements in MetaMod are inheriting.*

This model element is called a fragment, because every modeling element in MetaMod is a model fragment that can be used in other model fragments.

4.1.2.2 Semantics of groups

To better understand the grouping mechanism, we show a flattened view of a group, where its reused groups are dissolved. The flattening is possible because there are no cycles (groups, directly or indirectly, containing themselves). Unfortunately, this discussion will be interleaved with some implementation details as well, because the way we assign identities to elements in MetaMod is relevant during flattening.

The semantics of a group are given by how a group can be flattened. The flattening operation transforms a group into a group without reused groups, thus obtaining a flat collection of concepts and relations. From the resulting collection of concepts, duplicates are eliminated (as concepts with the same identity can be involved in different reused groups). The way we determine that two concepts have the same identity is encoded in Algorithm 2 in Section 4.1.4, the implementation section. We have outlined the algorithm in the implementation section because the equality between two concepts depends on how one chooses to identify a concept in the implementation. As for the resulting list of relations, only "exact same relation" duplicates (relations having the same name and the same source and target types) are eliminated. We do so because different combinations of sub-concept and concept sources and targets do not necessarily create completely overlapping relations; see Figure 4.11 and the different cardinalities of the *a_to_b* relations. The way we determine "exact same relations" is formalized in

Algorithm 3 in Section 4.1.4. From the list of "exact same relation" duplicates, we keep the relation that was introduced by the most recent group, or in case of a tie, the one defined in the group having the smallest lexicographical name (again, this is an implementation detail). The most recent group is the group that is in a reusing relationship to the other groups.

Because we only eliminate "exact same relation" duplicates, it can still happen that in a value model, for a certain combination of source and target value concepts, there are multiple type relations with the same name available to conform to. Let these relations be called R. That happens because R relations have the same name and the sources and targets are in subtype relationships. In this situation, we sort R relations, which have the same name, but different concept types. We first sort based on which relations have more specific concept types as source, and then, in case of a tie, which relations have more specific concept types as target. When we have the two value concepts specified in a value relation conforming to R, the R is going to be the first match among Rs where the value concepts are instances of type concepts in relation R. For an example, see Figure 4.11, where R is a_to_b.

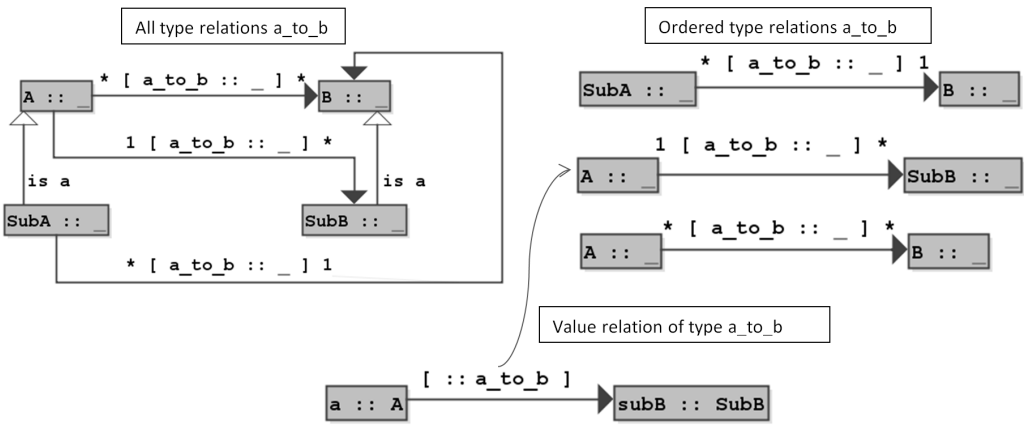


Figure 4.11: On the left-hand side of this figure, we showcase a type model containing relations with the same name but with different source and target concepts that are in a subtype relationship. On the right-hand side, we showcase the sorted relations, and an example of what relation is chosen for given value concepts a and $subB$.

4.1.3 Fragment abstraction and application extension

For the fragment abstraction extension, we envisioned a simple reuse mechanism to capture fragments of models with placeholders, where the placeholders will be substituted by actual model elements at the places of their application. This triggered the idea of using the substitution mechanism available in lambda calculus. Note that we are using lambda calculus for its substitution power mostly, and not for its computational power. Moreover, the fragment abstraction and application extension was part of our exploration and we did not integrate it smoothly with all features of MetaMod (see Section 5.2.2). The additions made to MetaMod with the fragment abstractions and applications are defined in Figure 4.12 and they are explained in the next sections.

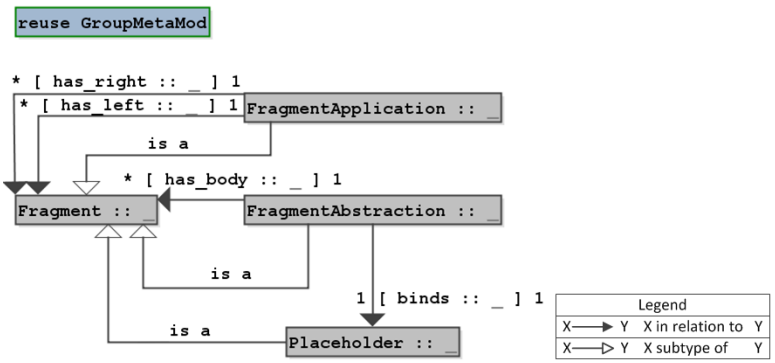


Figure 4.12: Fragment abstraction and application extension of the group meta-metamodel of MetaMod.

4.1.3.1 Pure untyped lambda calculus

The pure untyped lambda calculus is a computational paradigm where computations are achieved via substitutions. Its language is formed of lambda terms. A lambda term, in the pure untyped lambda calculus, can be any of the following:

- a variable x;
- a lambda abstraction $\lambda x.t$, where x is a variable and t is a lambda term;
- or a lambda application ts, where t and s are lambda terms.

The essential operation in untyped lambda calculus is called β-reduction. The β-reduction rule states that an application of the form $(\lambda x.t)s$ is reduced to $t[x := s]$. This means that x is going to be substituted by s in the body of t. Variable x is called a bound variable in this case because it is bound to the lambda abstraction where it was declared.

4.1.3.2 Model elements

The lambda calculus elements defined in the previous subsection have the following counterparts in MetaMod: a lambda abstraction is a **fragment abstraction**, a lambda application is a **fragment application** and a bound variable is a **placeholder**. Because all these elements are subtypes of the fragment element, they can be used anywhere in a group. Moreover, a placeholder is not typed, so anything can be plugged in a placeholder as long as the resulting model with the replacements is a valid model. Whether this is a valid model is to be discovered once the fragment abstraction is applied. This can be done at modeling time, when it is triggered by the user, or it can be done before processing the model, when it is triggered by MetaMod.

Fragment abstractions and applications help with the reuse goal. The model elements fragment application, fragment abstraction and placeholder residing in the meta-metamodel itself, give users the possibility to abstract away commonly occurring patterns and reuse them. Moreover, fragment abstractions can be viewed as a generalization of groups when the body of a fragment abstraction is a group (and this is how we generally use fragment abstractions at the moment).

4.1.3.3 Semantics of fragment applications

The semantics of the fragment application is given by how it is reduced. The first strategy to reduce fragment applications in MetaMod is a variation of call-by-name reduction [127]. We chose this reduction strategy because it is easy to implement and because it is enough for the simple cases we mostly use it for, that of substituting a specific model element. Moreover, the call-by-name strategy is also employed by the second reduction strategy we provide.

In Algorithm 1, the reduction process is depicted for a fragment element F. The reduction strategy is called on the constituent elements of fragment F (line 8 in Algorithm 1), except when fragment F is a fragment application (line 1 in Algorithm 1). For fragment applications, we make a reduction of F.has_left. If the result is a fragment abstraction, then we make a reduction on the result of the substitution of F.has_right for the placeholder F.has_left.binds in the body of the fragment abstraction. Otherwise, the fragment application does not reduce. One can see that the reduction process is based on substitution (line 4 in Algorithm 1). To better grasp Algorithm 1, Figure 4.13 shows the reduction of a lambda application and Figure 4.14 shows the reduction of a lambda abstraction.

Note that because we use a projectional environment, we do not need to worry about some issues that other implementations of lambda calculus are confronted with, such as name capturing. In textual substitution, name capturing happens when the substitution conflicts with the name of a free variable. In projectional editing, even if the substituting variable and the free variable have the same name, they are actually different variables.

Algorithm 1 Reduce(Fragment F)

```
1:  if F is FragmentApplication then
2:      replace F.has_left by Reduce(F.has_left)              ▷ Recursive call
3:      if F.has_left is FragmentAbstraction then
4:          substitute F.has_right for F.has_left.binds in F.has_left.has_body
5:          replace F by Reduce(F.has_left.has_body)
6:      end if
7:  else
8:      for relation R where F is a source concept do
9:          replace F.R by Reduce(F.R)
10:     end for
11: end if
```

We also have a variation of normal-order-reduction [127] to reduce lambda terms in MetaMod. We introduced normal order reduction because we also played with the computational power of lambda calculus, and the call-by-name reduction does not necessarily reduce a term to its normal form (form that can not be reduced anymore with beta reductions), while normal order reduction does. This allowed us to experiment with meta-programming [156] facilities, though in an inefficient way. This was part of our exploration, and we left the computational part out in the end because the models were going too much towards an imperative style [138].

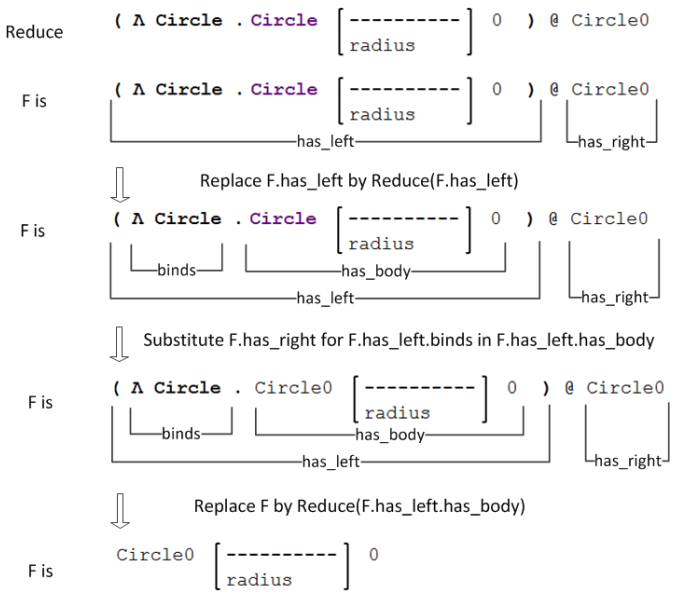


Figure 4.13: The steps of Algorithm 1 applied on a fragment application.

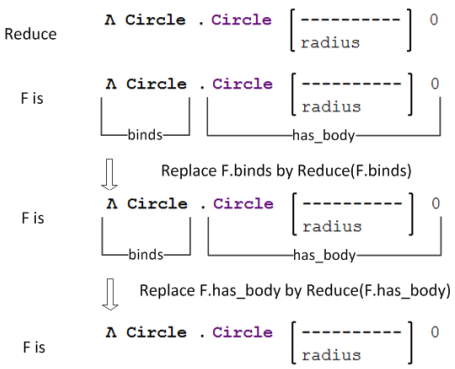


Figure 4.14: The steps of Algorithm 1 applied on a fragment abstraction.

4.1.4 Implementation extensions

In the previous sections, we treated MetaMod as something more abstract, independent of its textual and visual representations. In this section, we introduce the visual and textual representations of MetaMod. Before that, we discuss on implementation concerns of MetaMod and extra elements that we introduced for implementation.

4.1.4.1 Named elements

For implementation purposes, we have added a name attribute to the concept, relation, group, and fragment abstraction elements. Naming is the way to identify these modeling elements in MetaMod. Names are not necessary for identification in graphical models, but they are necessary in textual models. Given that MetaMod has both a visual and a textual representation (see Section 4.1.5) for its models, we have also introduced names

for the modeling elements that can be referenced in textual models.

Now that we know how to identify various elements in MetaMod, a more precise definition of when two concepts or two relations have the same identity can be given. These definitions are given in Algorithm 2 and Algorithm 3. The algorithm for "exact same relations" (mentioned in Section 4.1.2.2) is similar to Algorithm 3, just that in line 11, only exact concepts are accepted, and no sub-concepts.

Algorithm 2 AreSameConcept(Concept C_0, Concept C_1) returns boolean

```
1: if C0 == _ && C1 == _ then
2:     return true
3: end if
4: if C0 == _ || C1 == _ then
5:     return false
6: end if
7: if (C0.name == null || C1.name == null) then
8:     return false
9: end if
10: if (C0.name == C1.name) && AreSameConcept (C0.conforms_to, C1.conforms_to)
    then
11:     return true
12: end if
13: return false
```

Algorithm 3 AreSameRelation(Relation R_0, Relation R_1) returns int

```
1: if R0 == _ && R1 == _ then
2:     return true
3: end if
4: if R0 == _ || R1 == _ then
5:     return false
6: end if
7: if R0.name == null || R1.name == null then
8:     return false
9: end if
10: if (R0.name == R1.name) && (AreSameRelation (R0.conforms_to,
    R1.conforms_to)) then
11:     if (AreSameConcept(R0.source, R1.source) || R0.source is sub-concept of
    R1.source || R1.source is sub-concept of R0.source) && (AreSameConcept (R0.target,
    R1.target) || R0.target is sub-concept of R1.target || R1.target is sub-concept of
    R0.target) then  ▷ For "exact same relations", only the exact concepts are accepted,
    and no sub-concepts
12:         return true
13:     end if
14: end if
15: return false
```

One restriction that we impose as a result of introducing names for groups is that they have unique names in the modeling space. The restriction is in place for the disambiguation

of reused groups. Although it would be no problem for the projectional environment itself to identify two different groups with the same name, for a human user, that would not be the case.

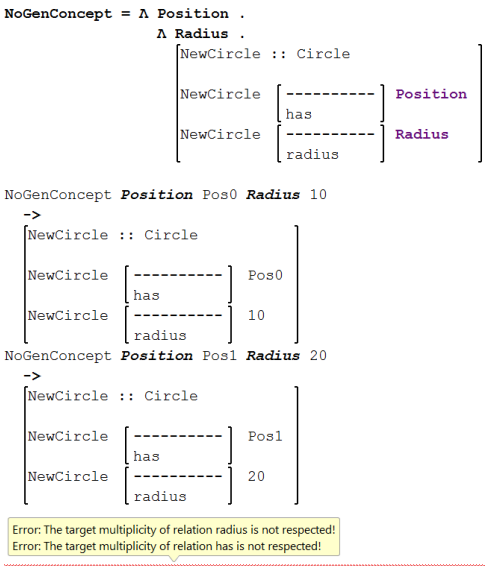
4.1.4.2 Extra elements introduced for implementation

The elements we present in this section were introduced as a result of practical observations while implementing DSLs with MetaMod. These complications arose because we chose names as identifiers for elements in MetaMod. Firstly, we noticed that when a fragment abstraction creates a regular concept, C, and the fragment abstraction is applied twice in the same model, then the two applications would create concept C twice in the same model. This would be erroneous in many cases. For instance, consider a fragment abstraction that creates a concept of type *Circle*, called *NewCircle*, and assigns a position and a radius to this circle by means of relations. Applied twice, *NewCircle* will have two positions and two radii assigned to it (see Figure 4.15a), which is erroneous because the associated metamodel does not allow it (see Figure 4.19). To solve this, we have introduced the *generational concepts*. Secondly, we noticed that when one reuses other groups, she might not want to mix certain reused concepts with concepts having the same name and conforming type concept in the current model (by default, they would be considered as having the same identity). For instance, consider type model G, that reuses type models G_1 and G_2. If both G_1 and G_2 contain concept $C :: \_$, then reusing them in G would result in treating both occurrences of $C :: \_$ as the having the same identity in G. Because there are cases when we want $C :: \_$ from G_1 to be different from $C :: \_$ in G_2 in the reusing group G, we have introduced *equivalence classes* and the *unique* field. All these new elements are explained in the next paragraphs.

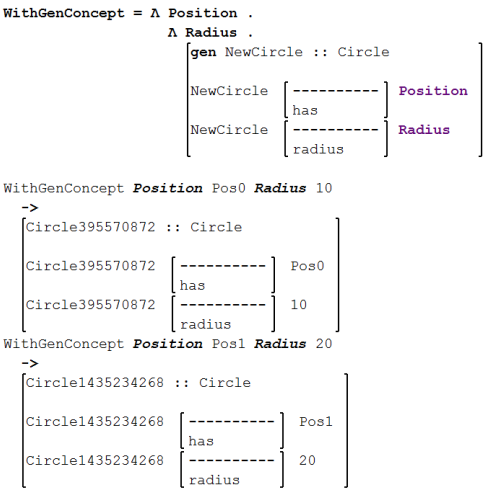
Generational concepts One element that we introduced specifically for the fragment abstractions is the generational concept. The generational concept has a name so that it can be referenced inside of the fragment abstraction, but this name is changed at application time by MetaMod. This construct is needed to create different concepts for different applications in the same model. For instance, if we go back to the example with circles in the previous paragraph, and we make *NewCircle* a generational concept, we obtain two different circles at application time (see Figure 4.15b).

Equivalence classes By default, in MetaMod, if one reuses two concepts that have the same name and the same type concept, they are considered to have the same identity. Nevertheless, there are cases when we wish that two concepts with the same name and type concept be different. Consider the case of a type model called *StateMachines* that reuses both simple state machines, *SimpleStateMachine* and composite state machines, *CompositeStateMachine*. Type model *SimpleStateMachine* contains a type concept *State* :: _ and type model *CompositeStateMachine* contains a type concept *State* :: _. In type model *StateMachines* we do not want the *State* :: _ from the simple state machines to be the same as the *State* :: _ from the *CompositeStateMachines*.

We introduced the equivalence classes so that DSL users can specify that a certain reused concept is not equivalent with other concepts with same name and same type concept in a given reusing group. With the introduction of equivalence classes, two concepts have the same identity if they belong to the same equivalence class. Moreover, the introduction of equivalence classes prevents the regeneration of code for the reused



(a) Lambda abstraction applied twice in the same model. This leads to an unintended model, because *NewCircle* has two relations of type *has* and two relations of type *radius*, which is not allowed by the metamodel (see metamodel in Figure 4.19).



(b) Lambda abstraction applied twice in the same model. This time, new circle concepts are created at the application because they are generational concepts.

Figure 4.15: Applications of lambda abstractions with and without generational concepts. These two model snippets are screenshots from MetaMod. The textual syntax of MetaMod is explained in Section 4.1.5.1, but here we give an explanation of the model snippets in words. Both model snippets start with a lambda abstraction that has a name followed by bound variables (introduced with λ), and followed by the body of the abstraction. In both cases, the body consists of defining a value concept and two value relations that conform to type relations *has* and *radius*. The lambda abstractions are followed by two lambda applications where values are assigned to the bound variables; immediately after the right arrow, the result of the application is shown (the body of the lambda abstraction with the bound variables assigned).

groups (see Section 5.2). A simple renaming would require that code from reused groups be regenerated.

An equivalence class is a list of concepts in a model that have the same name and type concept, and that represent equivalent concepts (concepts that have the same identity). A model is characterized by a set of equivalence classes. When reusing a group, the DSL engineer or user can specify what concepts from the reused group are not allowed to be mixed with concepts having the same name and concept type in the current model. We call those concepts from the reused groups *uniques*. Going back to the example of state machines introduced two paragraphs before, we can declare *State* from *SimpleStateMachine* and *State* from *CompositeStateMachine* as unique in *StateMachines*. Figure 4.16 shows how we define uniques in MetaMod.

Algorithm 4 shows how MetaMod computes the equivalence classes for a group.

A few informal rules that follow from Algorithm 4 for calculating the equivalence

Algorithm 4 CreateEquivalenceClasses(Group G)

Let list L be the list of concepts defined directly in G
Let *eqClasses* be the list of equivalence classes of G, initially empty ▷ An equivalence class is itself a list of concepts.
▷ First we create equivalence classes for concepts declared directly in G, that have the same identity according to Algorithm 2.
for index i in range $[0, size\ of\ L]$ **do**
    if L_i is not in any equivalence class of *eqClasses* **then**
        Create equivalence class Eq
        Add Eq to *eqClasses*
        Add L_i to Eq
        for index j in range $[i+1, size\ of\ L]$ **do**
            if AreSameConcept(L_i, L_j) **then**
                Place L_j in Eq
            end if
        end for
    end if
end for
Let *eqClassesReused* be the list of equivalence classes from reused groups, initially empty
for reused group RG of G **do**
    Add all equivalence classes from CreateEquivalenceClasses(RG) in *eqClassesReused*
end for
▷ Then, we add equivalence classes to G for all concepts that were defined unique in reused groups.
for *eqClass* in *eqClassesReused* **do**
    if any concept in *eqClass* is defined unique in G **then**
        AddUniques(*eqClass*, *eqClasses*) ▷ AddUniques first checks whether *eqClass* intersects (concepts with the same identity) with other equivalence class in *eqClassses*, and if so, then it adds the difference to the equivalence class of *eqClasses*. Otherwise, it adds *eqClass* as a new element to *eqClasses*.
        remove *eqClass* from *eqClassesReused*
    end if
end for
▷ Finally, we add the rest of the equivalence classes to G from reused groups.
for *eqClass* in *eqClassesReused* **do**
    AddNonUniques(*eqClass*, *eqClasses*) ▷ AddNonUniques first checks whether *eqClass* intersects (concepts with the same identity) with other equivalence class in *eqClassses*, and if so, then it adds the difference to the equivalence class of *eqClasses*. Otherwise, it checks whether *eqClass* intersects (same name) with other equivalence class in *eqClasses*, and if so, then it adds the difference to the equivalence class of *eqClasses*. Otherwise, it adds *eqClass* as a new element to *eqClasses*.
end for

```
StateMachines :: _ group {
  reuse SimpleStateMachine unique [ SimpleStateMachine.State]
  reuse CompositeStateMachine unique [ CompositeStateMachine.State]

  State :: _
}
```

Figure 4.16: The *StateMachine* group. This is a screenshot from MetaMod. This textual syntax is explained in Section 4.1.5.1, but here we give an explanation of the screenshot in words. The *StateMachines* group reuses *SimpleStateMachine* and *CompositeStateMachine*, while defining *State* as unique in both reused groups. Then, the *StateMachine* group itself defines a concept *State*.

classes are that 'uniques' have priority, 'uniques' are greedy, 'uniques' propagate from lower groups to upper groups, and 'non-uniques' with same name and conforming concept are part of the same equivalence class. In MetaMod, the creation of equivalence classes for a group and its containing groups is triggered whenever the DSL engineer or user is editing a model and adds a concept to this model.

Going back to the example of state machines, Figure 4.17 shows the equivalence classes of group *StateMachines* as calculated by Algorithm 4.

```
equivalence class for: State
  all concepts:
    State
  all containing groups:
    StateMachines
equivalence class for: SimpleStateMachine.State
  all concepts:
    SimpleStateMachine.State
  all containing groups:
    SimpleStateMachine
equivalence class for: CompositeStateMachine.State
  all concepts:
    CompositeStateMachine.State
  all containing groups:
    CompositeStateMachine
```

Figure 4.17: The equivalence classes of *StateMachines*. The visual representation of the equivalence class contains a list of concepts that were defined in all reused groups of group *StateMachines*, including group *StateMachines* (introduced by the line "all concepts"). These concepts all have the same name and the same conforming concept (we list them in the representation because users can click on the concept and go to its definition). The equivalence class also contains references to reused groups defining these concepts (introduced by the line "all containing groups"). This representation is for debugging purposes, but we show it here for clarity.

With the introduction of the equivalence classes, the algorithm for determining that two concepts have the same identity changes; it is not Algorithm 2 anymore. The check now boils down to verifying whether the two concepts are from the same equivalence class.

4.1.5 MetaMod syntax for models

We have implemented the ideas presented in Section 4.1 into MetaMod using Jetbrains MPS. MetaMod has both a textual and a graphical syntax, and the same model can be projected using any of these two syntaxes. We use examples in both syntaxes throughout the thesis.

4.1.5.1 Textual syntax

The textual syntax of our models is "mostly textual", because there are a few graphical additions to make some of the elements stand out. This also means we can not use BNF [96] to describe the syntax. Both value and type models are projected in the same way, with slight differences.

The textual syntax of MetaMod consists of the following elements.

- Concepts are described by the name of the concept (or an empty space if the concept is unnamed) followed by the '::' symbol and the name of the concept to which the concept conforms (or the '_' symbol if there is no such concept). See Figure 4.21 and type concept *Circle*, for an example.
- Relations are described by a source concept followed by the source cardinality, the '[' symbol, and a dashed line. Above the dashed line lies the name of the relation (or an empty space if the relation has no name) and underneath the dashed line lies the name of the relation to which the relation conforms (or the '_' symbol if there is no such relation). This sequence is followed by the ']' symbol, the target cardinality, and the target concept. Note that the notation with dashed lines spans multiple lines, making this "more than a textual" notation. See Figure 4.21 and type relation *radius*, for an example.
- Groups are described by the name of the group (or an empty space if the group is unnamed) followed by a '::' symbol, the name of the type model to which the group conforms (or the '_' symbol if there is no such type model), the 'group' keyword, the '{' symbol, a list of fragments and the '}' symbol. The groups are enclosed with a border for an extra visual hint of the structure of the model, making this "more than a textual" notation again.

  To reuse other groups, one can introduce a line that starts with 'reuse', followed by the name of the group that is to be reused and a list of concepts that are to be treated as unique. See Figure 4.21 and group *Circle*, for an example.
- Subtype relationships are described with a concept followed by the 'is a' words and a super-concept. See Figure 4.21 and the subtype relationship between circle and shape, for an example.
- Fragment abstractions are described with the name of the fragment abstraction, followed by the '=' symbol, the symbol 'Λ', the name of the placeholder, the '.' symbol and the body of the fragment abstraction. This notation was inspired by the lambda calculus notation. See Figure 4.28 and fragment abstraction *3ConcentricCircles*, for an example.
- Fragment applications are described by a left fragment followed by the name of the placeholder we are substituting and the right fragment. See Figure 4.29 and the fragment application on *3ConcentricCircles*, for an example.

Even if one can not create a custom syntax for MetaMod models, using fragment abstractions, fragment applications and groups, users can obtain models with custom look and feel (see Figure 4.29 and the application of *3ConcentricCircles*).

4.1.5.2 Visual syntax

Models in MetaMod also have a visual representation. The explanation of these elements follows:

- Concepts are represented by a rectangle containing the name of the concept, followed by '::' and the concept to which it conforms. See Figure 4.21 and concept *Circle*, for an example.
- Relations are represented by a filled arrow line labeled with the name of the relation, and the relation to which it conforms, and the source and target cardinality at the left-hand side and right-hand side of the label. The source concept and target concept are the source and target of the arrow line, respectively. See Figure 4.21 and type relation *radius*, for an example.
- Groups are enclosed in a thin border box with the name of the group followed by '::', the group to which this group conforms and by opening curly parentheses. The curly parentheses enclose a rectangle with the containing model elements of the group. The reused groups appear as green rectangles labeled with the name of the reused group. See Figure 4.21 and group *Circle*, for an example.
- Subtype relationships are represented by an open arrow line labeled with 'is a'. The sub-concept and super-concept are the source and target concepts of the arrow line, respectively. See Figure 4.21 and the subtype relationship between circle and shape, for an example.
- Fragment abstractions are represented as in the textual notation, just that the body will be visual. In the body, placeholders are represented with dark magenta rectangles labeled with their names. See Figure 4.28 and fragment abstraction *3ConcentricCircles*, for an example.
- Fragment applications are represented by a yellow rectangle labeled with the name of the lambda abstraction, and with edges towards all its arguments. The edges are labeled with the name of the argument. See Figure 4.29 and the fragment application on *3ConcentricCircles*, for an example.

4.1.6 Related work

In this section, we are going to look at modularity mechanisms targeted at value models (models of DSLs) and type models (metamodels). In most works on metamodels presented in the following paragraphs, there are no mechanisms to incrementally add relations to reused concepts, that are valid in a reusing context. They mostly rely on techniques similar to inheritance, where additions are made on inheriting concepts. Moreover, many of the works do not consider modularity mechanisms for value models, leaving those to be encoded in the type models themselves.

One of the most recent works on providing modularity for value models includes the work of Nuno Amalio et al. on Fragmenta [5]. Fragmenta establishes a mathematical

theory of model fragmentation for MDE that offers fragmentation strategies for value models. Thus, the way fragmentation is done in value models is determined at the type model level. With our approach the fragmentation in value models is not enforced by the DSL engineers.

The work on concern oriented software design [4] is also of relevance for us. The mechanism used in concern oriented development is based on concerns, which are units of reuse that encapsulate software models. A concern has three interfaces (variation, customization, and usage) that are used when instantiating the concern. Concerns are based on Reusable Aspect Models (RAMs) [75]. The focus in the concern oriented software design is on software models and on the metamodel level.

The work of Clark et al. [25] on a Meta-Modeling Framework (MMF) describes an interesting approach geared towards the creation of OO modeling languages. The two features of OO modeling technology the mechanism employs are package specialization (for support of reusable, modular, incremental language design) and package templates (for parametric model elements). The mechanism is focused on the metamodel level of a language definition and on OO modeling languages.

Catalysis [40] is a software design mechanism based on UML and focused on the specification and design of component-based systems. The mechanism is built on three modeling concepts: the type (the behavior of one object), the collaboration (the interactions among a group of objects) and the refinement (from business models to specifications to implementations). Recurring patterns of these three concepts are captured in frameworks with the help of template packages. The mechanisms is focused on the models used in the design of a software system.

Another interesting work is that of Varro et al. [153] on VPM, a visual, precise and multilevel metamodeling framework. They also rely on a kernel language formed of a few constructs, that they designed in order to alleviate the various problems they identified in MOF itself, among which structural redundancies in MOF are mentioned. They present mechanisms of reuse for both the static structure and dynamic structure of models. In what regards the static structure, they use refinement (meaning inheritance and instantiation) for classes, associations and packages alike. This gives rise to a mechanism of reuse. This style of extension does not allow for augmenting classes in reusing contexts. Moreover, we also have fragment abstractions and fragment applications that allow for reuse with variation points.

The work on generic metamodeling with concepts, templates and mixin layers of Juan de Lara et al. [32, 34] is implemented in a tool called MetaDepth. They use templates and so-called *concepts* to define requirements on the parameters of the templates.

Similar works are those described by Heidenreich et al. [56] and Clark et al. [26]. Heidenreich et al. describe a mechanism to accomplish language-independent model modularization. The mechanism is based on using interfaces for the language fragments to be combined. Then, Clark et al. define metamodel composition with the help of package extensions and package templates. Given that the composition mechanism was defined at the package level, it can be that the modularity capability is limited due to coarse granularity.

In the Reuseware Composition Framework [59], a generic approach to add modularity to arbitrary languages is defined. This approach is part of the Invasive Software Composition mechanism [7] and it uses a gray-box composition technique, interfaces, hooks and composition operations.

Another work, that on role-based language composition [163] transfers role-based modeling to the metamodels. Here, roles define placeholders for class types and a semantic

contract for objects that will play the roles. The composition of the languages is made by an external composition program. In contrast to our work, role-based language composition works at the metamodel level only.

UML package merge [114] is a mechanism to provide modularity in UML. It is a relationship between two packages and it indicates that the contents of the target package are merged into the contents of the source package under certain rules. Similarly, Kompose, a generic approach for automatic model composition is made in two steps: matching (specific to each modeling language) and merging (generic). On the same line of work, proxies in EMF allow for the partitioning of models [135]. On the other hand, none of these mechanisms provide parameterization techniques.

4.1.7 Challenges

In the previous sections we have introduced and motivated the elements of the meta-metamodel. Here, we present a couple of challenges that these new elements might pose. Firstly, the way relations are shared among groups could create difficulties for DSL engineers. They might find it hard to understand what cardinalities apply for a value relation conforming to a type relation that was duplicated in the type model (see Figure 4.11 again for an example). Secondly, the sharing of concepts might create difficulties as well for DSL engineers, because it can be hard to keep track of the equivalence clases. Thirdly, another challenge is raised by the fragment abstractions and the placeholders being untyped. Only after the application is the user notified of errors (although they can apply the fragment application in-place, thus allowing MetaMod to check the placeholders are of the right type). The first two challenges could be alleviated with good tool support (showing the equivalence classes and the duplicates per group, showing the relation chosen by the algorithm, etc.). The third problem could be solved by adding types to the fragment abstractions.

4.2 Organization of MetaMod meta-languages

In this section, we give an overview of the main meta-languages we have built for MetaMod. MetaMod is composed of six main meta-languages, dealing with the structure of DSLs (models and lambda calculus), the documentation of models (documentation), the API functions for querying and navigation of models (generic functions), the processing of models (processing units) and the model transformations (model transformations). These meta-languages were implemented using Jetbrains MPS. The implementation in MPS accompanies the definitions and the explanations of all the mechanisms for the design and implementation of DSLs we describe in this thesis.

MPS gave us freedom in the choice of notations for meta-languages of MetaMod. The projectional nature of MPS enabled the "almost textual" and visual notations presented in the previous section. Moreover, we have reused various other languages provided by MPS while building MetaMod itself. This was useful in getting a working prototype of our meta-languages ready within a few months.

The organization of the meta-languages in MetaMod is depicted in Figure 4.18. We leave out the languages that we reused from MPS, and concentrate only on the meta-languages that we built for MetaMod. In the following paragraphs, we present all these meta-languages. We have one more main component meta-language that is not in the figure and that we will present in Section 7, the *MappingChangeableModules*.

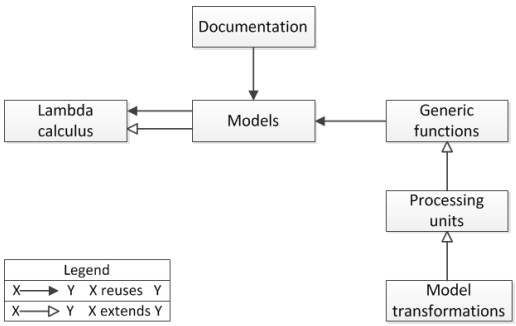


Figure 4.18: The meta-languages of MetaMod. What language reuse and extension mean in MPS is presented in Section 2.4.

4.2.1 Models

The *Models* meta-language is at the center of the meta-languages of MetaMod. This component contains most of the elements of the meta-metamodel that we described in Section 4.1, except for the fragment abstraction, fragment application and placeholder. These other elements are captured in the *Lambda Calculus* meta-language, that is reused in the *Models* meta-language.

4.2.2 Lambda Calculus

The *Lambda Calculus* meta-language has all the elements of untyped lambda calculus and it also provides two types of reductions for the lambda terms: a call-by-name reduction and a normal order reduction. This language can be used independently of MetaMod, for instance, for teaching purposes.

4.2.3 Documentation

The *Documentation* meta-language allows developers to write documentation for the models. This meta-language embeds elements from the *Models* meta-language. The documentation can lead to better understanding of the models and their modular structure because in the documentation one can explain the models, can provide embedded examples and can reference model elements that are written in separate files. On the other hand, modularity of models also helps during documentation because groups create a context in which to discuss about finer-grained elements (distinct groups and their contents).

For an example that captures most of the features of the documentation meta-language see Figure 4.23 in Section 4.3.1.3.

4.2.4 Generic functions

The *Generic functions* meta-language is a collection of primitive API functions and MetaMod types that help navigate, query, and create models. Moreover, we also define operations (the constituents of processing units) and ways to organize these operations in the *Generic functions* meta-language.

The functions that we provide for navigating and querying models follow. Note that the left and right angle brackets, "⟨" and "⟩", in the following definitions denote a

placeholder, a value that needs to be introduced when writing the function.

- $\langle VM \rangle.conceptsOfType(\langle CT \rangle)$: outputs those value concepts from value model VM that conform to type concept CT;
- $\langle CV \rangle.@src\#\langle RT \rangle\#\ in\ (\langle VM \rangle)$: outputs those value concepts from value model VM that are the target of a value relation conforming to type relation RT, where the source is value concept CV;
- $\langle CV \rangle.@tgt\#\langle RT \rangle\#\ in\ (\langle VM \rangle)$: outputs those value concepts from value model VM that are the source of a value relation conforming to type relation RT, where the target is value concept CV;
- $\langle CT1 \rangle.isTypeOf(\langle CT2 \rangle)\ in\ (\langle VM \rangle)$: outputs true if $CT1$ is a subtype of $CT2$, and false otherwise; this is done in the context of value model VM and its type model;
- $\langle CV \rangle.castTo(\langle CT \rangle)\ in\ (\langle VM \rangle)$: tells the type system to treat CV as conforming to type concept CT; this is done in the context of value model VM and its type model;
- $\langle CV \rangle.strValue$: outputs the name of concept CV.

The '@src' and '@tgt' functions always return a list of values, no matter the multiplicity end of the target or source concept. Moreover, one can notice that there are no operations for updating value models in this list. These operations are less interesting; they simply allow for the creation and deletion of concepts, groups, subtypes, and relations.

We also introduce the following three types.

- The $GroupType\#\langle GT \rangle\#$ denotes value models that conform to type model GT;
- The $ConceptType\#\langle CT \rangle\#$ denotes value concepts that conform to type concept CT.
- The $RelationType\#\langle RT \rangle\#$ denotes value relations that conform to type relation RT.

Note that, in the thesis, we will be referring to concept types when we talk about types that occur in operations of the processing units, and that originate from type concepts. As a reminder, a type concept is a concept from a type model. The same discussion holds for group types and relation types.

4.2.5 Processing units

The *Processing units* meta-language provides facilities for processing units and their operations: operation call resolution and operation overriding. The operations in the processing units are an extension of methods from the base language of MPS (that is a reimplementation of Java). That is why operations look similarly to Java methods, and can also have Java types as arguments.

The main code generator for MetaMod that we implemented in MPS is defined as part of this meta-language. This code generator transforms the processing unit operations into Java code, that is then run on the value models. We denote this by *processing unit transformation*. We need to distinguish the processing unit transformation from the following situation. The operations in the processing units themselves can contain code that generates code. We denote this by *code generation with the processing units*. In the next paragraphs we further explain these two types of generation.

Processing unit transformation For the purpose of explaining the transformation of the processing units into code, we need to introduce the notion of multi-operation here. A multi-operation is a special type of operation that can be overridden in MetaMod. No other type of operation can be overridden in MetaMod. We give full details on the multi-operation in Chapter 5. Moreover, we also give one more detail about the processing unit, the fact that it is defined per group and language aspect. Again, we give full details on the aspect reuse of processing units in Chapter 5.

A processing unit is defined for a group and for a language aspect. Assume we have groups G_i, where $i \in [1, g]$, available in the model space, and processing units $PU_{G_i}^{A_j}$ defined for groups G_i and aspects A_j, where $j \in [1, a]$. The highlights of the processing unit transformation are as follows:

- For all $i \in [1, g]$, a Java interface, I_{G_i}, is generated from every group.
- For all $i \in [1, g]$, I_{G_i} implements interfaces I_{G_k}, where G_i reuses G_k directly, where $1 \leq k \leq g$.
- For all $i \in [1, g]$, all multi-operations defined in $PU_{G_i}^{A_j}$, where $j \in [1, a]$, are placed as operation signatures in I_{G_i}.
- For all $i \in [1, g]$, a Java class, C_{G_i}, is generated from every group.
- For all $i \in [1, g]$, C_{G_i} implements interfaces I_{G_i}.
- For all $i \in [1, g]$, C_{G_i} contains a collection field with all classes C_{G_k}, where G_i reuses G_k recursively.
- For all $i \in [1, g]$, operations from $PU_{G_i}^{A_j}$, where $j \in [1, a]$, are placed as methods in C_{G_i}.
- In the generated code, the name of a multi-operation is prepended with the string 'multi_' and appended with the names of the parameter types.
- In the generated code, the name of an operation that overrides a multi-operation is made of the name of the operation concatenated with the names of the parameter types (we call this the actual name).
- The implementation of the interface operations in C_{G_i} is made of a switch among the different overriding operations in the order of the specificity of the arguments, as will be explained in Section 5.1.
- A call to a multi-operation in the processing unit is generated into a call to the interface operation implementation.

In summary, we generated Java classes per group and we expressed the flattened reuse hierarchy of groups as compositions in the generated Java classes. Moreover, we introduced Java interfaces per group so that multi-operations defined for any reused group are implemented in the Java classes generated per group. Lastly, we implemented multi-operations with a switch on the runtime type of the arguments of the multi-operation call such that the appropriate overridden method is chosen.

In Appendix A, we have placed code generated from the example with shapes from Section 4.3. All elements we talked about in this section can be inspected there.

Code generation with the processing units A DSL engineer can also generate code while processing the value models. For this purpose, MetaMod offers extra API functions to the processing units.

Because the processing units are an extension of the base language of MPS, we are able to combine them with other extensions built for the base language. One extension that we use for generation purposes is an API to create Java nodes programmatically, *openapi*, from package *org.jetbrains.mps.openapi* (this package was created by the MPS team). Another extension for creating Java nodes programmatically, but that is more compact, is *jetbrains.mps.lang.quotation* (this package was also created by the MPS team). The quotations are environments enclosed between %(and)%, that can be written inside the operations of a processing unit and that function similarly to template languages. One can write target Java code in this environment, and can escape to the scope of the operation in the processing unit by using so-called anti-quotations. The quotations and openapi were developed by the MPS team to be used in the language aspect DSLs of MPS itself. They are used in the type system DSL of MPS, for instance, to create the inferred type nodes. An example of code generation with the processing units can be seen in Section 9.3.

4.2.6 Model transformations

The *Model transformations* meta-language provides support for creating transformations from DSLs written in MetaMod to other DSLs written in MetaMod. The model transformation meta-language is actually an extension of the processing units meta-language and it brings in new functionality related to creating the output model. This meta-language is described in more detail in Chapter 8.

4.2.7 Discussion

One can notice that the *Models* component is reused in all other components, except for lambda calculus, that is actually part of the definition of the models. Thus, the implementation of MetaMod itself has the models meta-language as a central meta-language. We discuss about this phenomenon of a central component in Section 9.6.

4.3 Example DSL and models in MetaMod

In this section we present examples written in the main meta-languages of MetaMod. We describe type models, simple value models, value models with fragment abstractions, documentation models, processing units and extensions of DSLs. That is so one gets an idea of the process of creating DSLs and models of these DSLs in MetaMod. The DSL we use in all these examples is a DSL to create 2D shapes at specified points on a canvas, and it is called the *Canvas* DSL. The two incremental extensions to the *Canvas* DSL are related to adding a new type of shape and adding colors to the shapes, respectively.

The entire metamodel of the *Canvas* DSL is defined in Figure 4.19; this metamodel does not contain the elements from the two extensions. A canvas can contain two types of shapes, circles and rectangles, and all shapes are characterized by a position on the canvas. The assumption we make is that for the circle, the position represents its center, and for the rectangle, the position represents its upper-left corner.

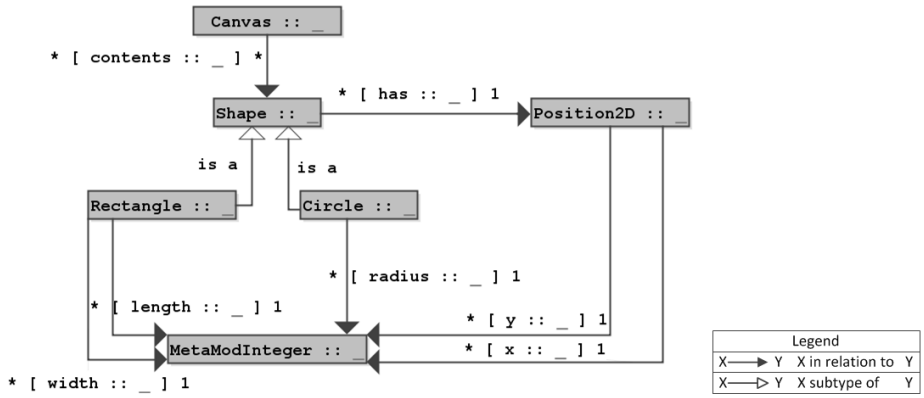


Figure 4.19: All the concepts and relations in the *Canvas* DSL - flattened view.

4.3.1 Implementation steps

To implement the DSL, we start off by implementing the metamodel and example models. We then build documentation models and processing units for various aspects, e.g. visualization and constraints.

4.3.1.1 Type models

We build metamodels in MetaMod using the group construct. All the groups we have built for the Shapes language, including the two extensions, are shown in Figure 4.20. Most of these groups require information from other groups and add its own information. For instance, the *Circle* group is defined in Figure 4.21. Thus, looking at the *Circle* group, we see that shapes are imported first, because the circle type is a subtype of the shape type which was defined in group *Shape*. Besides the inherited position which represents the center, the circle also needs a radius; thus, we define a radius relation for the circle.

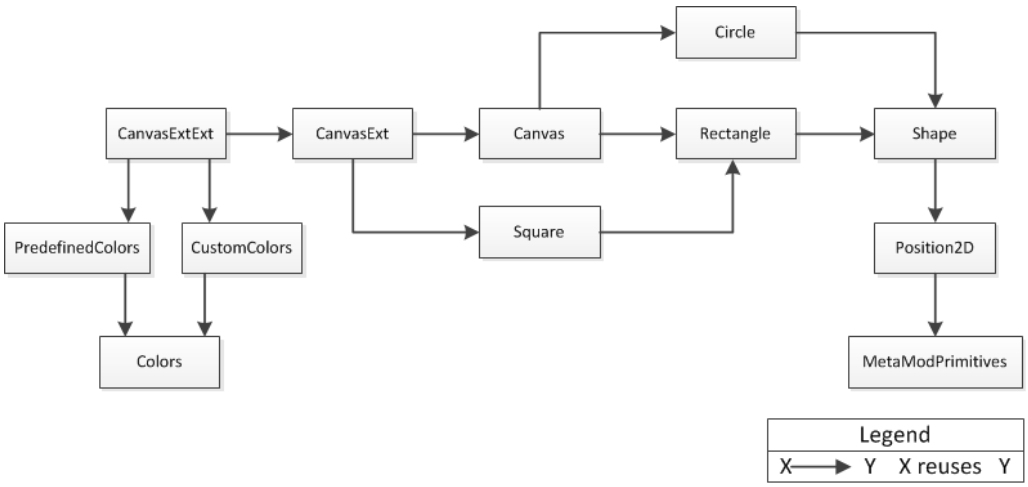


Figure 4.20: The components of the *Canvas* DSL.

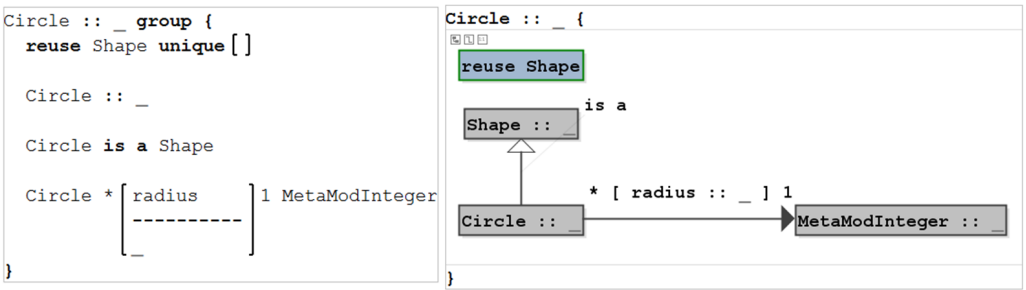


Figure 4.21: The *Circle* group in its textual form on the left-hand side and in its visual form on the right-hand side.

4.3.1.2 Value models

We write a model that contains one circle and one rectangle in the new *Canvas* DSL. We are modularizing the top-level model by creating two separate models: one that specifies the details of the circle (see Figure 4.22) and one that specifies the details of the rectangle (see Figure 4.22). Then, the top-level model reuses the previous two and places them on the canvas (see Figure 4.22). This example illustrates how value models can be modularized the same way as type models. One can notice that the value model defining a circle conforms to type model *Circle* and the value model defining a rectangle conforms to type model *Rectangle*, but they are both reused in a value model that conforms to type model *Canvas*. This is possible because both groups *Rectangle* and *Circle* are reused by group *Canvas* in the metamodel.

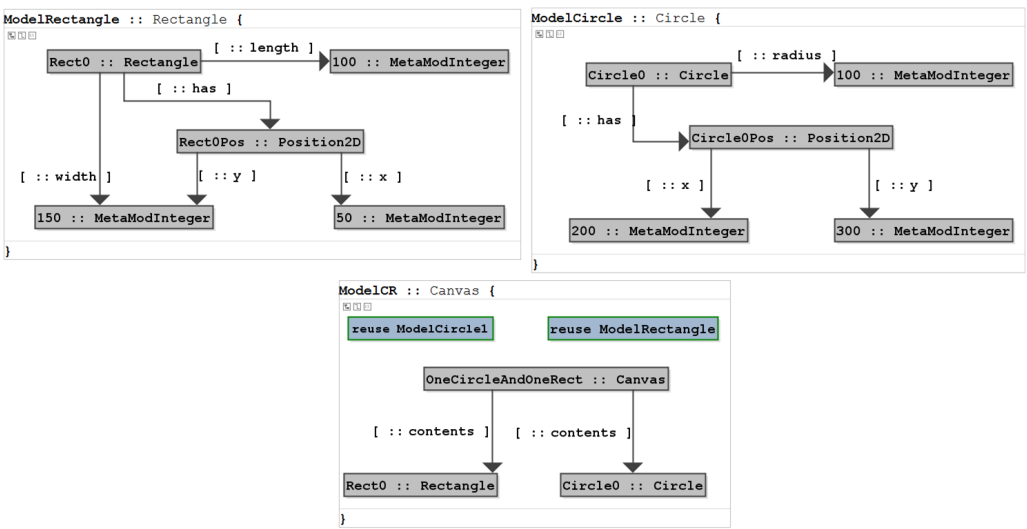


Figure 4.22: Value models defining a rectangle, a circle, and a canvas containing a circle and a rectangle.

4.3.1.3 Documentation

We also attach documentation to the *Canvas* DSL. This comes in the form of a document with chapters as in Figure 4.23. The document contains actual references to model elements, which means that a change in the original model will propagate automatically to the documentation. Moreover, the example also includes an embedded model example.

```
Documentation Canvas DSL of Canvas
{
  This document describes a simple DSL for creating a canvas with 2D shapes on it.

  Chapter Circle
  {
    In the first chapter we look at the circle shape. We start with an example of a
    value model named ModelCircle that conforms to type model Circle.

    ModelCircle :: Circle group {
      << ... >>
    }

    << ... >>
  }

}
```

Figure 4.23: Documentation for the *Canvas* DSL. There are references to actual model elements represented in blue and a model example embedded in this documentation.

4.3.1.4 Processing units

There are many types of processing that could be performed on the shapes language, from defining extra constraints on the shapes, to visualizing them in different tools, and checking mathematical properties on them. In this particular case, we do not need to define extra constraints on the models. We only want to draw the shapes on the canvas. That is why we create a processing unit with an aspect called *Draw* and we associate it to the *Shape* group. Here, we define a multi-operation called *drawShape* that draws the shape using a Java *graphics* object (see Figure 4.24). This multi-operation can then be overridden for a particular type of shape in groups reusing *Shape*; for instance, see the overridden operation for a circle shape in Figure 4.25.

To generate the actual drawing, the *drawShape* operation is subsequently called by another operation, *createPanel*, that creates a Java panel object to show the shapes in a GUI component. We mention this other operation, *createPanel*, because it is going to play a role later in an extension.

4.3.2 Extension one

The first extension showcases a simple way to augment a language. We add a new concept to draw on the canvas, the *Square* concept, and we create a group reusing both *Square* and the old canvas, called *CanvasExt*. We chose to make square a subtype of rectangle. Thus, we need to define a constraint, checking that the length and the width of the square are the same (see Figure 4.26). We also define a main operation for the group *CanvasExt*,

```
PU_Shapes_Draw
for group Shape
aspect Draw
       reuses << ... >>

@multi-operation
operation drawShape(GroupType#Shape# inputGroup, ConceptType#Shape# shape, Graphics graphics)
    returns void {
  error "drawShape needs to be overridden for " + shape;
}
```

Figure 4.24: The multi-operation *drawShape*.

```
@override drawShape
operation drawShape(GroupType#Circle# inputGroup, ConceptType#Circle# circle, Graphics graphics) returns void {
  graphics.drawOval(circle.@src#has# in (inputGroup).first.@src#x# in (inputGroup).first.intValue,
      circle.@src#has# in (inputGroup).first.@src#y# in (inputGroup).first.intValue,
      circle.@src#radius# in (inputGroup).first.intValue, circle.@src#radius# in (inputGroup).first.intValue);
}
```

Figure 4.25: The overriding of multi-operation *drawShape* for shape *Circle*. In this figure, one can see navigation and querying functions that we defined in Section 4.2. Because all the '@src' and '@tgt' functions return a list, no matter the multiplicity of the target or source ends of a relation, we use operation *first* to get the value for single-valued ends.

where we simply call the main operation defined for the original canvas. No changes to glue the original canvas with the new shape are needed. That is because square is of type shape and a canvas contains shapes.

```
@constraint depends on << ... >>
operation LengthAndWitdhEqual(GroupType#Square# inputGroup) returns boolean {
  for (ConceptType#Square# sq : inputGroup.conceptsOfType(ConceptType#Square#)) {
    if (!(sq.@src#length# in (inputGroup).first.intValue ==
        sq.@src#width# in (inputGroup).first.intValue)) {
      error "The length and width of a square need to be equal!";
      return false;
    }
  }

  return true;
}
```

Figure 4.26: Constraint checking that width and length are the same for a square shape.

4.3.3 Extension two

The second extension showcases an augmentation that is more invasive. We embed colors into the *Canvas* DSL and add a relation from the *Shape* concept to a color concept. Besides these additions to the metamodel, that are shown in Figure 4.27, we also need to make changes to the processing units. That consists in rewriting the *createPanel* in a new operation called *createPanelWithColor*, where we first change the color of the graphics object to the color of the shape before drawing the shape. Moreover, in the main operation we call *createPanelWithColor* instead of *createPanel*.

An alternative would have been to make *createPanel* a multi-operation, and to override

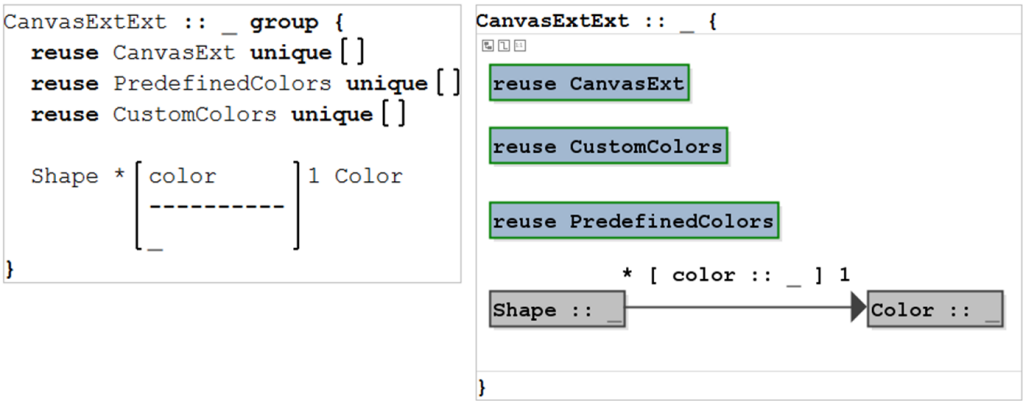


Figure 4.27: The group for the second extension in its textual form on the left-hand side and its visual form on the right-hand side.

it in the extended language. That implies having access to the original language, and making a non-invasive change. On the other hand, one could predict that such a change might take place and she could flag the *createPanel* operation as a multi-operation from the beginning. This shows the kinds of decisions DSL engineers face when they need to think of extensions.

4.3.4 Models with fragment applications

It is at the value model level that is most valuable to capture groups with placeholders, because there are more repetitive structures happening at this level. For instance, we have defined a fragment abstraction for defining three concentric circles in Figure 4.28. The body of the fragment abstraction is a group with placeholders *x*, *y*, *radius1*, *radius2* and *radius3*.

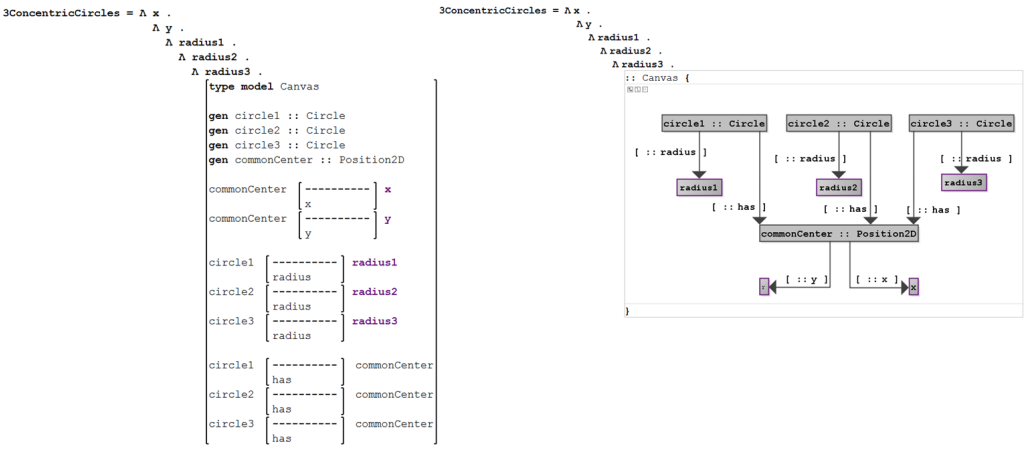


Figure 4.28: A fragment abstraction in its textual form on the left-hand side and its visual form on the right-hand side.

The application of the fragment abstraction is shown in Figure 4.29.

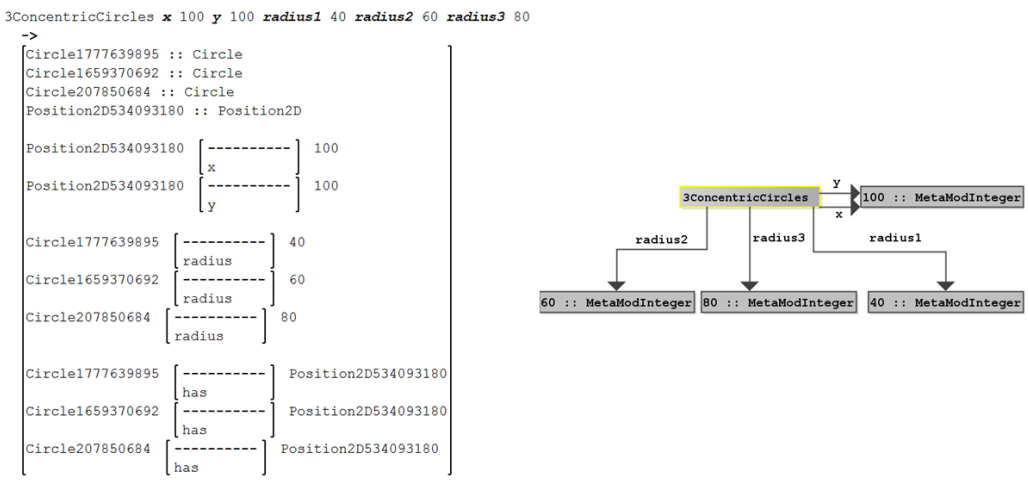


Figure 4.29: A fragment application in its textual form on the left-hand side and its visual form on the right-hand side.

4.3.5 Running the processing unit on the value model

Once we run the operations in the processing units associated to the *Draw* aspect and the *Canvas* group on a value model, we obtain a canvas with all the shapes in the model drawn on the canvas. To understand how does the processing unit get to run on the value model, see Figure 4.30. The processing unit contains concept types, groups types and relation types from the type model, and it makes use of these types in the code for navigating and processing value models. This code then transforms to executable code (such as described in Section 4.2.5). The generated code can then be run on any value model that is an instance of the type model for which the processing unit was defined.

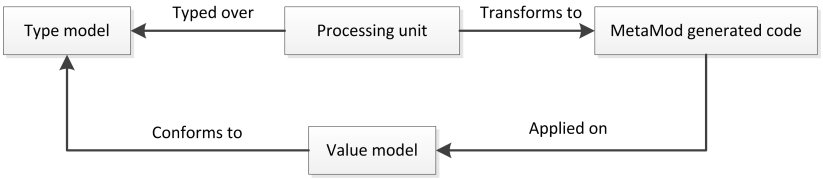


Figure 4.30: Figure showing how does the processing unit run on a value model.

4.4 Conclusions

In this chapter, we have defined the meta-languages of MetaMod. We have started with the central meta-language, the meta-metamodel, that defines type and value models. We have described it incrementally, starting with the core, and then adding the group extension, the fragment abstraction and application extension, and the implementation extension. The features of the meta-metamodel were guided by the goals we had with MetaMod: modularity, reuse and simplicity. Our contributions here consist of the way groups and group reuse share concepts and relations, the combination of model elements

with lambda calculus in fragment abstraction and application, and the incorporation of the conformance relationship in the meta-metamodel to create the multilevel nature of MetaMod.

We have then defined the other meta-languages of MetaMod. With the help of these meta-languages we have described the API functions for navigating and querying the models (the generic functions meta-language), and the way we generate code from the processing units (the processing unit meta-language). This is where we made another contribution. The way we generate code from the groups and their associated units is, to the best of our knowledge, unique. Other language workbenches usually generate a class per concept, while we generate a class per group; this, in turn, facilitates modularity and opens the possibility for separate generation (see Section 5.1.3.1).

Finally, we have presented the implementation of a DSL that highlights all of the meta-languages. Despite the simplicity of the meta-metamodel, we have sufficient power to create non-trivial DSLs.

Chapter 5

Features of MetaMod

In this chapter, we discuss the features of MetaMod from two different perspectives, that of modularity and reuse, and that of fulfilling the language workbench requirements. In the end, we discuss related work and we look back at the contributions of this chapter.

5.1 Features for modularity and reuse

In this section, we discuss features of MetaMod that contribute the most to the creation of modular and reusable DSL units. Although there might be other features in MetaMod that contribute to some degree to modularity and reuse of DSL units, we have chosen to illustrate only those that we think have the biggest impact. We demonstrate these features using examples from an expression DSL unit implemented in MetaMod. This expression DSL unit is presented in Section 9.2 and it is advised that one reads Section 9.2 before reading this section. In the process of discussing the features of MetaMod, we also get into implementation details of the expression DSL unit. The features we discuss in this section are as follows.

- That metamodels are organized and manipulated via groups that share concepts;
- That groups (through reuse) and concepts (through subtyping) give rise to two separate hierarchies: the group hierarchy and the concept hierarchy;
- That operations are defined in the processing unit and its associated group;
- That processing units do not have state, but they only have operations;
- That multiple dynamic dispatch is made on the concept hierarchy, raw Java types, and the group hierarchy.

Thus, this section addresses both research questions RQ_2 and RQ_3.

> **RQ_2:** *How can we organize metamodels of the DSLs such that we facilitate modularity and reuse of DSLs?*

RQ$_3$: *How can we organize processing units of the DSLs and the operations in the processing units such that we facilitate modularity and reuse of DSLs?*

As a convention, in the next subsections, we use the notations of Table 5.1.

	notation
group	G or H
aspect	X or Y
processing unit	PU_G^X

Table 5.1: Generic notations.

Each of the next subsections will discuss one feature. A subsection has the following structure: high-level discussion (we refrained from using any implementation details in this part), and meta-tools implementation details, with examples. Moreover, the focus in this section is going to be more on type models than on value models, because we talk about implementations of DSLs.

5.1.1 Metamodels are organized and manipulated via groups that share concepts

We have approached the issues of modularity and reusability starting from the core language aspect of a DSL, the metamodel. This is where we define and organize concepts and relations between them. The structure of metamodels is captured in the meta-metamodel that is depicted in Figure 5.1. This figure is the expansion of Figure 4.9 from Section 4.1, that represents the group extension. We expand it here for clarity.

As a small reminder, to understand the structure of metamodels in MetaMod, one needs to understand a couple of things, viz. the notion of a model element, or a fragment, of which MetaMod distinguishes three kinds, viz. *Group*, *Concept* and *Relation*. One also needs to understand the relationships between these model elements. In particular, a *Group* can contain zero or more *Fragment*s, and a *Relation* relates a source *Concept* to a target *Concept*. Moreover, there are also the conformance and the subtyping relationships defined on a concept. Although apparently simple, there are interesting implications of the organization of model elements given in Figure 5.1, as can be seen in the next two subsections.

5.1.1.1 High-level discussion

We have chosen to organize the metamodels into elements called groups. Groups are containers of groups, concepts, and relations. Note that a model, and in particular a metamodel, is a group. We refer to metamodels as groups when we want to put the emphasis on the organizational nature of the metamodel.

A group can reuse any other group (see relationship *contains* with *Group* as a subtype of *Fragment* in Figure 5.1). The only constraint on metamodels in MetaMod is that the reuse graph is acyclic. The semantics of *group reuse* is that a reusing group is equivalent to a group where the contents of all the reused groups are copied inside of the reusing group, recursively. The group reuse mechanism is a generalization of both the extension and the default reuse in MPS. One major difference with the mechanisms of language reuse in MPS is that reused concepts in MetaMod can have their definition[1] augmented in

[1]Note that *the definition of a concept* consists of all the relations where the concept is a source of.

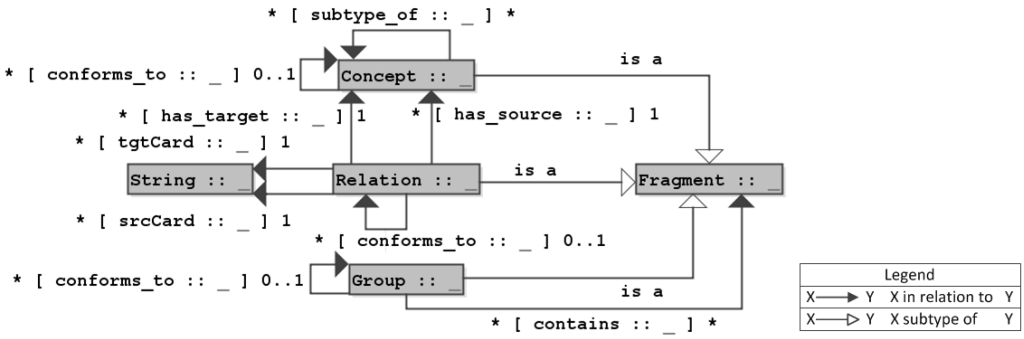


Figure 5.1: The meta-metamodel of MetaMod depicting the three main modeling elements: concepts, relations and groups, and the relationships among them.

reusing groups. Augmenting a concept means adding new relations in the reusing group, where the concept is a source of these relations.

Augmenting properties Augmenting concept definitions in reusing groups is mainly enabled by two properties of the group: concept sharing among groups and groups owning relations. The concept sharing mechanism of MetaMod is captured in relationship *contains* between *Group* and *Fragment* in Figure 5.1. *Concept* is a type of *Fragment* and the source cardinality of *contains* is $*$, which means that the same concept can be part of multiple groups. Moreover, the property of groups owning relations is also captured in relationship *contains* in Figure 5.1. Again, *Relation* is a type of *Fragment* and, thus, relations are part of groups. We call these two properties, the *augmenting properties*. The consequences of these two properties and how they empower augmenting concept definitions in reusing contexts is explained in the next paragraphs.

Consequences of augmenting properties As a result of the augmenting properties, the definition of a concept C is formed of all the relations where C is a source in the current group and all its reused groups. Figure 5.2 shows an example that exploits this consequence. Thus, relations can be added to a reused concept in any reusing group. This, in turn, leads to encouraging modular definitions of the metamodels. Yet another way of interpreting this consequence is that a group can confer more power to reused concepts. According to Stroustrup, "a concept does not exist in isolation; it co-exists with related concepts and derives much of its power with relationships with related concepts" [136]. This actually resembles a much older theory in cognitive science, the "Theory-Theory", that says that "Concepts are representations whose structure consists in their relations to other concepts as specified by a mental theory" [89].

Another consequence of the augmenting properties is that the definition of a concept can be split among different groups. Figure 5.3 shows an example that exploits this consequence. One can argue that having the definition of a concept split among different groups will make it harder to grasp the complete definition of a concept in a group but this can be solved with good tool support. The tools can show alternative views with the complete definitions of the concepts in a given group.

Additionally, the augmenting properties can lead to having less concepts in the metamodel because it reduces the need for concepts that are meant for implementation

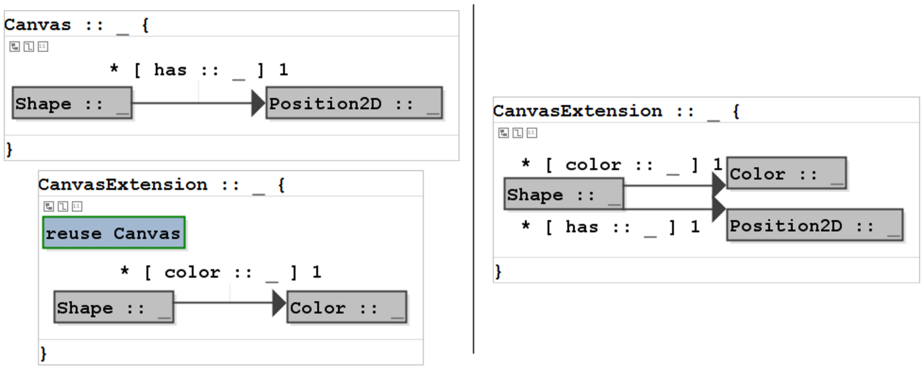


Figure 5.2: The two groups on the left-hand side show an incremental definition for concept *Shape*. The group on the right-hand side is a flattened view (reused groups eliminated) of group *CanvasExtension*.

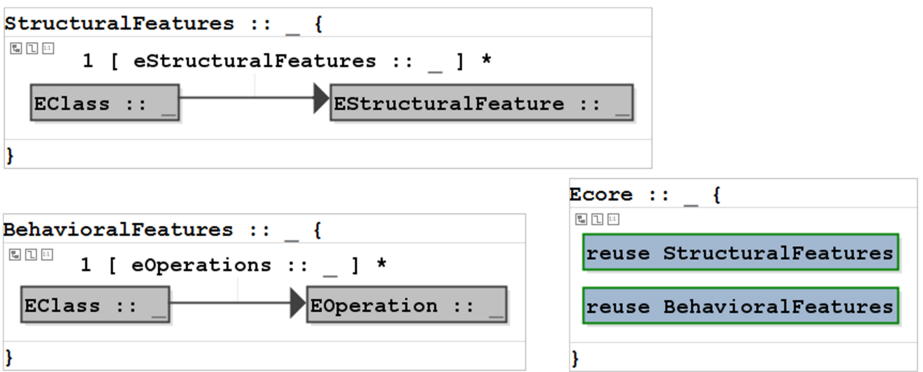


Figure 5.3: Excerpt from the Ecore metamodel. The definition of a class is split between behavioral features and structural features. The behavioral features of a class are captured in group *BehavioralFeatures* and structural features of a class are captured in group *StructuralFeatures*. The two definitions come together under the *Ecore* group.

only. To better illustrate this point, consider the following example. Assume that concept *CContainer*, which represents a container, accepts only a certain concept *CContent* as its content (there is a one to many relationship between *CContainer* and *CContent*). Concept *CContent* is introduced just to distinguish concepts that can be contained in *CContainer*. Any concept that we wish to place in *CContainer* needs to be a subtype of *CContent*. When reusing a concept from another DSL unit, *CExternal*, we cannot place it in *CContainer*, unless we specify *CExternal* is a subtype of *CContent*. In MetaMod, one can simply define *CExternal* as a subtype of *CContent*. In comparison, this is not possible outside of the reused DSL in inheritance-based language workbenches, like MPS. The only way to augment the definition of a reused concept in MPS is by first creating a new concept extending the reused concept that we want to augment. This implies we need to extend *CExternal* to specify that it is also a subtype of *CContent* (which might actually not be possible if there is no multiple inheritance support and *CExternal* is already a subtype of some other concept). This is not an ideal approach, because

the extended *CExternal* and *CExternal* itself, represent, conceptually, the same concept. Figure 5.4 shows an instance of this example.

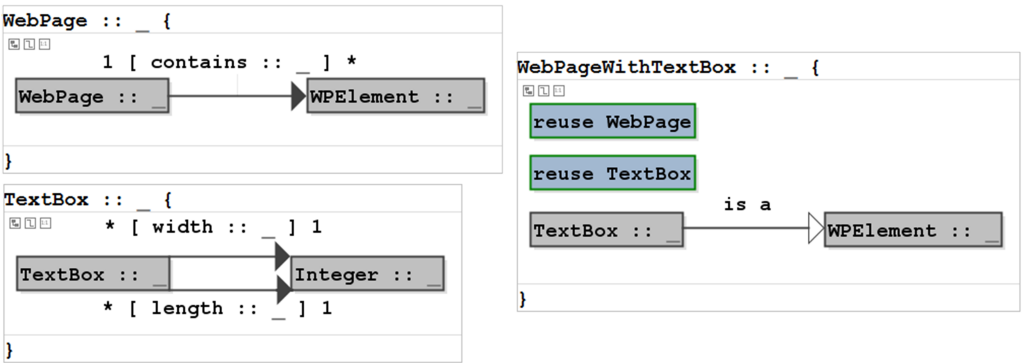


Figure 5.4: Modeling of a webpage and a textbox. This an instance of the container example, where the concept *WebPage* is *CContainer*, the concept *WPElement* is *CContent*, and the concept *TextBox* is *CExternal*.

5.1.1.2 Meta-tools implementation details, with examples

MetaMod implements the ideas described in the previous section. One important implementation point is that we have chosen to identify concepts by their name. Thus, whenever a group reuses two (or more) other groups, and these other groups contain identically named concepts that conform to the same concept, $C :: T$, then concepts $C :: T$ denote one and the same concept in the reusing group. There is also a mechanism in MetaMod to specify that a concept in a reused group is different from any other concept in the reusing group, and this was explained in Section 4.1.

In the next paragraphs, we take excerpts from the expression language defined in Section 9.2.

In regard to concept sharing, as an example, consider concept *Expression* in group *Contract* and concept *Expression* in group *ExpressionsAndTypes*. Looking at Figure 9.5, one can see that the two groups are defined separately. Subsequently, when the two groups are reused in the combination group *ExpressionsAndTypesAndContract*, the two concepts *Expression* represent the same concept.

In regard to adding relations to a concept in reusing groups, as an example, consider the combination of group *Alternatives* with group *SimpleTypes*, that is defined in group *BaseExprAndSimpleTypes* in Figure 5.5. Table 5.2 depicts the concepts relevant for this discussion from groups *Alternatives* and *SimpleTypes*. Group *SimpleTypes* defines, among others, boolean literals. One of the boolean literals it defines is the *OtherwiseLiteral*. This literal is used in expressions that have multiple boolean branches, and *OtherwiseLiteral* represents the default branch (in case that conditions of the other branches do not hold). Moreover, group *SimpleTypes* also defines a constraint which specifies that *OtherwiseLiteral* can only be placed in a *IValidOtherwiseContainer*. That is because there are operations of *OtherwiseLiteral* that make use of the fact that *OtherwiseLiteral* is contained in a *IValidOtherwiseContainer*. Group *Alternatives*, on the other hand is made of an alternatives expression that has multiple branches, where a branch is called an *AltOption*. Each of the alternative branches has relations to two expressions: one representing a

condition and one representing the expression to be evaluated in case the condition holds. When we combine *SimpleTypes* with *Alternatives*, we declare *AltOption* as a subtype of *IValidOtherwiseContainer* (see Figure 5.5), because we want the otherwise literal to be part of an alternative branch in the condition part. We could not do this in group *Alternatives*, because *Alternatives* was not aware of *OtherwiseLiteral*, and we could not do this in group *SimpleTypes*, because *SimpleTypes* was not aware of the *AltOption* concept. The most logical place to insert this subtype relation is in the combination of *Alternatives* and *SimpleTypes*, without creating a new *AltOption* that extends the original one. Thus, in the new context, that of the combination, *AltOption* is a valid container for *OtherwiseLiteral* concepts.

Group	Concepts defined in the group
Alternatives	*AltOption*
SimpleTypes	*OtherwiseLiteral* *IValidOtherwiseContainer*

Table 5.2: Part of the concepts defined in groups *Alternatives* and *SimpleTypes*.

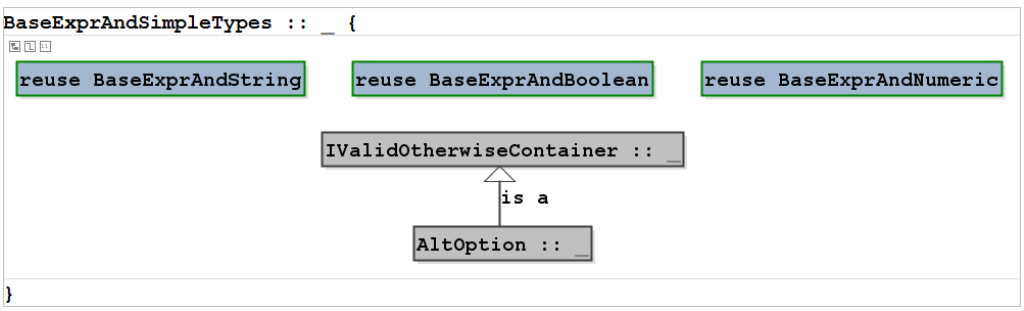


Figure 5.5: Introduction of a subtype relation when combining base expressions with simple types. Concept *AltOption* is from group *Alternatives* and concept *IValidOtherwiseContainer* is from group *SimpleTypes*.

5.1.2 Groups (through reuse) and concepts (through subtyping) give rise to two separate hierarchies: the group hierarchy and the concept hierarchy

The reuse relationships for groups in the metamodels give rise to a group hierarchy and the subtype relationships give rise to a concept hierarchy. In MetaMod, we take advantage of these hierarchies from type models (value models do not count here) by making operations polymorphic at runtime over the hierarchies. That means the same operation can have multiple implementations for different type concepts or groups that are in a child-ancestor relationship in these hierarchies. At operation call, the actual implementation is selected on the runtime type of the arguments.

5.1.2.1 High-level discussion

The group hierarchy is generated by the group reuse mechanism. The group reuse mechanism has similarities to inheritance in object-oriented programming languages. In this respect, one could say that the reusing group inherits the model elements in the reused group. That is so, because the reusing group gets the contents, i.e., contained model elements, of the reused group, and it can also add extra relations, concepts, and groups. Moreover, when using groups as types in the operations, we consider the reusing group as a subtype[2] of the reused group. That means the reusing group can be used in place of the reused group in the operations. That is, in processing units of reusing groups, operations defined with a reused group type parameter can be called, instead, with a reusing group type argument; thus, the group hierarchy is used in the type system of MetaMod.

Thus, the group reuse mechanism has two roles: reuse of model elements and subtyping on the operations level. The latter role gives yet another perspective on group reuse: given that the group represents a metamodel and that the metamodel is the core part of a DSL, one could say the DSL engineer treats DSLs as classes, and passes DSL models around as objects.

On the other hand, the concept subtype hierarchy is given explicitly in the metamodel through the subtype relationship among concepts (see Figure 5.1, and the *subtype_ of* relationship). On the metamodel level, the concept subtype relationship is a way of inheriting relations. That means that the sub-concept can be part of all relations that the super-concept can be part of. On the processing unit level, a sub-concept is a subtype of a super-concept in the type system of MetaMod. This implies that a sub-concept can be used anywhere where a super-concept is expected in operations of processing units.

5.1.2.2 Meta-tools implementation details, with examples

In MetaMod, a group value, concept value, or relation value conforms to a concept type, a relation type, or a group type. All types describe a set of values that conform to the type. The types are defined in a metamodel (type model), while the values are defined in a model (value model). Note that the operations are always defined on the group types and concept types level, and they are applicable to the group values and concept values level.

In the processing units, MetaMod denotes a group type with *GroupType##* and a concept type with *ConceptType##*. Between the hashtags, the DSL engineer can specify a concrete group name from the metamodel and a concrete concept name from the metamodel. For instance, defining the operation *typeOf* for the *Binary* group and concept *BinaryExpression* requires having these two types as parameters (see Figure 5.6). Because of the group hierarchy and the concept hierarchy, this operation can be called with any reusing group of *Binary* and with any sub-concept of *BinaryExpression*.

```
@override typeOf
operation typeOf(GroupType#Binary# inputGroup, ConceptType#BinaryExpression# binaryExp) returns
    TypeConcept
```

Figure 5.6: Signature of operation *typeOf* for binary expressions.

[2]Note that this does not have anything to do with the subtyping relationship on concepts, but it has to do with the subtyping notion from programming languages theory [91].

The group and concept hierarchies are used extensively in the operations of the processing units. Something that might be counter-intuitive is that group types and concept types are both used as types in operations of the processing units, despite concepts being contained in groups in the metamodel.

5.1.3 Operations are defined in the processing unit and its associated group

A processing unit is the unit of organization for the operations that process the models. The operations in the processing unit navigate and query the models, while at the same time they perform computations on the data obtained from the models. The way the processing units are organized is important because it influences the understandability and maintainability of the DSL implementation, and it also influences the chances that the DSL is reused.

The previous two features discussed were centered around the organization of the metamodels. We will see that the organization of the metamodels via the groups influences the organization of the processing units. This reinforces the importance of a good mechanism to group the metamodels.

5.1.3.1 High-level discussion

There are two dimensions to a processing unit: the processing unit is defined per group and per aspect; an aspect is a string that spells out the purpose of the processing unit (the same way the name of a class spells out the purpose of the class). The notion of aspect here is related to language aspects; related, and not the same, because DSL engineers can use aspect more loosely, and call code generation differently, for instance. DSL engineers are not limited to a fixed set of aspects in MetaMod (although they will probably define the usual ones as well, such as, code generation); they can define operations for any aspect they see fit. In that sense, aspect can be interpreted to mean any goal that the processing unit is meant to accomplish. For a comparison, we take the case of MPS. In MPS, there are a limited number of language aspects to define for a DSL: structure, editor, intentions, type system, etc.

A processing unit is a collection of operations and it can define operations with type concepts (concepts from the associated group, including concepts from reused groups) as parameters. This has important consequences on the extensibility of DSLs. It results in the freedom to add operations over reused concepts in reusing groups. Operations are not defined on the concept (like in the OO paradigm), but with the concept as a parameter (like in the functional paradigm). This is a key difference between MetaMod and many other language workbenches.

If we make a comparison between concepts and classes in Java, we see why it is easy to add operations to a concept in MetaMod, but not to a class in Java. In Java, and many other OOP languages, methods are mostly owned by classes, and no new methods can be added to a given class (we do not consider an extension of it) outside of its definition location. On the other hand, operations in our case are associated to groups. That means, operations handling reused concepts can be added in processing units of the containing groups. This resembles the mechanism of open-methods, where the methods are not owned by the classes [119]. Note that this is different from extending the class and adding the methods in the extended class. What is happening in MetaMod is that in a reusing

context, new operations handling reused concepts can be created, without first extending the concept.

One important characteristic is that a processing unit, PU_G^X, can reuse other aspects, for instance, X reuses Y. That means, processing unit PU_G^X has access to operations from all processing units defined for the reused aspect, Y, and for any reused group of G or G itself. Although one could argue that the processing units create a hierarchy as well through the aspect reuse, we do not use this hierarchy in the type system. That is because we introduces aspect reuse of processing units only for organizational and reuse purposes.

A processing unit is influenced directly by aspect reuse and indirectly by group reuse. More formally, processing unit PU_G^X has access to operations of any processing unit PU_h^y, where $H \times Y = \{(h, y) | h \in H \; and \; y \in Y\}$, with $H \times Y$ the cartesian product, H the set formed of G and all its reused groups, recursively, and Y the set formed of X and all its reused aspects, recursively. For an example, see Figure 5.7.

Moreover, the reuse order of the aspects of a processing unit is irrelevant. Aspects reuse of processing units does not affect the semantics of the processing unit: a processing unit with a reusing aspect is equivalent to a flattened processing unit containing the same operations as the processing units defined on reused aspects and on reused groups.

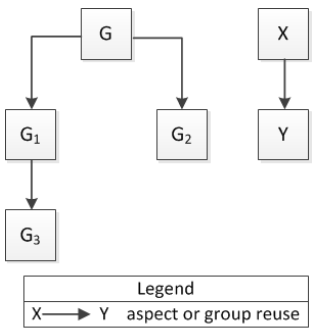


Figure 5.7: Figure showing a hypothetical group reuse hierarchy on the left-hand side and a hypothetical aspect reuse hierarchy on the right-hand side. Given these two hierarchies, PU_G^X would have access to operations in PU_G^X, PU_G^Y, $PU_{G_1}^X$, $PU_{G_1}^Y$, $PU_{G_2}^X$, $PU_{G_2}^Y$, $PU_{G_3}^X$, and $PU_{G_3}^Y$.

5.1.3.2 Meta-tools implementation details, with examples

In MetaMod, processing units are defined in a separate file, and they contain a set of operations.

Adding operations to reused concepts is one of the highlights of the processing units. We illustrate this feature by looking at the combination of group *Contract* with the core group, *ExpressionsAndTypes*. The group *ExpressionsAndTypes* defined an operation called *typeOf* for concepts that will be part of the type system of the DSLs extending *ExpressionsAndTypes*. When combining the expressions and types with contracts, we want to make a concept from contract, *ContractItem*, a part of the type system of the obtained DSL. This entails overriding operation *typeOf* for the *ContractItem* concept. If owned by the concept itself, the operation *typeOf* on *ContractItem* could not be declared in the new DSL, outside of the original definition for *ContractItem*. On the other hand,

in MetaMod, we can add a *typeOf* operation to *ContractItem* because *ContractItem* is part of the DSL combining *ExpressionsAndTypes* with *Contract* (see Figure 5.8).

```
@override typeOf
operation typeOf(
    GroupType#ExpressionsAndTypesAndContract# inputGroup, ConceptType#ContractItem# contractItem)
    returns TypeConcept
```

Figure 5.8: Signature of operation on *ContractItem* residing in the combination of the core DSL, *ExpresssionsAndTypes*, and DSL *Contract*.

5.1.4 Processing units do not have state, but they only have operations

Another feature that helps to a large extent with the reuse of DSLs, is that processing units do not have and do not modify any state (for comparison, in Java classes, a state would be represented by a field, also known as instance variable).

5.1.4.1 High-level discussion

Processing units are collections of operations, and they do not have any state information. This simplifies reuse because there is no state that the operations can affect, and therefore, no undesired interactions among states in operations either.

In OOP, multiple inheritance of state is the one that causes most problems, and not multiple inheritance of behavior (methods) [122]. In the latter, conflicting operations can be solved with overriding.

The sole goal of the aspect reuse mechanism of the processing units is thus to reuse operations. Moreover, there is no sub-typing relationship created as a result of the aspect reuse in processing units. Aspect reuse in processing units resembles the traits mechanism [122], where they emphasize the advantages of only reusing methods (operations) of classes.

5.1.4.2 Meta-tools implementation details, with examples

All language aspects of the expression DSL units have been implemented with operations that process elements of the metamodel. Processing in MetaMod starts with a main operation. Everything is handled through parameters and parameter passing, without changing state in the processing units.

Note that we can combine Java code with the processing units. DSL engineers can create any Java classes and they can make use of these classes in the operations of the processing units.

5.1.5 Multiple dynamic dispatch is made on the concept hierarchy, raw Java types, and the group hierarchy

Multiple dynamic dispatch is the process that allows a program to choose the most specific operation based on the runtime type of multiple parameters, in contrast with single dynamic dispatch that makes the choice based on one parameter. The dynamic dispatch mechanism is essential when reusing DSLs, because we often need to change the

behavior of operations in the reused groups based on the new context (the reusing group) and based on new sub-concepts.

5.1.5.1 High-level discussion

One of the main benefits of OOP comes from runtime polymorphism, also known as dynamic dispatch. This has been one of the reasons to bring dynamic dispatch into MetaMod. Besides that, dynamic dispatch takes further advantage of the two hierarchies that metamodels give rise to: the group hierarchy and the concept hierarchy.

In MetaMod, we call operations with multiple dynamic dispatch capabilities multi-operations. The dispatch is made only on these parameters: concept types, the raw Java types, and group types. We have already explained that the concept types and group types come from hierarchies in the metamodel. Java types, on the other hand, come from the operations being an extension of methods from Java and Java types being allowed as parameters.

The way we make this dispatch is encoded in the call resolution mechanism that has the following two main steps:

- Sort the overriding operations based on the specificity of the operation parameter types.
- The operation call is delegated to the first operation, *Op*, in the sorted list of overriding operations where the type of runtime parameters are more specific than or equal to the parameter types of *Op*.

The comparison among overriding operations is detailed in Algorithm 5. The gist of it is that in comparing two overriding operations, we give priority to the operation that has the first more specific *concept type* parameter in the order parameters appear in the parameter list. If all concept type parameters have the same specificity, then we select the operation that has the first more specific raw Java type parameter in the order parameters appear in the parameter list. If, again, all raw Java type parameters have the same specificity level, then we select the operation that has the first more specific group type parameter in the order parameters appear in the parameter list. Finally, if these parameter types have the same specificity level, the operation with the smallest lexicographical actual name is returned. The actual name of an overriding operation is the name of the operation concatenated with the names of its parameter types (the actual name is computed by MetaMod during the dispatch).

In the call resolution algorithm, one can notice that the decision on which operation to choose is based on multiple concept type parameters. This was motivated from the experience of building DSLs in MPS, where we noticed that in type system aspects, they often needed to make decisions based on the type of multiple concept type parameters. Because of that, they added custom support for these cases in MPS (see the operator overloading feature in the MPS documentation [66]).

It was equally important that in these multi-operations we also make multiple dynamic dispatch on the raw Java types. This was also motivated from the experience of building DSLs in MPS. In the interpreter DSL built by the mbeddr team [95], for instance, they check the Java types to which the left and right expressions of a binary expression evaluate, so that they decide on the operation to be performed on the binary expression. For example, if both types are integers, then integer arithmetic can be performed, while if one type is real, then real arithmetic is performed. However, we can do the checking only on raw Java types because at runtime we lose the generics information due to type erasure.

Algorithm 5 Comparator of 2 operations for dynamic dispatch

```
1: procedure COMPARATOR(op0, op1)
2:     ▷ Procedure that returns the operation with most specific parameters. Returns 1
   if op0 is more specfic, −1 if op1 is more specific and 0 if they are at the same level of
   specificity.
3:     ▷ p_op0_i / p_op1_i represents parameter on position i in the parameter list of op0 /
   op1
4:     for all parameter p_op0_i of type ConceptType from left to right do
5:         if p_op0_i is subtype of p_op1_i then
6:             return 1
7:         else if p_op1_i is subtype of p_op0_i then
8:             return −1
9:         end if
10:    end for
11:    for all parameter p_op0_i of type RawJavaType from left to right do
12:        if p_op0_i is subtype of p_op1_i then
13:            return 1
14:        else if p_op1_i is subtype of p_op0_i then
15:            return −1
16:        end if
17:    end for
18:    for all parameter p_op0_i of type GroupType from left to right do
19:        if p_op0_i is a reusing group of p_op1_i then
20:            return 1
21:        else if p_op1_i is a reusing group of p_op0_i then
22:            return −1
23:        else
24:            return lexicographical comparison result among actualName(op0) and
   actualName(op1)
25:        end if
26:    end for
27: end procedure
```

At the same time, we also do multiple dynamic dispatch on the group type. This allows us to modify the behavior encoded in operations in different contexts (we consider the group to be the context). This is enabled by the fact that the first parameter of an operation is always the group type that the processing unit is defined on.

In Algorithm 5, we use an asymmetric [119] technique to compare the different overriding operations. That means that the order of the parameters matters in the call resolution.

One of the consequences of the comparison described in Algorithm 5 is that if there are multiple overriding operations with the same arguments defined in different groups that are not in a reuse relation to each other, then the operation for the smallest lexicographical group name is chosen. To change this default choice, the operation can be overridden in a containing group.

For a better understanding of the call resolution mechanism, let us take an example with the type hierarchies that are depicted in Figure 5.9. The *G*s are group types, the

*C*s are concept types and *J*s are Java raw types. Let us also assume that C0 is defined in *G0*, *C1* and its subtype relationship to *C0* in *G1* and *C2* and its subtype relationship to *C0* in *G2*. Moreover, assume we have the following operations, where Foo_0 is the multi-operation:

$$
\begin{aligned}
Foo_0 &= \mathrm{foo}(\mathrm{G0}, \mathrm{C0}, \mathrm{J0}) \\
Foo_1 &= \mathrm{foo}(\mathrm{G1}, \mathrm{C0}, \mathrm{J1}) \\
Foo_2 &= \mathrm{foo}(\mathrm{G2}, \mathrm{C0}, \mathrm{J1}) \\
Foo_3 &= \mathrm{foo}(\mathrm{G3}, \mathrm{C0}, \mathrm{J0}) \\
Foo_4 &= \mathrm{foo}(\mathrm{G1}, \mathrm{C1}, \mathrm{J1}) \\
Foo_5 &= \mathrm{foo}(\mathrm{G2}, \mathrm{C2}, \mathrm{J0})
\end{aligned}
$$

Then, using the comparator in Algorithm 5, we get the following order, from most specific to most general: Foo_4, Foo_5, Foo_1, Foo_2, Foo_3 and Foo_0. Looking at the more interesting cases:

- operation Foo_1 comes before Foo_2 because *G1* comes before *G2* in lexicographical order;
- operation Foo_4 comes before Foo_5 because *C1* and *C2* are not in relation to each other and *J1* is a subtype of *J0*;
- operation Foo_3 comes before Foo_0 because *G3* is a subtype of *G0*.

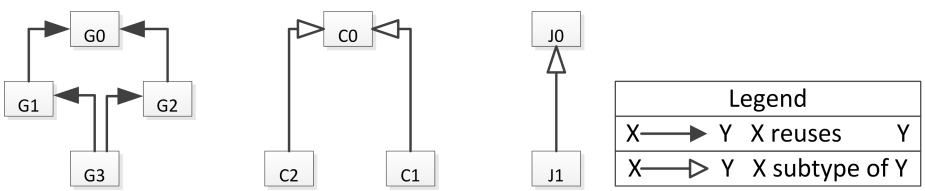


Figure 5.9: Type hierarchies relevant for the *foo* example operation.

Although one might say that it is hard to understand what operations can get called from this abstract example, in practice, at least in our experience with building the expression language where we have dozens of overridings, it was not hard to understand what operation will be called.

5.1.5.2 Meta-tools implementation details, with examples

In MetaMod, the operations in processing units have the look-and-feel of Java methods. The difference is that they are not defined within a class, but they are defined for a certain aspect and for a certain group.

The first parameter of an operation in MetaMod is of a group type. This is because all operations in MetaMod are tied to a group and the group can access concepts and relations defined in that group. There are no other restrictions to the parameters of the operations.

In order to be considered for overriding, an operation needs to have the annotation 'multi-operation'. Once operations in PU_G^X are tagged with the multi-operation annotation, they can then be overridden in any PU_h^y, where h and y are again elements of the cartesian product mentioned in Section 5.1.3.1.

Consider the example of the *typeOf* multi-operation. We declare this multi-operation in an aspect called *SystemWide_ET* associated to the group *ExpressionsAndTypes*. The operation is depicted in Figure 5.10 and it needs to be overridden by all the concepts that are typed in the reusing groups. The default implementation returns an error message. Groups reusing the *ExpressionsAndTypes* group override this multi-operation, such as group *Binary* and group *UnaryExpressions*. The signatures of two overriding operations can be seen in Figure 5.11. Let us take the *Binary* example. The operation is rewritten for the *Binary* group type, which is a subtype of the *ExpressionsAndTypes* group type because *Binary* reuses *ExpressionsAndTypes*. Moreover, the operation is rewritten for the *BinaryExpression* concept type, which is a subtype of the empty concept type. Note that the empty concept type, *ConceptType##*, is a supertype of all concept types.

```
SystemWide_ET
for group ExpressionsAndTypes
processing unit SystemWide

@multi-operation
operation typeOf(GroupType#ExpressionsAndTypes# inputGroup, ConceptType## conceptType) returns
    TypeConcept {
  error "Override typeOf! " + conceptType;
  return null;
}
```

Figure 5.10: Operation computing the type of a concept in a processing unit associated to the group *ExpressionsAndTypes*. Note that the syntax of operations in MetaMod is close to the syntax of Java.

```
@override typeOf
operation typeOf(GroupType#Binary# inputGroup, ConceptType#BinaryExpression# binaryExp) returns
    TypeConcept

@override typeOf
operation typeOf(GroupType#UnaryExpressions# inputGroup, ConceptType#UnaryExpression# ue) returns
    TypeConcept
```

Figure 5.11: Signatures of operations that override *typeOf* from Figure 5.10 with specific arguments.

5.1.6 Discussion

In this section, we highlight the main differences between the MPS approach and the MetaMod approach when implementing the expression language. Thus, this section also offers an answer to research question RQ_6.

> **RQ_6:** *How can modularity and reuse features of language workbenches be evaluated?*

Moreover, we draw parallels between the approach in MetaMod and general-purpose programming approaches so that we get a better understanding of MetaMod.

5.1.6.1 Comparison between MPS and MetaMod implementations

There are a few differences between MPS and MetaMod that influence the ease of developing modular and reusable DSLs. We are mostly going to use the term extensibility

in this section because reusability is strongly linked to extension, and extension is the preferred term in MPS

Looking at the various language aspects of a DSL, one can notice that each language aspect in MPS has its own extension mechanism. Firstly, this contributes to the high learning curve of MPS. Secondly, and more importantly, at least in our own experience, it is hard to decide what needs to be in a reusable DSL unit given that each language aspect can be extended in a different way. One needs to have a very good understanding of how each language aspect can be extended when designing a reusable DSL unit. It is worth mentioning that each language aspect in MPS is defined using a dedicated DSL, which is a common practice in language workbenches. Nonetheless, this does not mean that the extensibility aspect should also be treated differently from language aspect to language aspect.

In comparison, the modularity mechanism in MetaMod is uniform across all language aspects. Although we might decide to create a DSL for a specific aspect of a processing unit (the same way other language workbenches have a DSL for each language aspect), the modularity mechanism should be underneath that, and should surface to the DSL engineer in a way consistent to the syntax of the DSL of the aspect itself. Incidentally, an example of that can be seen in the language aspect for an editor in MPS. If we would consider that the underlying modularity mechanism in MPS is based on operations that can be overridden (which is not the case for all language aspects in MPS), then we can look at the editor as a function implementing the look-and-feel of a concept. Moreover, this editor can be composed of multiple editor components, that can be again regarded as functions. Thus, the editor function is calling the editor components functions, and all these functions can be individually overridden in an extending concept. This is an example of how an underlying modularity mechanism can surface in the DSL of a language aspect.

Moreover, in MPS, one cannot modify a reused concept in a reusing context, neither in the extension, nor in the default reuse case. One can only make additions to an extending concept. This results in spawning more concepts than conceptually needed, especially in trivial cases such as the one with concept *CContainer* in Section 5.1.1. Many times, concepts need small additions in new contexts. Thus, when reusing DSLs, one needs a mechanism different from inheritance to make additions to reused concepts; it is tedious to extend the reused concept and all its sub-concepts to fit the new context.

One major difference between MPS and MetaMod is that, in MPS, DSL users are allowed to combine DSLs in the user space (the DSL users choose the languages and the extensions they need in a solution, without prior combination by the DSL engineers), while in MetaMod, the DSLs are combined by the DSL engineers. It is probably because of the combination of DSLs in the user space, that some language aspect extensions are more restricted in MPS.

Nonetheless, MPS is an excellent tool and this is proven by the multitude of languages that were built in it by the mbeddr team and their customer experiences [158].

One of the downsides of MetaMod is that there are going to be performance losses at generation time (when we create the running code from the processing units) and at runtime (when we run the generated code of the processing units). There are plenty of optimizations that can be made to the generation process, but the most important one is that generation can be done incrementally. That implies to regenerate code for processing units of the modified groups and their containing groups only, instead of constantly regenerating all the code.

Moreover, although dynamic dispatch gives greater flexibility in language reuse, it

might cause comprehension problems. Assume a DSL user knows concept *A* from language *LangA* and she knows that this concept has a certain behavior. Now, assume *LangA* is reused in *LangB*, and the behavior of concept *A* is overridden in *LangB*. This change might cause confusion for a DSL user familiar with both *LangA* and *LangB*.

5.1.6.2 Parallels with other approaches

The processing units in MetaMod retain the advantages of common OOP languages through the use of dynamic dispatch, the concept types hierarchy and the group types hierarchy. Processing units also bypass a few of the limitations of common OOP languages, e.g. the expression problem [168]. It does that by allowing the addition of both metamodel elements and operations to reused DSL units and multiple dynamic dispatch.

There were attempts in the the realm of general-purpose programming languages to bypass some of the limitations of common OOP languages in extensions of OOP languages, such as Java [27], or in more general proposals, such as aspect-oriented programming [74] and mixin class composition [18]. In comparison, we did the bypassing in a more specific programming context, that of developing DSLs. Developing a DSL is a special case of general-purpose programming, and we took that into account when developing the processing units, for instance, by relating a processing unit to a group and an aspect. Moreover, processing units having access to other processing units for reused groups and reused aspects, and the call resolution mechanism for multi-operations, are also a consequence of the specific context of developing DSLs.

At the same time, MetaMod clearly separates the data (the metamodel) and its behavior (the processing units). In many language workbenches the data is tied to the behavior. This is the view of common OOP implementations, where data and code are encapsulated together inside a class. We depart from this view, in that the behavior is dependent on the data, but the data can be reused in separation, without the behavior or with select behavior units (processing units).

For a better understanding of the mechanisms available in MetaMod, we also do a comparison with two kinds of inheritance, according to the classification of Meyer [104]. He presents model inheritance ("'is-a" relationships between abstractions in the model), and software inheritance (relations in the software itself, and not the model). The subtype relationship in the metamodel achieves the model inheritance, and reuse of processing units (that boils down to reuse of groups and aspects) achieves the software inheritance.

Moreover, we can draw a parallel between our approach and context-oriented programming (COP) [60]. The layers in COP can be compared to the processing units in MetaMod; more accurately, the layer corresponds to the aspect associated to the processing unit. Layers can be activated/deactivated, the same way aspects can be reused or not in certain processing units. On the other hand, behavioral variation is obtained with multi-operations in MetaMod. At call resolution time, the operation is chosen based on its name, group (context), concepts, and raw Java types.

Lastly, there is a parallel between the mechanisms of MetaMod and the multi-dimensional separation of concerns as described by Tarr et al. [142]. A processing unit of MetaMod, together with its associated aspect and group, can be compared to a dimension. The multiple dimensions are given by the multiple aspects implemented for a group. The composition of dimensions, on the other hand, is achieved through the reuse of processing units.

5.2 Features for language workbench requirements

We now take each requirement presented in Section 3.4 and we discuss how it is fulfilled in MetaMod. The requirements are defined for the type model level, so in the discussions in the following section we concentrate on this level. This section contributes to the answers to research questions RQ_2 and RQ_3.

> **RQ_2:** *How can we organize metamodels of the DSLs such that we facilitate modularity and reuse of DSLs?*

> **RQ_3:** *How can we organize processing units of the DSLs and the operations in the processing units such that we facilitate modularity and reuse of DSLs?*

As noted in Section 3.4, the requirements are related to the extended expression problem that was introduced by Zenger et al. [168], so we use the example provided in their technical report [168] throughout this section. The original example was written using data types and processors on the data types, and we make a one-to-one transformation into type models and processing units. The example is about expressions, in particular, the plus expression and the numeral expression.

We start with a simple *Base* type model that contains concepts *Exp* and *Num* (see Figure 5.12). A processing unit associated to this type model then defines an *eval* function on the expressions (see Figure 5.13). For this reason, the processing has aspect *Eval* associated to it. We subsequently extend *Base* with a new concept, *Plus*, and then we extend *Base* with a new operation, *show*. The extensions are going to be presented in the next subsections.

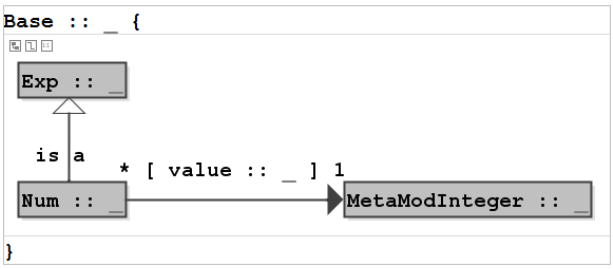


Figure 5.12: The *Base* type model.

5.2.1 When reusing a DSL unit, the meta-language allows a DSL engineer to add new elements (concepts and relations) and new processing operations on reused concepts to the reused version of that DSL unit.

This first requirement proposes that the extension of the DSL unit be possible both on the metamodel dimension and the processing unit dimension. We have presented the features for these additions in more detail in Section 5.1. A short summary follows. Assume that G_0 is a group, and that G_1 is a group that reuses G_0. The addition of data is enabled by the addition in G_1 of concepts and the addition of relations involving reused concepts from G_0. Then, the addition of new operations is achieved through defining new

```
PU_Base_Eval
for group Base
aspect Eval
       reuses << ... >>

@multi-operation
operation eval(GroupType#Base# inputGroup, ConceptType#Exp# exp) returns int {
  error "Override this operation!";
  return -1;
}

@override eval
operation eval(GroupType#Base# inputGroup, ConceptType#Num# num) returns int {
  return num.@src#value# in (inputGroup).first.intValue;
}
```

Figure 5.13: A processing unit defined for the type model *Base* and for the evaluation aspect.

processing units with operations on the new concepts in G_1 and on the reused concepts from G_0.

As an illustration of these additions, one can look at the two extensions of *Base*. The first extension defines a new *Plus* expression (see Figure 5.14). For the plus expression to be complete, we define an *eval* operation for it (see Figure 5.15).

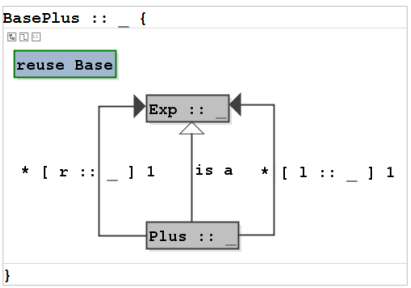


Figure 5.14: The *Base* type model extended with a *Plus* expression.

The second extension adds a new processing operation on the *Exp* type concept. This extension does not add anything to the type model (see Figure 5.16), but adds one new operation, *show* (see Figure 5.17). For that reason, the processing unit is defined for aspect *Show*.

Besides what has been shown possible by these two extensions (namely, the addition of new concept types, and new operations), the first requirement proposes that new relations be added to the reused concepts as well. An example of this addition was shown in the extension of *CanvasExtExt* with colors in Section 4.3.2. In this extension, we added colors to the shapes via a relation between the reused *Shape* and *Color*. In this new context, that of *CanvasExtExt*, shapes have also colors.

Non-breaking additions We are now going to informally demonstrate why the addition of relations to the reused concepts (that can exist both as source or as target of

```
PU_BasePlus_Eval
for group BasePlus
aspect Eval

@override eval
operation eval(GroupType#BasePlus# inputGroup, ConceptType#Plus# plus) returns int {
  return |Base| eval(inputGroup, plus.@src#l# in (inputGroup).first) +
      |Base| eval(inputGroup, plus.@src#r# in (inputGroup).first);
}
```

Figure 5.15: The *eval* operation defined for expression *Plus*. The notation |*GroupName*| represents the group name where the called operation is defined. This notation is available only in overriding operations, because one might want to call the current operation recursively, the multi-operation itself, or some other overriding operation, and they all have the same name.

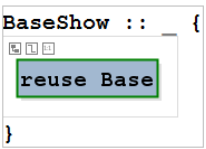


Figure 5.16: The *BaseShow* type model reuses *Base*, but does not add anything.

```
PU_BaseShow_Show
for group BaseShow
aspect Show
       reuses << ... >>

@multi-operation
operation show(GroupType#BaseShow# inputGroup, ConceptType#Exp# exp) returns string {
  error "Override this method!";
  return "";
}

@override show
operation show(GroupType#BaseShow# inputGroup, ConceptType#Num# num) returns string {
  return num.@src#value# in (inputGroup).first.strValue;
}
```

Figure 5.17: The *show* operation defined for expressions.

the relations) does not break operations defined originally for the reused concepts. We assume we do not need to explain the addition of concepts, because this is the common case in mainstream OOP languages, like Java, where this addition translates to adding completely new classes and subclasses of existing classes.

The main point of this informal demonstration is that, in general, we only make conservative additions to a reused DSL unit in a reusing context. That means that the additions one makes to a reused DSL unit should not break structural or user-defined constraints defined on the original reused DSL unit. There are two cases that could create problems. Firstly, one can add contradictory constraints to the reusing DSL unit. For instance, if a user-defined constraint of the reused DSL unit says that there should always be a path between a concept of type *CA* and a concept of type *CB*, but a constraint of

the reusing DSL unit says that there should never be a path between those two, clearly no value model will be able to satisfy both constraints. This way, the value model will not be valid, and the operations will not be executed on it. Secondly, if one needs to forbid a model element from the reused DSL unit in the new DSL unit, then there are two cases. If the forbidden element is a concept, then we write a constraint checking for the presence of that concept in the value models and we report an error if the concept occurs. If, on the other hand, the forbidden element is a relation, then the actions required from the DSL engineer depend on the cardinalities of the relation's ends. If the relation has zero as the lower bound at both ends, then the DSL engineer only writes a constraint checking that the relation does not appear in the value models. Otherwise, in the editor aspect, she needs to add a default value for the mandatory end of the relation.

5.2.2 The meta-language offers strong static type checking of the DSL implementations: no processing operation is applied on elements that it cannot handle.

The additions that were discussed in the previous requirement cannot occur ad hoc, but they need to be made under certain conditions. This second requirement imposes one condition, that the processing units are strongly statically type-checked.

MetaMod processing units have been implemented by reusing the Java implementation of MPS, called the base language. Operations in MetaMod are actually extensions of methods from the base language of MPS. Thus, everything that is allowed in a method in Java is allowed in an operation in MetaMod, except for the *this* and *super* references, because they do not have meaning for processing units. As for the type system, we use the type system rules defined for Java, and we add rules for the types and operations introduced by MetaMod. All the new types and type rules introduced by MetaMod, and their interplay with Java types and type rules enforce the strong static type-checking of the DSL implementations. For instance, an error is reported in Figure 5.18 when we try to call an operation with the wrong argument type.

```
@override eval
operation eval(GroupType#BasePlus# inputGroup, ConceptType#Plus# plus) returns int {
  return |Base| eval(inputGroup, inputGroup) +
      |Base| eval(inputGroup, plus.@src#r# in (inputGroup).first);
}
```

Error: type GroupType<BasePlus> is not a subtype of ConceptType#Exp#

Figure 5.18: Error reported for operation call with wrong argument type.

We used the facilities of MPS to make the interplay between the type system of Java and that of MetaMod. Firstly, MPS allows language concepts to be declared as types. We made the group type, $GroupType\#\langle GT\rangle\#$, and the concept type, $ConceptType\#\langle CT\rangle\#$, part of the type system, by declaring them as MPS types, and making them subtypes of the Java *Object* type. As a consequence, everywhere where Java objects are allowed, one can place group types and concept types, e.g. in collections and type declarations.

Secondly, MPS has a DSL for defining type system rules. The type system rules come in many forms, but we have used mostly 'type of' rules, 'supertypes of' rules and checking rules.

For a better understanding of how these rules function, we will give some examples from MetaMod. For instance, consider the type of function $\langle CT1\rangle.isTypeOf(\langle CT2\rangle)$ *in* $\langle VM\rangle$. The type of this operation (which is represented as a node in MPS) is boolean (see Figure

5.19 for the implementation in MPS), so we tell the type-checker that whenever it is asked the type of this operation, it should report boolean. Other times, determining the type of an operation needs more computations. For instance, the type of operation $\langle CV \rangle.@src\#\langle R \rangle\#$ *in* $\langle VM \rangle$ is a list type, where the type of an element in the list is the type concept that was defined as target of relation R in the type model of VM.

```
rule typeof_IsTypeOf {
  applicable for concept = IsTypeOf as isTypeOf
  overrides false

  do {
    typeof(isTypeOf) :==: <boolean>;
  }
}
```

Figure 5.19: The type system rule that says that the type of node *isTypeOf*, as implemented in MPS, is boolean. This syntax is specific to type system equations that MPS offers with the type system language.

Another kind of rule for the type system is the 'supertypes of' rule. With this rule, one can build a set containing types that are to be treated as supertypes of the type under discussion. For instance, in MetaMod, we take all the super-concepts of a concept C in a subtype relation in the metamodel, recursively, and make these the supertypes of C.

We have also used checking rules to verify the signature of overridden operations (they should always match a multi-operation) and the signature of constraint operations (they should always return a boolean), for instance. An example of incorrect signature for an overridden operation is shown in Figure 5.20.

```
@override eval
operation eval(GroupType#BasePlus# inputGroup, ConceptType#MetaModInteger# plus) returns int
  return |Base| eval(inputGroup, plus.@src#l# in (inputGroup).first) +
      |Base| eval(inputGroup, plus.@src#r# in (inputGroup).first);
}
```

Error: Parameters of the overridding method should be subtypes of the parameters of the overriden method!

Figure 5.20: Error reported in the signature of an overriding operation. The notation $|Base|$ suggests that the *eval* operation called is that from the *Base* group, viz. the multi-operation.

One point where both Java and MetaMod break the strong static typing is the down-casting operator. In MetaMod, the $\langle CV \rangle.castTo(\langle CT \rangle)$ *in* $\langle VM \rangle$ operator is a down-casting operator (the concept type of CV has to be a super-type of CT) and it can only check whether the concept type of CV is actually CT at runtime.

Another point where MetaMod breaks the strong static typing is in the multi-operations. The body of the multi-operation stands as the default case for when and overriding operation is not defined for a certain parameter type. Thus, MetaMod fulfills the strong static typing requirement only to a certain degree.

It is important to note that, before being type-safe, MetaMod also checks that value models comply to the structural and user-defined constraints. This is fulfilled by construction in MetaMod. A value model that breaks the structural constraints is going to be flagged as erroneous during typing. We did this by using the constraints aspect of MPS.

As for the user-defined constraints, in the current implementation, they are checked just before any processing unit code is run on the value models. Thus, no processing unit code is going to be run on a value model that breaks the user-defined constraints. MetaMod reports errors in the console if the value model breaks the user-defined constraints. In the future, we plan to do these checks while the user is typing.

Note that when fragment abstractions and applications are present in the value models, one needs to apply the fragment applications first, and then she can proceed with the writing of the processing units on the resulting value models. This is because the fragment abstraction extension was part of our exploratory process, and we did not integrate it smoothly will all features of MetaMod.

5.2.3 The meta-language should be such that reusing a DSL unit does not require to modify or to duplicate the reused DSL unit.

This third requirement enforces another condition on the way additions introduced by requirement one can be made. The condition is that language workbenches do not modify or duplicate the reused DSL units. We have two parts to the DSL units: the type models (groups) and the processing units. We will show that neither of these are modified or duplicated in the reusing DSL units.

In the case of the type model, at runtime, the algorithms of MetaMod look at concepts and relations recursively in the reused groups. The reused groups are unaware of the additions made to reused concepts in reusing groups. These additions are available only in groups where the reusing groups are present. Thus, the algorithms implemented in MetaMod know how to navigate the reused groups, and no duplication or modification of the original reused groups is needed.

In the case of the processing units, they only make calls to operations in reused processing units. As a result, there is no need to modify operations from reused processing units at the place of definition, or to duplicate those operations. Moreover, one can override operations from reused processing units if needed and then a call to such an operation is going to be redirected to the overriding operation.

5.2.4 The meta-language should be such that if you have a type-checked DSL unit for which code was generated, then reusing that DSL unit should not require type checking it again, and regenerating the code.

This fourth requirement enforces another condition on the way additions introduced by requirement one can be made. The condition is that no regeneration of code and no re-type checking of the reused DSL units occurs in the reusing context. There are two features of MetaMod that could raise problems here: reusing groups with uniques and the multi-operations.

Reusing other groups with uniques does not lead to changes neither in the generated code nor in the type-checking of the reused groups or its processing units. The equivalence classes for a given group, G, do not change when G is reused in some other groups. Moreover, in the generated code, the concepts are represented with their name appended to the name of the group they belong to; so, we always know to what equivalence class the concept belongs in the generated code (see Figure A.3 and *Rectangle.Rectangle*). When

running the generated code, MetaMod is always checking the equivalence classes of the top-most group[3], and the code generated by the reused groups contains concepts that the top-most group knows about. Note that the generated code here refers to the code generated by transforming the processing units into Java code (see Section 4.2.5).

As for multi-operations, where the algorithm places the code for the multi-operation counts for modularity. The multi-operation needs to know about all overriding operations. The solution we chose was to place a dispatching method in the MetaMod generated code of each group that contains a multi-operation in the associated processing units; the multi-operation could be defined in the processing units of the group itself, or the processing units of its reused groups.

Although no regeneration is needed for the reused DSL units, their source still needs to be present at runtime because generating code for the reusing DSL unit needs the reused groups and their processing units around so that it can collect information about the overriding operations, for instance.

Note that the current implementation of MetaMod does not have the separate generation implemented. Nonetheless, we are confident this is possible because of the informal discussion in this section and because of tests we have conducted on the generated code. In Solution *Tests*, model *separateGeneration* in the Github repository, one can find a two DSL units that we used to test that DSL units with uniques and multi-operations generate the same code when reused in different places.

5.2.5 The meta-language allows the combination of independently developed DSL extensions.

This fifth requirement enforces another condition on the way additions in requirement one can be made. The condition is that the combination of independently developed DSL extensions is possible. In MetaMod, this translates to combining groups in MetaMod and to rewriting operations for the combination in case of need. The amount of effort to glue the extensions depends on whether the extensions have conflicting features or not.

Let us go back to the example we started with in this section. In the discussion of the first requirement, we introduced two new extensions to *Base*, namely *BasePlus* and *BaseShow*. These two extensions are independent extensions. We combine the two extensions in *BasePlusShow* (see Figure 5.21). For this combination, we create an aspect, *EvalShow*, that reuses both the *Eval* and the *Show* aspects defined by previous processing units. Moreover, we add a *show* operation to expression *Plus* (see Figure 5.22). This was the only operation that was missing from the combination.

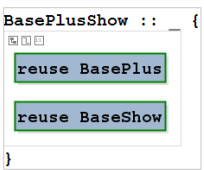


Figure 5.21: The *BasePlusShow* metamodel reuses the two independent extensions, *BasePlus* and *BaseShow*.

[3]The top-most group is the type model of the value model on which we run the generated code.

```
PU_BaseShowPlus_EvalShow
for group BasePlusShow
aspect EvalShow
       reuses Eval
              Show

@override show
operation show(GroupType#BasePlusShow# inputGroup, ConceptType#Plus# plus) returns string {
  return |BaseShow| show(inputGroup, plus.@src#l# in (inputGroup).first) + "+" +
      |BaseShow| show(inputGroup, plus.@src#r# in (inputGroup).first);
}
```

Figure 5.22: The *EvalShow* aspect with the *show* operation defined for expressions.

Note that each extension can do their own augmentation, independent of each other (they do not interfere). They start interfering only once they are combined.

When combining two extensions, the same overriding operation can exist in both extensions. By default, the overridden operation residing in the group with the smallest lexicographical name is given priority, but one can decide to override the operation in the combination, if the default behavior is not desired.

Another issue to take care of during combinations is the different aspects defined for the different extensions from the combination. To make use of all the aspects from the extensions, the combination needs to reuse all these aspects in the processing unit for the main aspect.

5.3 Related work

In this section we look at features for modularity and reuse in a handful of other language workbenches. We skip MPS as it was already discussed in Section 2.4. Moreover, we also look at systems used in the grammar world, and especially at systems from the attribute grammar world. We do so because they have relevant extension mechanisms for the grammars and the compilers.

5.3.1 Language workbenches

For the next three tools, we are not going to make a direct comparison because we do not have enough information on them. Nonetheless, the available information is relevant enough to include it here.

Intentional Software Intentional Software [63], a continuation of Intentional Programming [130, 131], is among the first creators of a language workbench. The code is closed source and there is not much information available. From the few resources we could reach [63, 157], we could not deduce all the internal workings. The structure is described slightly differently in Intentional Software. The relations between concepts are not part of the concept itself, which is also the case in MetaMod, but unlike in MetaMod, they also allow defining operations on the relations, and the relation can be reused together with the operations defined on them. Moreover, they also define operations for references, which allows references to have different projections, for instance. On the other hand, adding new constructs to a reused language in a reusing context seems to be easily achievable.

Cedalion Cedalion is a language for LOP [92, 93]. It is not a traditional language workbench per se, because it enables the creation of internal DSLs, but it brings features from language workbenches, such as structure, projection, static semantics and type system definitions. These are additional to the support for dynamic semantics that comes from the host language itself, Cedalion. Cedalion is built on top of Prolog and it is itself a logic programming language with the extra facilities for DSL creation mentioned above. In terms of modularity and reusability, there seems to be a tight connection between the structure of the language and all its other aspects (projection, type checking, static semantics, dynamic semantics, etc.). This hinders reuse because all the language aspects of a DSL need to be reused, regardless of the needs of the reusing context. Thus, one cannot use the structure together with only a selection of the language aspects, they come as a package. Moreover, from the examples provided in the literature, it is also not clear how can one modify the behavior of a reused language. On the other hand, extending a language with new constructs seems to be easily achievable [94].

Whole Platform Whole Platform [132] is another language workbench, but although not closed source, there is not much documentation available online. It has an interesting mechanism, called foreign types relations where one can define relations among types of the current language and types of other languages. These relations can be equivalence or subtyping relations. It is not clear though in what ways the behavior of the external languages can be modified or added to.

In the following paragraphs, we are going to describe a few of the most know language workbenches from the perspective of features for modularity and reuse.

Xtext Xtext [15] is a textual language workbench based on EMF. Referencing another language in Xtext requires importing the metamodel of that other language. This enables the DSL engineer to reference the metaclasses (type concepts) and to use them as base types. On the other hand, extending another language in Xtext allows one to override a grammar rule (concrete and abstract syntax) of a reused metaclass. However, it does not allow to add to a reused concept, but only to override it completely. Xtext also leverages the ANTLR parser [117], which means that during the composition of languages extra care needs to be taken for grammar ambiguities (or it can even be impossible to make the combination). Moreover, a very restrictive feature in Xtext is that it can only extend one other language. This can be very limiting in practice as, for instance, the mbeddr ecosystem of languages could not be built in Xtext [157]. As for the dynamic semantics of the language, this can be specified using other languages, e.g., Xtend [15], ATL [69], or ETL [80]. Thus, modularity and reuse at the operations depends on the features of these languages.

MetaEdit+ MetaEdit+ [72] is a commerical graphical language workbench for domain-specific modeling. In their book on domain-specific modeling [73], the authors of MetaEdit+ specify that languages can be integrated either through model transformations, or at the language level. At the language level, modeling languages can be integrated with shared or linked modeling constructs. For instance, a data concept can be used on one hand to specify a workflow model and, on the other hand, to specify the database schema. Thus, concept reuse among different DSLs is possible. On the other hand, it is not clear to what extent code generation reuse is possible. Code generation is done with the help of a template language, and we think that template languages are not fit

for modularity, because one has to specify an entire entity from the target language, thus limiting the modularity of the DSL generation to the modularity available in the target language. Nonetheless, one interesting aspect of MetaEdit+ is that every DSL created is an extension of or an addition to the language workbench itself [47]. This probably gives the possibility to reuse many of the language aspects that are already defined in the language workbench. Moreover, MetaEdit+ does not use parsing, allowing arbitrary notations to be combined.

Spoofax Spoofax is a language workbench that uses declarative DSLs to define all language aspects [70]. Each DSL has its own modularity and reuse mechanism. For instance, the abstract and concrete syntax modules, called the SDF3 (formerly SDF2) modules, can import other modules for reuse or separation of concerns. Moreover, a module may extend the definition of a non-terminal in another module. This is equivalent to adding new relations to a reused concept in MetaMod in the context of the reusing group. The dynamic semantics of a language, defined with DynSym (formerly Stratego) makes use of rewriting rules. For modularity reasons, these are not pure rewriting rules, but they have been extended with dynamic rules (to make the rules context-aware) and programmable rewriting strategies (to control the application of rewritings). Although being able to control the order of rewriting rules using strategies offers flexibility, it can be tedious to define such strategies (for instance, defining the order for rewrite rules of all the sub-concepts of the expression concept).

What is interesting to note is that both in the case of Xtext and Spoofax, one can delegate modularity to the generated target language. That is, one can use Java classes, inheritance and overriding to provide modularity to the DSLs themselves [157].

Monticore Monticore [83, 84] is a textual language workbench. Its main modularity mechanisms are multiple language inheritance and language embedding that are assisted by the parser, editor and tree traversal algorithms. Inheritance allows incremental changes to a language and embedding allows to combine different language fragments. Monticore is based on a grammar definition language that allows defining a concrete textual syntax as well as the abstract syntax of a language. On language inheritance, the extending grammar defines only the differences between the original language and the new one. Differences consist in adding new productions and overriding existing productions in the original language. This way, algorithms written for the original language work also for the new language, with the exception of the operations for the overridden productions, which need to be modified. They also introduce the notion of language interface, that can be implemented by a concrete production in a sublanguage. One drawback is caused by the lexical analysis that restricts the kinds of sublanguages one can build for a language. On language embedding, the lexer and parser are compiled separately per language, and a super-ordinated lexer/parser delegates the task to the adequate language. To combine languages, a special DSL is used for the configuration of the language components and their communicating rules. One more interesting feature of Monticore is that it adds associations in the grammar definition language, thus bringing it closer to the metamodeling world, but also making it more difficult to establish links among objects after parsing. Monticore also takes care of the modular development of domain-specific tools associated to the languages. For the combination of languages, modular visitors are employed where the visitors invoke the different methods automatically. For language inheritance, the different visitors can be subtyped to change the behavior of newly added or overwritten productions.

Melange Another recent work is that of Deguele et al. on Melange [35], a meta-language for building DSLs by assembling and customizing legacy DSL artifacts. Melange uses EMF to define the metamodel and Kermeta 3 (K3) [37] to define the operational semantics. In Melange, the focus lies on reusing legacy artifacts. Their work is based on typing relations and defines a set of operators for language assembly and a set of operators for language customization (extensions and restrictions). In Melange, a language is defined by abstract syntax, semantics and a model type (this can be inferred automatically). A model type is a structural interface over the metamodel of a language, and it itself takes the form of a metamodel. Most operators that Melange introduces are targeted at reusing either the metamodel or the semantics as is, plus glue code. The language inheritance operator, on the other hand, is the only one meant to reuse and adapt the original language.The inheriting metamodel can modify the inherited metamodel to the extent to which the structural interface is not violated. It is also not clear what operations from the inherited language can be overridden in the inheriting language. This probably depends on Kermeta, the language used to define operational semantics for Melange.

Kermeta Kermeta [37,67] is a language that can specify operational semantics or translational semantics for Ecore metamodels. Kermeta makes up for the lack of modularity in Ecore, by allowing the addition of features (attributes and references) and methods to the classes of the Ecore metamodels via aspects. This does not solve the modularity problem of Ecore, but makes up for it outside of the modeling formalism. Moreover, additions to the metamodels are not made using the Ecore files, but using Kermeta files instead, thus mixing concerns in the Kermeta files (additions of both structural and behavioral aspects). Kermeta, together with two other languages, one for defining ecore metamodels, and one for defining OCL constraints, forms the Kermeta language workbench. In this language workbench, a DSL implementation requires an abstract syntax (via Ecore), a static semantics (via OCL), and a behavioral semantics (via Kermeta language). As in MetaMod, in Kermeta, one can create variants of a language, by mashing up different behavioral and static semantics with a given ecore file. An advantage of this approach is that Kermeta is inter-operable with many other tools built for the Ecore ecosystem (e.g., Xtext for defining the textual syntax of a language, and Sirius for defining the graphical syntax of a language).

Neverlang Neverlang [147] is a framework for language development. A language component is a self-contained bundle where syntax is put in relation to different evaluation phases (processing units in our terminology). An evaluation phase has several features (operations in our terminology). A feature can also be associated with a placeholder, meaning that the feature does not have an implementation in this language component. This creates an unsatisfied dependency in the language component. A language implementation is a set of language components where all these dependencies are satisfied. In Neverlang, the way to extend a language is by providing a placeholder in the extended language and mapping it to a feature in another language component. Moreover, the underlying execution semantics of Neverlang is based on a modular visitor pattern. This allows extension both on the processing phase dimension and on the language constructs dimension. The focus in Neverlang is on the composition of languages, where variation points in the language are defined in the language component itself.

Rascal Rascal is advertised as a DSL for meta-programming, but it can be considered a language workbench as well, because it can deal with all aspects of language design and implementation [12]. Rascal is the continuation of the ASF+SDF Meta-Environment [148]. It supports extensible concrete syntax (defined with grammars), abstract syntax (defined with abstract data types) and operations on these concrete and abstract syntax trees (defined with functions, which are rewrite rules in disguise) [12]. The definition of any language aspect is made in a module in Rascal and there are two ways to reuse a module in Rascal: import and extend. One interesting aspect is that modules are lexically closed. A consequence of this is that when one imports module B into module A, although operations in module B can be called in module A, nesting of data structures from A and B does not work well when calling operations, because the operations in B are treated locally, without knowledge of operations and structures in A. This can be solved by extending the module, instead of importing it. For grammars and ADTs, the contents of the extending and extended modules are simply merged. Because Rascal uses a scannerless GLL algorithm for parsing [126], ambiguities have to be dealt with when extending grammars, and Rascal allows that with the help of a list of reserved keywords. Extending functions, on the other hand, relies on pattern-based dispatch, where a function can be provided for each case in an algebraic data type. Interestingly, they have default function definitions in Rascal that behave as a catch all rule when no other rules are applicable. Another goal of the default function definition is to signal that these can be overridden in extending languages. Thus, later extensions of languages have to be anticipated by making functions default.

5.3.2 Extensible languages and compilers

The following tools are from the grammar world and are related to extensibility.

JastAdd JastAdd [43] is a meta-compilation system that supports extending compilers and related tools, e.g., specialized analyzers and transformation tools. The modularization and extensibility of languages in JastAdd are enabled by three main ingredients: object-orientation, static aspects and declarative computations. The object-orientation aspect comes from the object-oriented abstract grammar that generates a Java class hierarchy with deep class hierarchies, late bound methods and reuse through inheritance. The static aspect, on the other hand, comes from no behavior declared in the abstract grammar, but outside, in behavior modules. These modules can contain both new equations, but also new attributes for the AST node classes. Finally, the declarative computations come from the behavior modules being declared in a declarative way. In comparison, in MetaMod, we can add to the structure of the reused concepts (their equivalent is AST node classes); that is, we can add relations to concepts in new contexts (new containing groups), and not only subtype concepts in new contexts.

SugarJ SugarJ [45] integrates syntactic extensibility into the libraries; libraries being the main extension mechanism of programming languages. SugarJ involves extending the grammar of the base language and defining a transformation of SugarJ nodes into the base-language AST. The extensibility power of SugarJ relies, to a large extent, on the extensibility power of SDF and Stratego, because it uses SDF and Stratego underneath, and thus, all the comments that we made in the paragraph with Spoofax hold for SugarJ as well. This work was generalized with Sugar* [46], where the base language can be an arbitrary language, and not necessarily Java.

AbleJ AbleJ [152] is an extensible language framework that allows importing domain-specific extensions into an extensible implementation of Java 1.4. The tool supports the modular specification of composable language extensions. The extensible language is specified as a complete attribute grammar and language extensions are specified as attribute grammar fragments. If they fulfill a few restrictions, language extensions in AbleJ can be designed to be composable with other language extensions without any intervention from the DSL user side. This restricts the types of language extensions to a certain extent. The extensibility of the attribute grammar specification, done with Silver [150], is enabled by forwarding [151], collection attributes [17], pattern matching and aspect productions [150]. One downside of AbleJ is that they combine the concrete syntax, the abstract syntax and processing operations in the same module, which means that the language cannot be reused with select processing operations in other projects.

LISA In the realm of attribute grammars, a similar line of work is that of Mernik et al. in LISA [100–102] that described multiple inheritance and templates in attribute grammars. An attribute grammar is composed of a grammar, a finite set of attributes and a finite set of semantic rules. LISA (*L*anguage *I*mplementation *S*ystem using *A*ttribute grammars) is a language description system for language composition. Mernik [100] is using similar concepts to those in general software object-oriented programming to define modularity in grammars. He defines inheritance in attribute grammars as inheritance of lexical, syntax and semantic specifications.

Polyglot Another approach is that of Polyglot [110], which is an extensible compiler framework that supports the creation of compilers for languages similar to Java. They use various object-oriented design patterns in the traversal of the extended AST, such as abstract factories, delegation and proxies to increase the extensibility of the framework.

5.3.3 Discussion of language workbench requirements fulfillment for language workbenches in related work

Some of the language workbenches discussed in Section 5.3.1 do not allow the addition of relations involving a reused concept as a source to the reusing or extending languages. In few cases, this restriction happens because of the way one generates code from the metamodels. For instance, if one generates a Java class per concept and adds the relations with the concept as a source in the fields of this class, then, an extending context cannot add fields anymore to this class. Language workbenches that fulfill this requirement are Rascal, Spoofax, Kermeta and Monticore.

To the best of our knowledge, the discussed tools do fulfill the second requirement related to strong static typing, and the third requirement related to the modification or the duplication of the reused DSL units. As for the fourth requirement, that deals with the regeneration of code, only Neverlang and MPS seem to fulfill it. The process used to generate code is even more important for this requirement. For instance, Kermeta generates one Scala trait per class for the mashup of the static semantics and that of the behavioral semantics. This means that the addition of new methods and new constraints to a class will require the regeneration of the Scala trait.

Finally, the fifth requirement is again fulfilled by most language workbenches, with the exception of Xtext, that does not allow the extension of more than one language.

5.4 Conclusions

In this chapter, we have presented features of MetaMod from two different perspectives: that of modularity and reuse, and that of fulfilling the language workbench requirements we discussed in Chapter 3. Through these two descriptions, we gave a complete picture of MetaMod. The features for modularity and reuse were showcased on a non-trivial expression language, that is a reimplementation of part of an expression language from MPS. This also allowed us to make an extensive comparison between MetaMod and MPS. At the same time, we showed how are the language workbench requirements fulfilled with the help of more or less the same features discussed for modularity and reuse. This allowed one to see the same features from yet another angle.

One can notice that the features we discussed in the context of modularity and reuse fulfill other roles as well. For instance, multi-operation and overriding could be considered as control abstractions. Furthermore, the metamodel elements could be considered as data abstractions. In this chapter, and in the first section in particular, we looked at all these abstractions using modularity and reuse as lenses.

The contributions we have made in this section are manifold. Firstly, we have leveraged two hierarchies created by the metamodel, that of the group reuse and that of concept subtyping. They are used in the type system of the processing units. Secondly, we have introduced multiple dynamic dispatch with an unique operation call resolution (that sorts the concept types, the Java raw types, and the groups types; thus, we consider the characteristics of DSLs). Thirdly, we have made a combination of declarative and OO with the processing units and the operations that they contain (reuse of operations among processing units, no state, multi-operations). The usability and the usefulness of the features for MetaMod was demonstrated through the expression language reimplementation and the informal demonstration that the language workbench requirements hold with MetaMod. Finally, we made an extensive comparison to MPS with the help of the expression language, and we have presented an extensive list of language workbenches, with their modularity and reuse features.

Chapter 6

Modularity of value models

In today's landscape of more and more software-driven functionalities, spanning more and more fields, model-driven engineering (MDE) promises to ease the development of software. To accomplish this goal, MDE employs domain-specific languages (DSLs). The problem is that, on one hand, DSLs are not easy to create, and, on the other hand, as a result of the increased software-driven functionalities, they need to deal with bigger value models. In dealing with these big value models, modularity mechanisms at the value model level are employed regularly by DSLs. These mechanisms need to be introduced over and over again into the developed DSLs, adding to the effort of creating them and to the effort of learning them (especially when DSL users have to deal with multiple DSLs). To ease the development of DSLs, we propose to introduce a modularization of value models that is independent of the DSLs. MetaMod offers two mechanisms, viz. groups and fragment abstractions, that can be used for modularity in value models.

6.1 Introduction

With the increased use of software in all fields of business, the need for automating software construction increases. In the context of today's software development challenges, model-driven engineering promises to bring improvements. MDE raises the level of abstraction in programming languages and brings them closer to the domain of operation. In MDE, domain-specific languages - computer programming languages, usually declarative, with limited expressiveness and focused on a particular domain - are key players.

One drawback of DSLs is that creating them requires a significant amount of work. Moreover, the large value models that DSL users need to deal with nowadays complicate the creation of DSLs even more. That is so because DSL engineers need to cater for these big value models by introducing modularity mechanisms into the DSLs themselves. For instance, if one needs a concept for grouping in the value models, this needs to be defined in the metamodel. Sometimes, the same modularity mechanisms need to be added to different DSLs over and over again (it might be possible to reduce this burden via reuse on the level of type models). This adds to the effort of designing and implementing the

DSLs considerably.

Moreover, the problem of repeatedly adding modularity mechanisms for value models in the DSLs was also identified by people in industry that we talked to at the beginning of the Crystal project[1]. They told us that, most of the times, they need to add at least a macro system to their domain-specific languages in order to deal with repetitive structures and with large value models.

That is why we looked at how modularity exists at the value model level. There are two main types of modularity that we encountered at the value model level in the DSLs that we worked with.

The most basic type of modularity for value models is simple grouping. Taking a look at any programming language, one can see them from a grouping perspective: classes are grouped into packages, properties and methods are grouped into classes, statements are grouped into blocks or methods, formal and actual parameters of methods are grouped into enclosing parentheses, variable declarations of the same type can be grouped together and so on. We call this type of modularity, *modularity-in-the-small*. It concerns a value model or a value model fragment that exists in the same location. This type of modularity has as goal increasing the understandability of the models.

Another basic type of modularity is related to some form of importing existing value models. This also includes the mechanism of libraries. This type of modularity can have many flavors depending on the strategy used to handle the namespaces of the value models. The strategies employed to resolve various kinds of conflicts between the importing and imported value models are diverse. For instance, one can choose to keep the namespaces of the value models separate, or to merge them based on names, or to explicitly disambiguate the use of any ambiguous name, etc. We are going to call this *modularity-in-the-large*. This type of modularity has as goals increasing the understandability of models, reducing the time to construct the models, reducing the size of models (when the alternative is incorporating the copy) and reducing maintenance effort (change only reused module, rather than all copies).

That is why we introduced generic mechanisms that cover these two common modularity use cases and that can be used in value models of newly developed DSLs. They considerably ease the burden of the language developers because they do not need to add constructs in the metamodels of the DSLs for these forms of modularity on value models anymore. Moreover, tooling can provide different kinds of syntactic sugar for these mechanisms to help DSL users. This chapter thus deals with research question RQ_5.

> **RQ_5:** *What modularity and reuse mechanisms can be applied to models, irrespective of the DSL?*

Of course, we do not cover the modularity mechanisms that need semantic information. If DSLs need special rules for the scoping of groups, for instance, the DSL engineer will need to implement the modularity mechanism herself. Moreover, if the DSL users need more complex modularity mechanisms in their value models, they would need to provide it themselves.

The next sections cover the implementation of a DSL, called Kaja, in Jetbrains MPS and that in MetaMod, followed by a discussion of these implementations. Moreover, we then present related work and we conclude the chapter.

[1]As can be seen on the first page of the thesis, this work was partially sponsored by the European Union project named Crystal.

6.2 The Kaja DSL - Jetbrains MPS implementation

Kaja is one of the DSLs in the samples project that comes with an installation of Jetbrains MPS. It was implemented by the MPS team to highlight features of MPS. In this section we discuss the Kaja DSL that comes with MPS 3.2 from April 9, 2015.

The Kaja DSL is a DSL for specifying the route of a robot through a grid where there are walls and marks. The dimension of the grid is fixed and the grid has exterior walls by default, but the walls inside the grid and marks are introduced by the DSL user. The DSL user also introduces the journey of the robot through the grid. Thus, the Kaja DSL consists of commands to build the layout of a grid of cells and place marks in this grid, which we call *grid commands*. The grid commands consist of building and destroying walls at given coordinates in the grid, and of dropping and picking marks at given coordinates in the grid. The Kaja DSL also consists of commands to specify actions that a robot can do in the grid, which we call *robot commands*. The robot commands consist of stepping, dropping marks, picking marks and turning to the left. Besides the grid and robot commands, there are also *meta-commands* for repeat statements, while loops, if statements, and trace messages. The meta-commands are accompanied by logical expressions that check whether a cell is full, it has a mark, it has a wall, or the robot is heading in a certain direction. Examples of grid commands, robot commands and meta-commands, can be seen in Figure 6.1.

For complicated routes and actions through the grid, the size of Kaja value models can clearly increase to a point where a flat value model is barely understandable. The DSL, thus, requires some support for modularity of the value models. The DSL implementers of Kaja have added two special instructions for that. The first addition are *routines*, where the DSL users can encapsulate a list of commands. A routine can be called in the value model, and its commands are expanded at the calling place. The second addition are *libraries*. Libraries are collections of routines that can be imported in value models to make the routines available in the value models. Examples of routine calls and definitions, and a library definition, can be seen in Figure 6.1.

Adding support for routines and libraries to a DSL is not trivial. This clearly increases the amount of work needed to implement the DSL. There are structure, editor, constraints, data flow and code generation aspects defined for routine calls, routine definitions and libraries.

In the next section we show how this kind of routines and libraries are offered to value models of MetaMod, irrespective of the DSLs.

6.3 The Kaja DSL - MetaMod implementation

The implementation of the Kaja DSL in MetaMod did not require introducing constructs for defining and calling routines, or for defining and using libraries. This was possible because MetaMod offers the same modularity mechanism at the value model level as at the type model level. This is enabled by the meta-metamodel of MetaMod, that defines both type models and value models.

As a short reminder, the modularity mechanism provided in MetaMod is based on groups and parameterized groups. Groups are meant to organize model elements and due to their semantics, they are self-sufficient. Parameterized groups (also called fragment abstractions) are a generalization of the groups and are based on model element substitution. One novel aspect in fragment abstractions is the fact that we leveraged

```
Library PlaygroundDefinition defines
  routine buildSimplePlayground means
    build wall at row: 1 col: 4
    build wall at row: 2 col: 4
    drop mark at row: 4 col: 3
  end
  routine buildMaze means
    build wall at row: 4 col: 1
    build wall at row: 4 col: 2
    build wall at row: 4 col: 4
    build wall at row: 4 col: 5
    build wall at row: 5 col: 5
    build wall at row: 6 col: 5
    build wall at row: 7 col: 5
    build wall at row: 8 col: 5
    build wall at row: 8 col: 4
    build wall at row: 8 col: 3
    build wall at row: 8 col: 2
    build wall at row: 8 col: 1
    drop mark at row: 5 col: 1
  end
end
```

(a) A library with two routines that define grid commands

```
Script Maze runs as
  require PlaygroundDefinition
  buildMaze Library call

  # Definitions
  routine findDoor means
    while wall ahead do
      turnLeft
      step
      turnRight Library call
    end
  end

  routine sniffAround means
    while (not mark and not wall ahead) do
      step
    end
    if wall ahead do
      turnRight Library call
      sniffAround
    end else do
      pick
      trace -> Found a mark <-
    end
  end
end
```

(b) A script with library imports (the *require* instruction), routine calls and routine definitions.

Figure 6.1: Examples of grid commands, robot commands, meta-commands, routines and libraries. The syntax in this figure is the syntax defined for the Kaja DSL by the MPS team.

the substitution mechanism in untyped lambda calculus by combining untyped lambda calculus with modeling elements, as explained in Section 4.12.

Going back to the Kaja DSL, we implemented the grid commands, the robot commands and the meta-commands. On the other hand, we did not need to implement the routine definitions, the routine calls and the libraries. That is because routine definitions can be replaced by fragment abstractions. Even more, the fragment abstractions are more powerful than the routines in Kaja, because one can also have placeholders in fragment abstractions. On the other hand, routines in Kaja do not have parameters. Then, routine calls can be replaced by fragment applications. Finally, libraries can be declared in separate groups and reused in the current group.

Due to the default syntax of MetaMod (either the textual or the visual one), that is more verbose than the custom syntax for Kaja developed in MPS, we show a part of the playground definition and a part of the maze script defined in Figure 6.1. These two parts are shown in Figure 6.2 and Figure 6.3.

```
buildSimplePlayground = Λ commandList .
  [type model Kaja

   reuse Numbers

   gen bW14 :: buildWall
   gridCommand commandList commandList command bW14 row 1 col 4
   gen bW24 :: buildWall
   gridCommand commandList commandList command bW24 row 2 col 4
   gen dM43 :: dropMark
   gridCommand commandList commandList command dM43 row 4 col 3

   bW24 [----------] bW14
        [pred     ]
   dM43 [----------] bW24
        [pred     ]]
```

Figure 6.2: Fragment abstraction for building a simple playground that given a command list, places walls in two cells and drops an mark in another cell. The grid commands are built with an application of *gridCommand*. Moreover, we explicitly encode the order of the commands with the *pred* relation.

```
Maze :: Kaja group {
  reuse PlaygroundDefinition

  MazeScript :: Script
  RobotCL :: CommandList
  BuilderCL :: CommandList

  MazeScript [----------    ] RobotCL
             [robotCommands ]
  MazeScript [----------    ] BuilderCL
             [buildCommands ]

  buildSimplePlayground commandList buildCommands
  findDoor commandList RobotCL
}
```

Figure 6.3: Value model of Kaja that builds a simple playground and that tells the robot to move around according to commands defined in *findDoor*. A script for Kaja in MetaMod is made of a command list for grid commands and a command list for robot commands.

6.4 Discussion

The modularity mechanisms offered to value models by the meta-metamodel itself in MetaMod can decrease the development time of new DSLs. Elements such as routines, libraries and strategies to handle the namespaces are not trivial to get right in a DSL. However, there are also limitations to this mechanism. If a different conflict resolution strategy, or a different scoping strategy are needed for the modularity mechanisms of the new DSLs, then the DSLs need to introduce these modularity mechanisms themselves. Nonetheless, DSL engineers could piggyback these modularity mechanisms on top of (parameterized) groups. So, not everything needs to be defined from scratch.

As already mentioned, in MetaMod, DSL engineers cannot create their own custom syntax. Nonetheless, the fragment abstraction and fragment application elements give programs a custom look-and-feel. For instance, one can look at the fragment application

for grid commands in Figure 6.2. The name of the lambda application followed by a grouping with placeholder name and actual argument create a domain specific look. Of course, if the name of the fragment abstraction and those of the placeholders are not suggestive, then there is no such custom look-and-feel.

Modularity at the value model does not reflect in the API functions for querying value models, because MetaMod treats reused value models as if they would be defined in the value model itself. Moreover, MetaMod treats fragment applications as if their result would be defined in the value model itself. That means, when querying for concepts and relations, the concepts and relations from the reduction of fragment applications and those from reused models are returned together with the concepts and relations defined directly in the value model. To solve this, we could introduce API functions that return reused value models, fragment abstractions and fragment applications. This would also allow customization of the modularity mechanism for the value models of the newly built DSLs. This is part of future work (see Chapter 10.3).

One relevant question is whether when introducing custom syntax for DSLs, the syntax of the grouping mechanism and that of fragment abstractions will be satisfactory. That is, will the syntax of the grouping mechanism and that of the fragment abstractions be in line with the custom syntax of the DSL? One way we could tackle this is to redefine the editor for the grouping mechanism and the fragment abstraction mechanism for the purpose of the newly built DSL.

6.5 Related work

The related work discussed in Section 4.1.6 is also relevant for this section. Here we present in more detail only three approaches, that we found most relevant.

Fragmenta [5] aims to modularize models by metamodel-defined fragmentation. In this theory, a model is a container of clusters, which, in turn, are containers of other clusters and fragments. Fragments can reference elements from other fragments of the same model by means of proxies and according to the fragmentation strategy in the metamodel. The way elements from other fragments are referenced is dictated by the type of design chosen in the strategy for the metamodel: top-down (continuation) or bottom-up (importing). To create the overall model, Fragmenta has two composition operators: one based on set-union composition (merges fragments without resolving the proxies) and colimit composition (merges fragments by resolving the proxies). For the composition to be free of inheritance cycles, there are local fragment constraints that need to hold. One of these constraints is that proxies cannot inherit. If we make a parallel to our approach, we can equate fragments to groups, clusters to groups, and proxies to concepts with the same name and conforming type concept. Moreover, when fragmenting a type model, we do not impose any restrictions on the equivalent notion of a proxy. In case of an inheritance cycle, MetaMod will report an error to the DSL engineer. Moreover, we do not prescribe the fragmentation strategy at the type level, and we let the DSL users decide on how they want to organize their value models.

There is an entire class of related work on aspect-oriented modeling (AOM) [2]. One work in the realm of AOM is that of Heidenreich et al. [56], where they discuss modularization techniques for arbitrary domain-specific languages. They based their work on the Invasive Software Composition approach [7] and implemented it in Reuseware [59]. In this work, they provide fragment components with reference points and variation points. These points give rise to two pairs: “hook-prototype” for the variation point and

its replacing fragment, and "slot-anchor" for the reference point and its bound fragment. Moreover, a fragment composition interface defines a set of ports that are linked to reference points and variation points. The fragment interfaces are defined by the fragment developer, while a fragment user defines fragment composition programs. An optional step is to extend the metamodel to specify only a select part of the original metamodel elements as valid variation and reference points, thus restricting which compositions are possible at the value model level.

Note that the previous two related works address the issue of modularity for value models outside of the formalisms themselves. Both of them build on top of several formalisms, one of them being Ecore [135].

Jetbrains MPS offers default facilities to structure the programs of a language. A program written using MPS languages is called a model. The model is a collection of root nodes. The model has also meta-information, such as imported models and the languages that the root nodes of the model conform to. Furthermore, the models are bundled into a module. A special kind of module is the solution. Models can be imported from the current solution, or from other solutions as well. Moreover, DSL engineers are given API functions that allow them access to imported models, so that during the generation, for instance, they are aware of the structure. Note that, in MPS, languages can be composed at runtime as well. One downside of this is that the DSL user might need to specify an ordering for the generators of the languages involved in the model, thus making them aware of the implementation details.

6.6 Conclusions

The biggest contribution of this chapter was to showcase that MetaMod and its meta-metamodel offers modularity mechanisms for the value model level. We have seen from the Kaja example, that in the MetaMod implementation of Kaja we did not need to introduce elements such as routines and libraries. We think that many simple DSLs do not need more advanced modularity mechanisms than these ones.

These modularity mechanisms at the value model level, provided by the meta-metamodel itself, lead to a decrease in the development time of a DSL and also to more robust DSLs, because these modularity mechanisms are well tested by many other DSLs. Furthermore, having the same modularity mechanisms in multiple DSLs makes these DSLs easier to learn as well (it is often the case that DSL users need to work with several DSLs). On the other hand, more advanced modularity mechanisms still need to be introduced by the DSL users in the DSLs.

Chapter 7

Reuse mappings

One of the impediments to the wide adoption of DSLs is their high cost of design, implementation and maintenance. Some of these impediments can be alleviated by reusing language aspects of previously developed DSLs. We propose a mechanism whereby operations (for interpreters, model transformations, etc.) defined on a base DSL unit can be reused in structurally dissimilar DSL units. Our mechanism relies on the observation that these operations have in common a means of querying models. Once a query on the base DSL unit can be expressed in terms of queries on structurally dissimilar DSL units, the operations become reusable in the dissimilar DSL units. We enable this with our mechanism called reuse mapping. To demonstrate the feasibility of our ideas, we have implemented the mechanism of reuse mapping in MetaMod. In addition, we also discuss how the mechanism of reuse mapping can be supported by other tools.

7.1 Introduction

One of the biggest problems of DSLs is the high cost of designing, implementing and maintaining a DSL [149]. Each DSL is designed with a certain purpose and with certain users in mind, and so, even in the same domain, there can be a lot of variations among the DSLs and their underlying metamodels. That can be seen in the collection of metamodels existing on AtlanMod Zoo[1], where different metamodels exist for the same problem domain (e.g., 11 different Petri net metamodels) [121]. Although logically extending each other, the corresponding Petri net metamodel parts differ to a large extent. This leads to the reimplementation of the operations associated to these metamodel parts, because they depend heavily on the metamodel. Consider the examples in Figure 7.1 that are taken from the AtlanMod Zoo. In the Petri net metamodel in the upper part of the figure, the arcs between transitions and places are represented through relations, while in the Petri net metamodel in the lower part of the figure, they are being represented through explicit arc concepts. Although it contains more details (the arc weight, for instance),

[1] `http://web.emn.fr/x-info/atlanmod/index.php?title=Zoos` - website with a collection of metamodels used in different languages

the lower-part metamodel logically embeds the upper-part metamodel. However, the differences between the two metamodels make it difficult to reuse the operations defined on the upper-part DSL in the lower-part DSL.

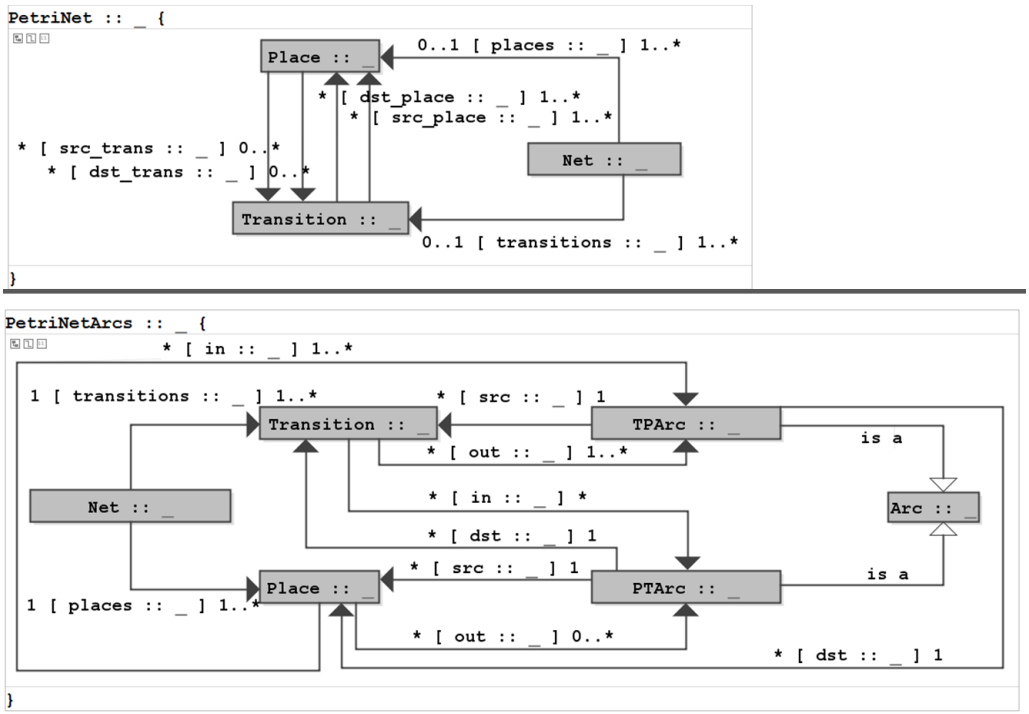


Figure 7.1: Metamodels of two Petri net variants taken from AtlanMod Zoo. The two diagrams are drawn using MetaMod.

The mechanism that we present in this chapter facilitates the creation of DSLs by leveraging previously implemented DSLs. We make it possible to reuse language aspects of a *base DSL unit* even in structurally dissimilar DSL units, called *reusing DSL units*. The key observation that lead us to create the mechanism described in this chapter is that all the constituents revolving around the metamodel (model transformations, interpreters, etc.) are implemented with some kind of *operations*, and what these operations have in common is that they query the model (take, for instance, queries in QVT [111]). This triggered the idea that the operations associated with a base DSL unit become reusable once the querying functions from the base DSL unit can be translated in terms of querying functions from the reusing DSL unit. These translations among DSLs are obtained with the mechanism of reuse mapping. Note that only the operations of the base DSL unit are reused, and not its metamodel.

We have implemented the mechanism of reuse mapping in MetaMod. Nonetheless, it is possible to implement it in other language workbenches as well. Our reuse mechanism can be applied as long as the metamodels and querying functions on value models exist in the language workbench, which is the case for most language workbenches. Metamodels are the central components in graphical language workbenches (e.g. Microsoft DSL Tools [105]) and projectional language workbenches (e.g. MPS [65]). In textual language workbenches the metamodel is encoded in the grammar of the DSLs. The metamodel can

be automatically reconstructed from the grammar (cf. Xtext [146]), or the metamodel can be the base for the grammar (cf. EMFText [57]).

Compared to other approaches, one of the characterizing features of our approach is that it does not require that the metamodels of the reusing DSLs have identical parts with the metamodel of the base DSL. Moreover, we are also supporting multiple levels of reuse mapping, giving rise to a chain of reusing DSL units. That is, imagine a Petri net DSL reusing operations from a simpler Petri net DSL that, in turn, reuses operations from a graph DSL. This is a chain of reusing DSLs, and operations in the graph DSL should be available in the last Petri net in the chain. Lastly, at the price of making a reuse mapping, one can reuse all the operations defined on the base DSL unit. To get an idea of the benefits, imagine a graph DSL and its operations being reused. There are a handful of operations that can be defined on a graph, from graph processing algorithms and statistical information on the graph to operations that transform it to all kinds of forms usable with visualization tools. The graph structure is very common in many other structures, and so having these operations already implemented can decrease the development effort significantly.

One limitation of our approach is the fact that it cannot reuse operations that update the value models, but only operations that navigate and query the value models. Why this limitation is in place will become clear in the next sections.

Moreover, the related work discussion is treated in the next chapter, because there we discuss another mechanism of reuse in the context of structurally dissimilar DSL units.

The major contribution of this chapter is the introduction of the mechanism of reuse mapping among DSL units. This leads to reusing operations defined on the base DSL unit in the reusing DSL units even when the metamodel of the base DSL unit is not identical to part of the metamodel of the reusing DSL unit. Given that these operations are used in the implementation of model transformations, interpreters, etc., such a mechanism can decrease the development time of DSLs and increase their quality (by reusing stable base DSL units). With this solution we give an answer to research question RQ_4.

> **RQ_4:** *How can we facilitate reuse of operations despite structural differences among domain-specific languages?*

7.2 Motivating Example

In this section, we are going to present the examples that triggered the work on reuse mapping. To this end, we consider graphs and another classical formalism, state machines. State machines can be viewed as graphs, and this can be exploited in the DSLs by reusing operations on the graphs in the context of state machines.

7.2.1 Graphs

Figure 7.2 depicts the metamodel of a DSL for plain directed graphs (without labels and weights). According to this metamodel, a graph consists of nodes that are linked with each other via edges.

There is a set of common operations that are used on graphs. For instance, the node and edge counts, the fan-in and fan-out of nodes, unreachable nodes from a certain node, etc., are used in various other algorithms. They could be part of a collection of common operations over graphs. Other collections could contain operations that translate the graph to other formats for visualization purposes (e.g., DOT [1] files). Furthermore,

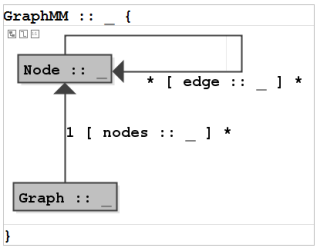


Figure 7.2: Metamodel of a plain directed graph.

operations that implement various kinds of graph processing algorithms can be considered part of yet another collection. These are only a few examples of operations that can be envisioned for graphs. Having the ability to reuse these operations in other DSLs that involve graphs can bring significant value. This is even more valuable, as graph structures can be found in many other metamodels. Figure 7.3 depicts the signatures of five common operations defined on the graph DSL.

```
PU_GraphCommons_GraphMM
for group GraphMM
aspect GraphCommons
       reuses << ... >>

operation noOfNodes(GroupType valueModel) returns int
operation noOfEdges(GroupType valueModel) returns int
operation fanIn(GroupType valueModel, ConceptType#Node# node) returns int
operation fanOut(GroupType valueModel, ConceptType#Node# node) returns int
operation unreachableNodes(GroupType valueModel, ConceptType#Node# startNode) returns
    list<ConceptType#Node#>
```

Figure 7.3: Signature of five common operations defined on the graph DSL. This is a screenshot from MetaMod. The variation in syntax is explained in Section 7.4.2.

7.2.2 State machines

Figure 7.4 depicts the metamodel of a DSL for state machines (simplified version of UML state machines [90]). We now discuss how the state machine can be viewed as a graph. This view on state machines can be obtained if one considers states as nodes and state machines as graphs. In this view, the role of the *nodes* relation from the graph metamodel can be fulfilled in the state machine metamodel with the *states* relation. What about the *edge* relation among nodes in the metamodel? Its role is fulfilled by the *Transition* concept in the state machine and its *source* and *target* relations to the *State* concept. Thus, the role of the *edge* relation from graphs can be fulfilled by a group of elements in the state machines.

We want to define an operation in the state machine DSL that deletes states unreachable from the initial state (states for which no path exists in the state machine that passes through them). The goal is to reuse the unreachable nodes operation from the graph DSL. To make that possible we have to relate state machines to graphs, which is feasible given the discussion in the previous paragraph. In that case, an operation from the state machine could reuse the unreachable nodes operation from the graph DSL, as depicted in Figure 7.5.

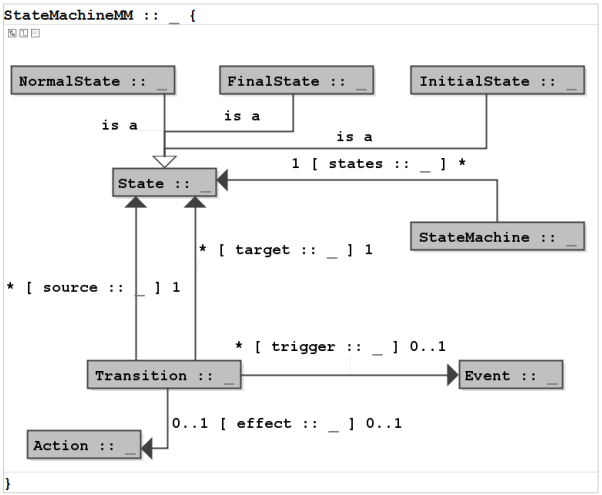


Figure 7.4: Simplified metamodel of a state machine.

```
PU_Processing_StateMachineMM
for group StateMachineMM
aspect Processing
       reuses GraphCommons

operation simplifyStateMachine(GroupType group) returns void {
  ConceptType#InitialState# initialState = initialState(group);
  list<ConceptType#State#> unreachableStates = unreachableNodes(group, initialState);

  // Delete all states that we cannot reach.
  for (ConceptType#State# unreachableState : unreachableStates) {
    group.delete(unreachableState);
  }
}
```

Figure 7.5: Definition of the *simplifyStateMachine* operation for the state machine DSL that reuses the unreachable nodes operation from the graph DSL.

This section illustrated the motivation for reusing operations defined on a base DSL in dissimilar DSLs; these reusing DSLs do not necessarily embed the structure of the base DSL. We explain how we achieve this with our reuse mechanism in Section 7.3.

7.3 Reuse mapping

In this section we describe steps required by our reuse mechanism, its visualization at the value model level, and extra validity conditions. Before that, we define two terms that we use in the remainder of the chapter: DSL units and their interfaces. Note that DSL units have an extra ingredient in this chapter, the operation signatures. Moreover, interfaces play a role only in this chapter. Interfaces and signatures were part of our exploration process and they were most useful for this phase of our work.

Definition 8 *A DSL unit consists of:*

- *a metamodel,*
- *operation signatures,*

- *and operation definitions.*

If we denote the metamodel by MM, the operation signatures by OS, and the operation definitions by OD, a DSL unit is the following tuple:

$DSL = \langle MM, OS, OD \rangle$,

where the operation signatures are defined in the operation definitions.

Given these three constituents of DSL units, we define an interface of a DSL unit as follows:

Definition 9 *The interface of a DSL unit consists of:*

- *its metamodel,*
- *and its operation signatures.*

Thus, the interface of a DSL unit is the following tuple:

$DSLI = \langle MM, OS \rangle$,

Now, consider that we have a DSL for graphs,

$DSL_b = \langle MM_b, OS_b, OD_b \rangle$,

and that we are developing a DSL for state machines, DSL_r, with the following interface:

$DSLI_r = \langle MM_r, OS_r \rangle$.

As already discussed in Section 7.2, DSL_r can intuitively be viewed as DSL_b; the question is how to enable this in tools such that we can reuse operations of DSL_b in DSL_r? For that, we need to define a reuse mapping between the two DSLs.

7.3.1 Steps for the reuse mapping

The reuse mapping between two DSLs is asymmetric. That is so, because the two DSLs involved in the reuse mapping play two different roles: one is a *base DSL unit* and one is a *reusing DSL unit*. Any DSL unit can potentially play any of the two roles. In the case presented in this section, DSL_b is the base DSL unit and DSL_r is the reusing DSL unit.

The steps required for our reuse mechanism are: (1) provide the reuse mapping, (2) translate the operation signatures (done automatically) and (3) translate the operation definitions (done automatically, as well).

What makes the explanations in the following sections hard to follow is the mix of meta levels that we need to deal with: the definitions that occur at the type model level and the execution and its effects that occur at the value model level.

7.3.1.1 Providing the reuse mapping

This step is carried out by the DSL engineer. A definition of the reuse mapping follows.

Definition 10 *A reuse mapping is a mapping from the base DSL unit to the reusing DSL unit that specifies how type concepts of the base DSL unit translate to type concepts of the reusing DSL unit and how relation queries from the base DSL unit translate to functions in the reusing DSL unit.*

The definition takes this direction of mapping because the operations from the base model will be executed on the reusing model. Thus, those operations need to be written in terms of concepts and queries of the reusing DSL unit.

A visualization of the reuse mapping defined between graphs and state machines can be seen in Figure 7.6. In this figure, one can see the mapping of the type concepts in the top-level box and the mappings of the type relations in the two bottom-level boxes. In the visualization, the type relations from graphs map to groups of type elements in state machines. These groups of type elements can be captured in a function that traverses the elements in the group, doing some processing as well, if needed.

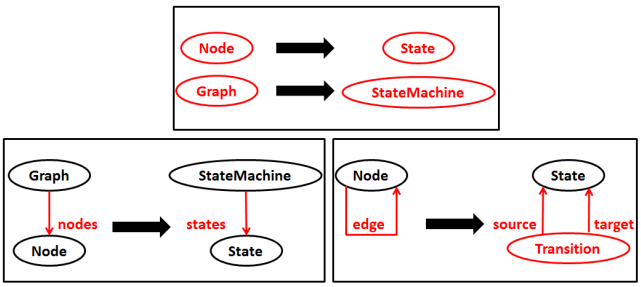


Figure 7.6: Visualization of reuse mapping between graphs and state machines.

Hence, there are two parts to the reuse mapping: the mapping of the type concepts and the mapping of the relation queries.

Reuse mapping of type concepts

The part of reuse mapping that is straightforward to define is the mapping of the type concepts because it is a one-to-one mapping.

Reuse mapping 1 *For every type concept in the base DSL unit, C_b, there is one and only one type concept in the reusing DSL unit, C_r, to which C_b is mapped.*

The reuse mapping of type concepts is not necessarily injective, that is, the same type concept in the reusing DSL unit can be mapped by different type concepts in the base DSL unit. Moreover, the reuse mapping of type concepts is not necessarily surjective, because not all of the type concepts in the reusing DSL unit need to be mapped by the base DSL unit. Looking at Figure 7.6, one can see that the type concepts from the graph DSL, *Node* and *Graph*, are each mapped to one type concept in the state machines DSL, *State* and *StateMachine*, respectively.

Reuse mapping of relation queries

Relations in MetaMod are bidirectional, so we need to handle both ends of a relation during the reuse mapping. To get an understanding of the returned elements when querying one end of a relation, consider Figure 7.7, where we show a type model and a value model that conforms to the type model. In this case, for instance, a query for target value concepts of value relation conforming to R with value concept *S1* as a source, would return a set composed of value concepts *T0*, *T1* and *T2*. That is so, because all the value relations connecting *S1* with *T0*, *T1* and *T2*, respectively, conform to type relation R.

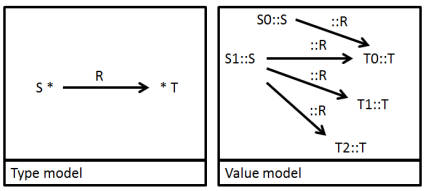


Figure 7.7: Figure depicting a type model on the left-hand side and a value model conforming to this type model on the right-hand side. The $A :: B$ notation means that value concept or value relation with name A conforms to type concept or type relation with name B.

To define the reuse mapping for relation queries, consider a type relation RT with source type concept CT_S and target type concept CT_T in the base DSL unit: $CT_S - RT - CT_T$. Then:

Reuse mapping 2 *For every type concept CT_S in relation $CT_S - RT - CT_T$ in the base DSL unit, there is a function $f(CT_S, RT)$ that returns a set of instances conforming to the mapped type concept of CT_T from the reusing DSL unit.*

The same definition holds for the other end of relation RT, where an instance of the target CT_T is given as the parameter of f. At execution time, function f returns instances conforming to type concepts from the reusing DSL unit, because the execution takes place on a value model of the reusing DSL unit.

To make this step clearer, consider the reuse mapping in Figure 7.6. When querying for the target value concepts of type relation *edge*, given a source value concept conforming to *Node*, the result is expected to be a set of value concepts that conform to *Node* in the context of the graph DSL. When translating this to the state machines DSL, the result is expected to be a set of value concepts that conform to *State*, given that *Node* is mapped to *State* in the reuse mapping for type concepts.

7.3.1.2 Translating operation signatures

Once the reuse mapping is provided, the interface of the reusing DSL changes as follows:

$DSLI_{r'} = \langle MM_r, OS_r \cup OS_{b'} \rangle$

The reuse mapping brings the operation signatures from the base DSL unit (changed to b' because of the translation of the signatures) to the reusing DSL unit (changed to r' because it contains new signatures). As a consequence, the DSL engineer can call these operations to implement operations specific to the reusing DSL unit. Making a parallel between our reuse mapping mechanism and the object-oriented paradigm, one can consider the operations from the signature as being public operations of the reused DSL unit and all the other defined operations from the reused DSL unit as being private operations.

The operation signatures from the base DSL unit change accordingly, and that is why we have a union with $OS_{b'}$ instead of OS_b in $DSL_{r'}$. The translations that occur in the signatures are straightforward: each argument type and each return type that are type concepts of the base DSL unit translate to type concepts of the reusing DSL unit (see Figure 7.3 and Figure 7.17 for such a translation). These translations are taken from the reuse mapping of the type concepts. Note that this step is automated.

7.3.1.3 Translating operation definitions

Once all the operation signatures are translated, the operation definitions need to be translated as well. The translation of an operation definition means translating the occurrences of type concepts and relation queries from the base DSL unit to the reusing DSL unit in the body of the operation. This is done with the help of the reuse mapping. All occurrences of type concepts from the base DSL unit are translated using the reuse mapping on concepts. All occurrences of the relation queries from the base DSL unit, on the other hand, are translated with the reuse mapping on relation queries. How this is done exactly in MetaMod is shown in Section 7.5. Again, note that this step is automated.

In the current implementation, we translate all the operations from the base DSL unit, because any of these can be called from the operations that appear in the signature. Recall that reuse mappings work only if the operations are not updating the value model. Thus, we now assume that no operation is updating the value model. In the future, we could check what operation updates the value model and flag this operation and all other operations that call it, such that they cannot be used in the signatures and they are not considered for translation.

7.3.2 Reuse mapping at the value model level

Reuse mappings are defined at the metamodel level, but the tool executes them at the value model level. In this subsection we describe what happens at the value model level intuitively when using our reuse mechanism.

An illustration that captures the gist of the reuse mapping at the value model level is depicted in Figure 7.8. On the left-hand side of the figure is a value model conforming to DSL_r. Here, with red circles, we have depicted value concepts conforming to type concepts from DSL_r that are mapped in the reuse mapping. These value concepts can be interpreted as value concepts conforming to type concepts from DSL_b, because of the one-to-one type concept mapping. On the right-hand side of the figure, we have extracted these value concepts and we have reconstructed value relations from the reuse mapping for relation queries (fabricated example). This gives rise to the value model representation from DSL_b. The idea is that value concepts conforming to type concepts from DSL_r embed the value concepts conforming to type concepts from DSL_b. That is, the value model conforming to DSL_r contains different relations and extra value concepts compared to its representation as DSL_b. Note that the representation as a base DSL unit is a virtual value model in the sense that the value model of the base DSL unit is not constructed anywhere in memory, but it exists implicitly with the reuse mapping.

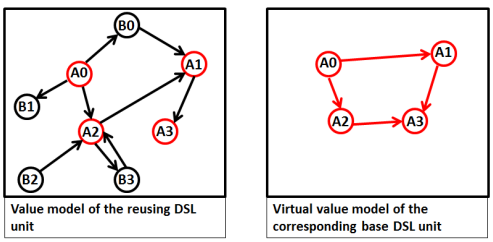


Figure 7.8: Figure depicting a model of a reusing DSL unit on the left-hand side and its representation as a base value model on the right-hand side.

7.3.3 Validity conditions for reuse mapping

There are three extra conditions that should hold in order for the reuse mapping to be valid:

1. The subtype relationship between two type concepts in the base DSL unit needs to be preserved between the mapped type concepts in the reusing DSL unit.
2. The constraints on the base DSL unit (structural and additional constraints) hold on the virtual value model of the base DSL unit.
3. The reused operations from the base DSL unit only contain querying functions and no updating functions for value models.

The first condition states that subtype relationships from the base DSL unit need to be preserved in the reusing DSL unit. That is so, because sub-concepts can be queried for relations existing in their super-concepts in operations of the base DSL unit, and these queries need to be mappable to something valid in the reusing DSL unit. Moreover, it also assures consistency between type hierarchy queries on the base DSL unit and on the reusing DSL unit.

The second condition asserts that a valid reuse mapping requires that the virtual base model respects the constraints (structural and additional constraints) defined on the base DSL unit. Note that checking the constraints can be done automatically on the virtual value model of the base DSL because operations for constraints only use queries on the value model. That means constraints defined for the base DSL can simply be run on the value model of the reusing DSL unit.

The third condition states that the operations that are reused from the base DSL unit do not contain calls to API functions that update the value models. That is so, because a deletion or an addition of a base value relation, for instance, can trigger many actions on the value model of the reusing DSL, and some of these actions can not be automated. Think, for instance, of the graph and state machine example. Adding an edge to the graph model, would require adding a new concept of type *Transition* in the reusing model, but no decision can be made on the kinds of actions and events that this transition is associated with.

7.4 Reuse mappings in MetaMod

In this section we showcase the mechanism of reuse mapping in the context of MetaMod. The reuse mapping mechanism is implemented in a meta-language called *MappingChangeableModules* in MPS. This meta-language depends on the *Models* meta-language and extends the *GenericGroupMethods* meta-language. Meta-language *MappingChangeableModules* extends some of the operations in *GenericGroupMethods*.

7.4.1 Querying functions and metamodel types in MetaMod

In this subsection, we illustrate how we handle querying functions from the base DSL unit in the reusing DSL unit in MetaMod with the help of reuse mappings. How this can be extended to other metamodeling languages is shown in Section 8.5.

In essence, a metamodel in MetaMod is made of the following elements: type concepts, type relations and a type concept hierarchy. When querying a value model, we need the

following functionality: to search for value concepts in the model, to verify hierarchical relationships among type concepts, and to find an end of a relation given the other end. These are all the functions we need to query a value model, and these also exist in various forms in other environments (see Section 8.5). This functionality is captured with the API functions presented in Section 4.2.

We now show how the API functions and types are translated from the base DSL unit to the reusing DSL unit.

The reuse mapping of the relation queries guarantees that the two querying functions, @*src* and @*tgt*, will behave correctly in the reusing DSL unit and the reuse mapping of the type concepts assures that functions *conceptsOfType*, *isTypeOf*, and *castTo*, and the *ConceptType* refer to the right concept types in the reusing DSL unit. Basically, a reuse mapping from the base DSL unit to the reusing DSL unit specifies indirectly how querying functions and type concepts from the base DSL unit are replaced in order to reflect the structure from the reusing DSL unit.

All the operations in the processing units use only these four functions to query the value models. For instance, the definition of four common operations for graphs is depicted in Figure 7.9. The pieces of text with a blue background highlight type concepts and type relations from the metamodel.

```
PU_GraphCommons_GraphMM
for group GraphMM
aspect GraphCommons

operation noOfNodes(GroupType group) returns int {
  return group.conceptsOfType(ConceptType#Node#).size;
}

operation noOfEdges(GroupType group) returns int {
  int noOfEdges = 0;

  for (ConceptType#Node# node : group.conceptsOfType(ConceptType#Node#)) {
    noOfEdges += node.@src#edge#.size;
  }

  return noOfEdges;
}

operation fanIn(GroupType group, ConceptType#Node# node) returns int {
  return node.@tgt#edge#.size;
}

operation fanOut(GroupType group, ConceptType#Node# node) returns int {
  return node.@src#edge#.size;
}
```

Figure 7.9: Definition of four common operations for the graph metamodel.

7.4.2 Examples of reuse mappings

We use the examples introduced in Section 7.2 to illustrate reuse mappings. The reuse mapping from the graph metamodel to the state machine metamodel are shown in Figure 7.10. Note that in the figures with reuse mappings, metamodel elements that correspond to the base DSL unit have a blue background, while metamodel elements that correspond to the reusing DSL unit have a yellow background. Moreover, the reuse mapping mechanism was developed in a previous version of MetaMod, that lives in branch *ReuseMethod1* in the Git repository of MetaMod. There are three differences in the querying functions of MetaMod. The first difference is that in the latest version of the querying functions the value model appears as a parameter, while in the version for reuse mappings the value model is not a parameter; (in ($\langle VM \rangle$) does not appear anymore in $\langle CV \rangle$.@src#$\langle RT \rangle$# in ($\langle VM \rangle$)). The value model is considered fixed and it is considered

to be the input value model. Partly for the same reason, the second difference is that *GroupType* is not parameterized with the name of the group (the group type is always considered the group type specified in the header of the processing unit). Moreover, the multi-operation feature is not present in this version of MetaMod.

We now explain the reuse mapping labeled 1 in Figure 7.10. Given the *nodes* relations and a value concept of type *Graph* as source of these relations, the result should be a set of value concepts of type *Node* (see the metamodel of the graph DSL in Figure 7.2 and the *nodes* relation with *Graph* as a source and *Node* as a target). That is, given a fixed value concept of type *Graph*, and all the instances of the *nodes* relation that have this instance as a source, the function returns instances of type *Node*. How does this translate to the context of the state machine DSL? The mapped concept for *Graph* is *StateMachine* and the *states* relation fulfills the same goal as the *nodes* relation. Thus, we use the *StateMachine* concept as a source for the *states* relation to obtain concepts of type *State* (see the metamodel of the state machine DSL in Figure 7.4 and the *states* relation with *StateMachine* as a source and *State* as a target). *Node* concepts map to *State* concepts, thus obtaining valid return types. In the reuse mapping labeled 2 in Figure 7.10, one can see the reverse query function on the *nodes* relation.

Each type relation is handled by two mappings, one where the source concept is known and one where the target concept is known.

```
View StateMachinesAsGraphs
  view StateMachineMM as GraphMM

  StateMachine as Graph
  State as Node                    ⟶ Reuse mapping for type concepts

(Graph).@src#nodes# {                                          1
  return (StateMachine).@src#states#;
}

(Node).@tgt#nodes# {
  return (State).@tgt#states#;                                 2
}

(Node).@src#edge# {
  list<ConceptType#State#> states = new linkedlist<ConceptType#State#>;

  for (ConceptType#Transition# transition : (State).@tgt#source#) {
    states.addAll(transition.@src#target#);
  }                                                            3

  return states;
}

(Node).@tgt#edge# {
  list<ConceptType#State#> states = new linkedlist<ConceptType#State#>;

  for (ConceptType#Transition# transition : (State).@tgt#target#) {
    states.addAll(transition.@src#source#);
  }

  return states;
}
```

Figure 7.10: Reuse mappings from graphs to state machines. The notation (*ConceptType*) means an instance of type *ConceptType*. In the body of the mapping, the instance is always of type the type concept resulting from the mapping of the concept in the prologue.

Now we are going to discuss how did we map the *edge* relation. For that, one should

look at the *edge* relations with an instance of *Node* as a source in the reuse mapping labeled 3 in Figure 7.10. In the context of state machines, this is equivalent to looking for the target states given a source state in a transition. To obtain the target states starting from a source state, we iterate over the transitions that are attached to the states via the *source* relations, and then we return the states that are attached to the transitions via the *target* relations. This reuse mapping gives a hint of the power of the mechanism. The metamodel of the base DSL unit does not need to be embedded in the metamodel of the reusing DSL unit for the reuse mapping to be applicable.

7.4.3 Chaining of reuse mappings

One interesting feature of our approach is that reuse mappings can also be chained. To this end, we first present two flavors of the Petri net metamodels.

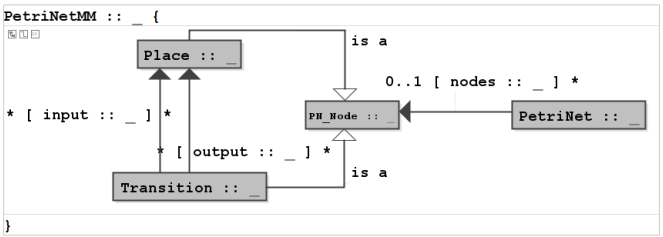


Figure 7.11: Simple Petri net metamodel.

Figure 7.11 shows the first flavor of Petri net metamodel which we call simple Petri nets. Simple Petri nets can be viewed as graphs. To this end, the role of concept *Node* in the graph metamodel is played by concept *PN_Node* in the simple Petri net metamodel. Thus, both places and transitions play the role of nodes in Petri nets. Moreover, the role of relation *edge* from the graph metamodel is played by both the *input* and *output* relations in the simple Petri net metamodel. These translations are captured in the reuse mapping from graphs to Petri nets in Figure 7.12.

Now imagine that we implement the Petri net DSL in Jetbrains MPS and that we want to render the Petri net value models graphically. If one wants to accomplish this in Jetbrains MPS, a plugin for graphical visualizations can be used. However, this plugin requires that the metamodel has a certain structure. In particular, each relation to be visualized must be 'classified', i.e., have a corresponding concept in the metamodel. In the Petri net example, if we want to represent both places and transitions with geometrical figures in the visualization, and the relations between them with edges, then we need to turn these relations into concepts. This is just an example of why one would want to create this flavor of Petri nets. Figure 7.13 shows the modified Petri net metamodel. We are going to refer to this flavor as the visualization Petri net.

This second flavor of Petri nets has a different structure for visualization purposes, but the operations available on the simple Petri nets should be available on the visualization Petri net as well. That is so, because the visualization Petri nets have been introduced to cater for the structural needs of the plugin. The visualization Petri nets need the same operations as those from simple Petri nets. An excerpt of the mapping between simple Petri nets and visualization Petri nets is shown in Figure 7.14.

After these two levels of reuse mapping, one is still able to use the operations from the graphs in the visualization Petri nets. This is shown in Figure 7.15. In the operation

```
View PetriNetsAsGraphs
  view PetriNetMM as GraphMM

  PetriNet as Graph
  PN_Node as Node

(Graph).@src#nodes# {
  return (PetriNet).@src#nodes#;
}

(Node).@tgt#nodes# {
  return (PN_Node).@tgt#nodes#;
}

(Node).@src#edge# {
  if ((PN_Node).isTypeOf(ConceptType#Place#)) {
    return (PN_Node).castTo(ConceptType#Place#).@tgt#input#;
  } else {
    return (PN_Node).castTo(ConceptType#Transition#).@src#output#;
  }
}

(Node).@tgt#edge# {                                          4
  if ((PN_Node).isTypeOf(ConceptType#Place#)) {
    return (PN_Node).castTo(ConceptType#Place#).@tgt#output#;
  } else {
    return (PN_Node).castTo(ConceptType#Transition#).@src#input#;
  }
}
```

Figure 7.12: Reuse mappings from graphs to simple Petri nets.

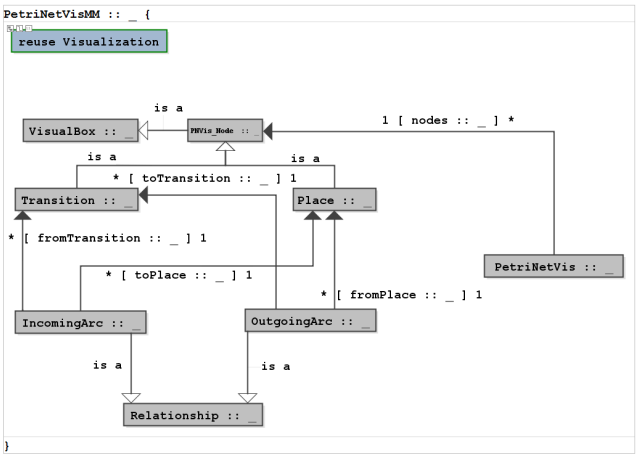


Figure 7.13: Visualization Petri net metamodel.

statisticalInformation of the visualization Petri net we are printing some information on the number of nodes and arcs in the model.

```
View PetriNetVisAsPetriNet
  view PetriNetVisMM as PetriNetMM

  PetriNetVis as PetriNet
  Place as Place
  Transition as Transition
  PNVis_Node as PN_Node

(Transition).@src#input# {
  list<ConceptType#Place#> states = new linkedlist<ConceptType#Place#>;

  for (ConceptType#OutgoingArc# outArc : (Transition).@tgt#toTransition#) {
    states.addAll(outArc.@src#fromPlace#);
  }

  return states;
}
```

5

Figure 7.14: An excerpt of the mapping from simple Petri net to visualization Petri net.

```
PU_Statistics_PetriNetVisMM
for group PetriNetVisMM
aspect Statistics
       reuses GraphCommons

operation statisticalInformation(GroupType group) returns void {
  info "Number of places and transitions: " + noOfNodes(group);
  info "Number of arcs: " + noOfEdges(group);
}
```

Figure 7.15: Operation definition for the visualization Petri net metamodel that reuses the number of nodes and number of edges operations from the graph metamodel.

7.5 Execution of a reused operation in MetaMod

In this section, we give a few insights into how the mechanism of reuse mapping is implemented in MetaMod. To this end, we take Petri nets as an example again.

The most interesting part of the execution is the one related to how MetaMod handles the call of an operation from the base DSL unit in the context of the reusing DSL unit. To understand this, consider the visualization Petri net, the simple Petri net and the graph DSLs. Let us trace an invocation of the *fanIn* operation when it is called on a visualization Petri net value model. Figure 7.16 shows the definition of the method *fanIn* in the graph DSL. The body of the method is expressed in terms of a querying function on the metamodel of the graph DSL. That is, the function queries for the source nodes of the *edge* type relation with value concept *node* (parameter of the operation) as a target. When called in the context of the visualization Petri net DSL, this querying function needs to be re-expressed in terms of querying functions on the visualization Petri net metamodel. The querying function for type concept *Node* as target for type relation *edge* is handled by the reuse mapping from graphs to simple Petri nets (see Figure 7.12). In Figure 7.12, in the reuse mapping that we labeled 4, one can see the translation in terms of simple Petri nets of an instance of type concept *Node* as target for type relation *edge*.

In Figure 7.16, the body of the *fanIn* operation for visualization Petri net contains a call to function *Node_asTarget_edge* instead of the querying function on metamodel elements of the graph. The body of this function is actually the reuse mapping labeled 4 from Figure 7.12. Again, because this reuse mapping is using querying functions on the

simple Petri net metamodel, these need to be transformed into querying functions on the visualization Petri net metamodel. This is done in the reuse mapping from simple Petri nets to visualization Petri nets, part of which can be see in Figure 7.14. In this figure, the reuse mapping that we labeled 5 is expressing how instances of type concepts *Transition* as source for type relation *input* are translated. Again, in Figure 7.16 one can see how in the body of the *Node_asTarget_edge* function, the querying function on *Transition* as a source for type relation *input* is transformed into a function call, *Transition_asSource_input*. The body of this function contains the reuse mapping labeled 5, which is expressed in terms of elements of the visualization Petri nets. Thus, the *fanIn* operation for the visualization Petri nets only needs to handle querying functions expressed on the metamodel of the visualization Petri net DSL, which it knows how to handle.

Figure 7.16: Visualization of how the *fanIn* operation from the graph DSL is executed on the visualization Petri net DSL.

One more important aspect to illustrate in the execution is how the operation signatures of the base DSL unit are translated into operation signatures of the reusing DSL unit by means of the reuse mapping of the concepts. For instance, the operations defined on the graph DSL and depicted in Figure 7.3 are translated into the operations in Figure 7.17 such that they are expressible in the visualization Petri net DSLs. The correspondences between the concepts of the two DSLs was established with the help of the reuse mapping in Figure 7.12 and Figure 7.14.

7.6 Discussion

In this section we are going to draw a parallel between our mechanism of reuse mapping and object-oriented programming (OOP) in an effort to get a better understanding of its

```
Interface_GraphCommons_PetriNetMM
for group PetriNetVisMM
aspect GraphCommons

operation noOfNodes(GroupType valueModel) returns int
operation noOfEdges(GroupType valueModel) returns int
operation fanIn(GroupType valueModel, ConceptType#PNVis_Node# node) returns int
operation fanOut(GroupType valueModel, ConceptType#PNVis_Node# node) returns int
operation unreachableNodes(GroupType valueModel, ConceptType#PNVis_Node# startNode) returns
    list<ConceptType#PNVis_Node#>
```

Figure 7.17: Signatures of operations for the graph DSL transformed into signatures of operations for the visualization Petri net DSL.

basics. Moreover, we are going to discuss benefits, limitations, guidelines and ways to implement the reuse mapping in other tools.

To some extent, there is a parallel among our mechanism and specialization in object-oriented programming. One can regard the base DSL unit as a base class and the reusing DSL unit as a specialization of the base class. Similar to OOP, the reusing DSL is required to have the structural and behavioral characteristics of the base DSL, but it can also have additional structural and behavioral characteristics. The parallel is not complete because structural-wise, only the concept inheritance hierarchy is required, and not the relations among concepts.

7.6.1 Benefits and limitations of the reuse mapping

The biggest contribution of the reuse mapping mechanism is leveraging previously developed DSL units in the context of structural differences among the corresponding parts of the DSL units. Once the mapping between the metamodel of the base DSL unit and the metamodel of the reusing DSL unit is made, one can reuse operations defined on the base DSL unit in the reusing DSL unit (see Figure 7.15 and the reuse of *noOfNodes* and *noOfEdges* operations).

The reuse mapping pays off especially for base DSL units with a large number of operations, having many details to them (think only of the complexity of the graph processing algorithms).

Given the reuse mapping mechanism, one possible scenario is to create repositories of stable base DSL units that occur in multiple projects. These can capture reoccurring patterns in DSL design. Using the base DSL units and our reuse mechanism, DSL engineers can benefit from the operations defined on them in their own DSLs. One can also imagine reusing legacy DSL units as base DSL units, when those legacy DSL units are stable enough.

One limitation of the reuse mapping mechanism is the creation of extra function calls from the translation of the relation queries. This could result in performance penalties. Another limitation is the fact that a change in the metamodel of the base DSL unit, will most likely require a change in the reuse mapping as well. That is why, the base DSL unit should be a stable DSL unit.

7.6.2 Guidelines for the mechanism

If one creates a DSL unit with the sole purpose to be a base DSL unit, we have a few guidelines to make it as reusable as possible without compromising on functionality:

- Prefer relations over concepts. The more concepts one can transform into relations, the more reusable the base DSL unit. For instance, if we chose edge to be a concept in the graph metamodel in Figure 7.2 instead of a relation, then we could not map it to the simple Petri net (see Figure 7.11). That is because we do not have a corresponding concept for edge in the simple Petri net metamodel.

- Make multiplicities of relation ends as encompassing as possible. For instance, consider creating a state machine with multiple start states instead of a state machine with a single start state, because then, the one with a single start state is a particular case of the one with multiple start states. Multiplicity violations on the virtual base model are caught with checking the constraints defined for the base DSL unit.

- Avoid subtype relationships. The fewer the subtype relationships, the fewer the restrictions that need to hold for the reusing DSLs.

7.6.3 Interaction of reuse mapping with other features of MetaMod

In this section, we look at how other features of MetaMod, and especially features missing from branch *ReuseMethod1*, interact with the reuse mapping mechanism. For this section, assume we are mapping the reusing DSL with top-level group, *Top*.

The modular structure of the groups created with group reuse does not affect reuse mappings, because operations defined for reused groups operate only with metamodel elements that are known in the top-level group, *Top*, that is being mapped. Thus, those metamodel elements are mapped as well.

The multi-operation feature of MetaMod is not present in branch *ReuseMethod1*. Nonetheless, the addition of multi-operations should not add any complications, because it can be treated as a normal operation from the perspective of the reuse mapping.

The parameterized $GroupType\#\langle GT\rangle\#$ is also not present in branch *ReuseMethod1*. Nonetheless, a simple solution to this addition would be to translate all group types to the top-level group type, $GroupType\#Top\#$ (thus, all the generated queries are allowed).

7.6.4 Possible implementations for existing tools

One way to support the reuse mapping mechanism in other tools is to avoid functions that update the models in reused operations and to map the querying functions from the base model such that they are valid in the reusing DSL model.

Consider the following scenario: one creates DSLs by creating metamodels in the Eclipse Modeling Framework EMF [135] and by creating grammars for them using Xtext so that models can be represented textually. Moreover, consider the model management language Epsilon Object Language - EOL [79]. This language can be used to navigate, create and update models. Using EOL as a language for specifying operations for the DSLs, our mechanism should be applied in the context of EOL. To make that possible, one needs to consider the following steps:

- Reusable operations are not allowed to use creating and updating model functions. In the specific case of EOL, functions like *newInstance()*, *delete()*, and *add(Any)* and *remove(Any)* on the collections belonging to the model are not allowed. Moreover, feature accesses on classes cannot appear on the left-hand side of an assignment

(an attribute or a reference of a class are accessed via the class with the '.' or '→' operators).

- We need to provide a reuse mapping for classes in the base model and for feature accesses in the model (this is more straightforward than in MetaMod because MetaMod has double navigation on a relation, while in EOL we can only access elements in the direction of the reference).

The operation signatures in this case could be considered the operation signatures of the classes defined in the metamodel itself, as there are no other modules where the signatures are specified.

7.7 Conclusions

In this chapter we showed how to leverage previously created DSL units in the implementation of new DSLs when the base DSL unit and the reusing DSL units do no have structurally identical parts. This was accomplished with the reuse mapping mechanism, that translates querying functions on the base DSL unit in terms of querying functions on the reusing DSL units. The benefits of the reuse mapping mechanism were showcased on a graph DSL as a base and state machines and Petri nets DSLs as reusing entities. As a result of applying reuse mappings on these examples, one can reuse all the operations defined on the graph DSL in the state machines and Petri nets DSLs. This could lead to an increase in the development productivity and also an increase in the quality of the DSLs. Although the mechanism is currently implemented in MetaMod, reuse mapping could be transferred to other technologies as well as noted in Section 7.6.4.

Chapter 8

Delegated operations

Most language workbenches provide support for reuse of DSL aspects when one DSL is structurally similar to other DSL. The problem is that, although logically similar, the corresponding parts in these DSLs can differ considerably in structure because of the specific usage scenarios that they are developed for. In the last chapter, we have presented a possible solution to this in the form of the reuse mapping mechanism. The problem was that reuse mappings enforced the subtype hierarchy to be preserved, which we found limiting in many cases.

In this chapter, we propose another approach for reusing language aspects from an existing DSL in the definition of another DSL despite the structural differences among the two. These DSL aspects are ultimately implemented in the language workbenches using operations. Our approach enables reusing operations defined on a base DSL (the existing DSL) in the context of a DSL under development, and it has two levels: the definition and the execution. On the definition level, the DSL engineer writes model transformations from the DSL under development to the base DSL, and she also writes the reused operations' signatures in the DSL under development. On the execution level, the reused operations are called on the model of the DSL under development and they are executed on its base model representation. Our approach is called delegated operations *because reused operations from the base DSL in the DSL under development are delegated to the base model representation. Our approach of delegated operations is able to cope with bigger structural differences between the base DSL and the corresponding part in the DSL under development than reuse mappings.*

To test the feasibility of our approach, we have implemented it in MetaMod. Moreover, we showcase the approach on a case study.

8.1 Introduction

The tools used to implement DSLs, language workbenches, offer little support for reuse of DSLs that are structurally different. This is important to achieve, given that DSLs targeting the same domain often involve different metamodels. This happens because

DSLs are built for specific purposes, and thus, the metamodel reflects the specific usage scenarios (see explanation in Section 7.1).

In this chapter, we present our approach of delegated operations, which consists of reusing operations defined on a *base DSL* in a *DSL under development*, even when the metamodel of the base DSL is structurally different from the corresponding part in the metamodel of the DSL under development. Our approach is explained at two levels: the definition and the execution level. On the definition level, model transformations are used to transform the value models of the DSL under development to value models of the base DSL. Furthermore, instead of reimplementing the operations from the base DSL for the DSL under development, only a redefinition of their signatures in the DSL under development is needed. This can be partially automated in many cases. On the execution level, whenever these operations are called on a value model expressed in the DSL under development, their execution is delegated to the base DSL representation and executed on that representation. With this solution, we contribute to research question RQ_4.

> **RQ_4:** *How can we facilitate reuse of operations despite structural differences among domain-specific languages?*

Our approach thus focuses on the reuse of auxiliary DSL aspects[1] and not the reuse of metamodels. The metamodels form the connection between the base DSL and the DSL under development by defining a model transformation among them.

Delegating operations to the base model representation can lead to more efficient computations because computations are done at the right level of abstraction. Take as an example a compositional state machine DSL (contains composite states) and a simple state machine DSL. Instead of writing operations that model check the compositional state machine, it is more convenient to have a representation of the compositional state machine value model as a simple state machine value model and perform the model checking operations on the simple state machine value model. If one implements the model checking operations directly on the compositional state machines, the operations need to keep track of the composed states at all steps of the computations, leading to a significant overhead. Computations at the right level of abstraction could outweigh the performance penalty caused by the introduction of a new level of indirection with our approach.

Another advantage of the approach is the fact that there can be significant structural differences between the metamodel of the base DSL and the logically corresponding part in the metamodel of the DSL under development. Many of the existing approaches of reuse require that you have the same metamodel or a small variation of it (see Section 8.6). This is restrictive, especially for domain-specific languages, which are created for specific purposes.

Moreover, the approach leverages existing DSL implementations. There are DSLs and variations of them that occur frequently in other DSLs. As an example, take a graph description DSL. The graph structure can be found in many other DSLs and given the amount of graph processing operations that can be defined on a graph DSL, the advantage of reusing them in the other DSLs is significant.

One limitation of our approach of delegated operations is the fact that reused operations that modify the base model representation do not propagate the modifications back to the model of the DSL under development. That is not a problem, however, for DSL aspects

[1] An auxiliary DSL aspect (or language aspect) is any DSL aspect that is not the metamodel, which is considered the central DSL aspect. Thus, interpretation, code generation, etc., are auxiliary DSL aspects.

such as code generators, interpreters or model transformations that do not modify the source model. They only need to query the source value models.

We have implemented our approach of delegated operations in MetaMod, and we have tested it on a number of case studies; some of these we give as examples in this chapter. With our approach we implemented applications of reuse that cannot be achieved with other approaches for reuse; for instance, a composite state machine DSL reusing the operations of a simple state machine DSL.

8.2 Motivating Examples

In this section, we discuss three example DSLs: a graph DSL, a simple state machine DSL and a composite state machine DSL. The main idea of these examples is that a simple state machine can be regarded as an extension of a graph, and a composite state machine can be regarded as an extension of a simple state machine. So why not reuse operations defined on the graph DSL and the simple state machine DSL in the simple state machine DSL and the composite state machine DSL, respectively?

You will notice that the structure of the graphs and simple state machines is slightly different from the structure of the corresponding DSLs in the previous chapter. This is reinforcing the idea that metamodels for the same DSL can take many different forms, depending on the specific usage scenario. Moreover, we introduce the composite state machines in this chapter. Note that the reuse of simple state machine operations in composite state machines was not possible with reuse mappings.

8.2.1 Graph DSL

The first DSL is a graph DSL. The metamodel of the graph DSL is depicted in Figure 8.1. The concepts that play a role in the graph DSL are the graph concept, the node concept and the edge concept. The graph is directed, so the edge is associated to a source and a target node.

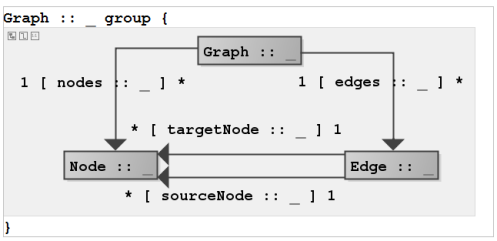


Figure 8.1: Metamodel of a plain directed graph. This is a snapshot from our tool, MetaMod. An explanation of the elements in this diagram is given in Section 4.1.5.2.

There are numerous operations that can be defined on a graph model (for code generation purposes, analysis, etc.), from graph processing algorithms to transformations and visualization formats for different tools. Many of these operations are not trivial to implement. That is why it would benefit DSL engineers to have a reference graph DSL with all kinds of operations defined on it for reuse of DSLs that have a graph flavor to them. The signature of some operations defined on graphs can be seen in Figure 8.2.

```
operation noOfNodes(GroupType#Graph# inputGroup) returns int
operation noOfEdges(GroupType#Graph# inputGroup) returns int
operation fanIn(GroupType#Graph# inputGroup, ConceptType#Node# node) returns int
operation fanOut(GroupType#Graph# inputGroup, ConceptType#Node# node) returns int
operation unreachableNodes(GroupType#Graph# inputGroup, ConceptType#Node# startNode) returns list<ConceptType#Node#>
```

Figure 8.2: Operation signatures for the graph metamodel.

8.2.2 Simple state machine DSL

The next DSL we explore is a simple state machine DSL. The metamodel of the state machine DSL is represented in Figure 8.3. There are states and transitions with events and actions. An important observation here is that the state machine can be seen as an extension of the graph. States can be seen as nodes of the graph and transitions can be seen as edges of the graph. Besides these, there are more details added to the simple state machine DSL. Thus, in theory, the operations defined for the graph DSL could be reused for the simple state machine DSL, e.g., the operation for finding dead states in the state machine DSL (see Figure 8.4) could be the same as the operation for unreachable nodes from the graph DSL (see Figure 8.2). Note that this depends on the semantics of the simple state machine.

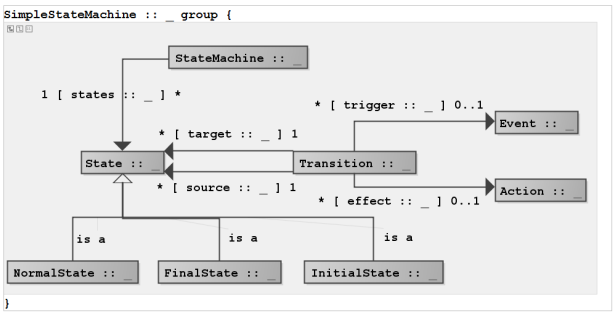


Figure 8.3: Metamodel of a simple state machine.

There are numerous operations that can be implemented for the state machine as well. Consider, for instance, the operations for model checking a state machine. These operations check that certain properties hold on a state machine, e.g., a state is always eventually reachable. Reusing these operations in other DSLs that can have a simple state machine representation would greatly benefit the DSL engineers. The signature of some operations defined on simple state machines can be seen in Figure 8.4.

```
operation deadStates(GroupType#SimpleStateMachine# inputGroup, ConceptType#State# startState) returns
    list<ConceptType#State#>
operation checkDeterminism(GroupType#SimpleStateMachine# inputGroup) returns boolean
operation generateNuSMV(GroupType#SimpleStateMachine# inputGroup) returns NuSMV
operation sIsAlwaysEventuallyReachable(GroupType#SimpleStateMachine# inputGroup, ConceptType#State# s, NuSMV nuSMV)
    returns boolean
operation sIsGloballyTrue(GroupType#SimpleStateMachine# inputGroup, ConceptType#State# s) returns boolean
operation sRespondsToPGlobally(GroupType#SimpleStateMachine# inputGroup, ConceptType#State# s, ConceptType#State# p)
    returns boolean
```

Figure 8.4: Operation signatures for the simple state machine metamodel related to model checking. In these operations, NuSMV represents the symbolic model checker [24].

8.2.3 Composite state machine DSL

Lastly, there is the DSL for composite state machines. This DSL differs from the simple state machine DSL because it has an additional state, the composite state that contains other states, both simple and composite states. This gives rise to a hierarchical organization of the states. The composite state machine can be regarded as a simple state machine (through a transformation) and model checking operations defined for the simple state machine could be executed on the composite state machine.

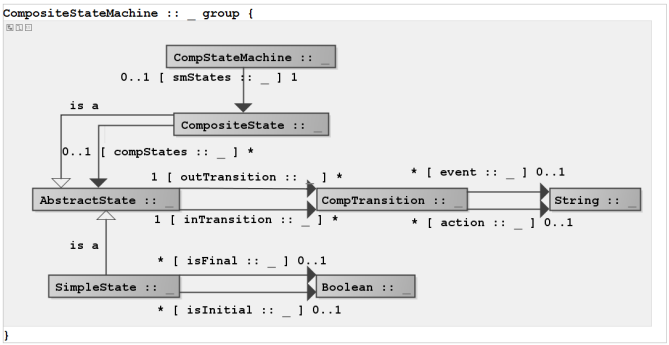


Figure 8.5: Metamodel of a composite state machine.

8.2.4 Reuse terminology and questions

Our approach to reuse operations defined on a DSL in another DSL is usable in cases when one of the metamodels logically extends the other, but the structure in the extended metamodel is not necessarily similar to the equivalent part of the structure in the extending metamodel. We consider two different DSLs:

- A base DSL, BL, with a metamodel, and operations defined on it, Op_{BL}.
- A DSL under development, LUD, with a metamodel, and for which operations are being defined, Op_{LUD}.

Our intention is to reuse Op_{BL} in LUD. Thus, there are two asymmetric roles in the relationship between two DSLs, one is the BL and one is the LUD. Going back to the examples in Section 8.2, one can notice at least two applications for reuse: first, the graph DSL is the BL and the simple state machine DSL is the LUD, and, second, the simple state machine DSL is the BL and the composite state machine DSL is the LUD.

The questions that arise from the intention of reusing Op_{BL} in LUD are as follows:

- How does the signature of the reused operations from Op_{BL} transform when these operations are reused in Op_{LUD} (we denote them by Op_{LUD_BL})?
- What is the semantics of Op_{LUD_BL} in terms of Op_{BL}?

The approach we describe in the next section answers these two questions.

8.3 The approach of delegated operations

In this section we cover our approach for reuse of auxiliary DSL aspects. The description that we provide in the next subsections is independent of MetaMod. For the delegated operations, we use only ingredients that can be found in other language workbenches. We start with two main insights behind our approach, and we then continue to explain the two levels of the approach, the definition and the execution.

There are two main insights behind our approach of delegated operations. Keep in mind that these are not necessarily new insights. The first insight is that the metamodel is the central aspect of a DSL, and all the other DSL aspects, auxiliary DSL aspects, depend on it. This led to the decision to make the metamodel of a DSL the connection to other DSLs. The connection between two DSLs is established through defining a model transformation between the two metamodels of the DSLs. The second idea is that we do not necessarily need to execute operations on the *LUD* value model, but we can delegate them to a different representation of it, the *BL* value model.

For the purpose of the following explanations, we consider a generic operation that has two types of parameters: *in* and *out* parameters.

A visual description of the definition and execution of the approach of delegated operations can be seen in Figure 8.6 and Figure 8.7. Figure 8.6 depicts the steps that define the approach and their succession. First, a model transformation needs to be created between the *LUD* and the *BL*. Afterwards, the operations from *LUD* that are executed on the *BL* need to specify the delegated operation from *BL* and the conversions of the *in* and *out* parameters. Then, Figure 8.7 depicts the execution of the approach. First, a model transformation is applied on the *LUD* value model (step 0.0), resulting in a *BL* representation of the *LUD* value model that is validated (step 0.1). The *BL* operations that are called on elements of the *LUD* value model are delegated to the *BL* value model after the conversion of the *in* parameters (step 1). After the operation is executed, the converted *out* parameters are returned to the *LUD* value model (step 2).

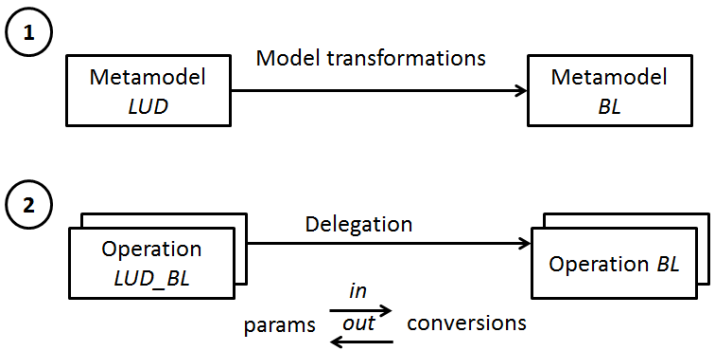


Figure 8.6: High level description of the definition process of the delegated operations.

The DSL engineer defines the approach of delegated operations at the metamodel level. To define it, she needs to take the following two steps:

- Define (write) a model transformation from the *LUD* metamodel to the *BL* metamodel;
- Define (write) the signature of the *LUD* operations that reuse *BL* operations;

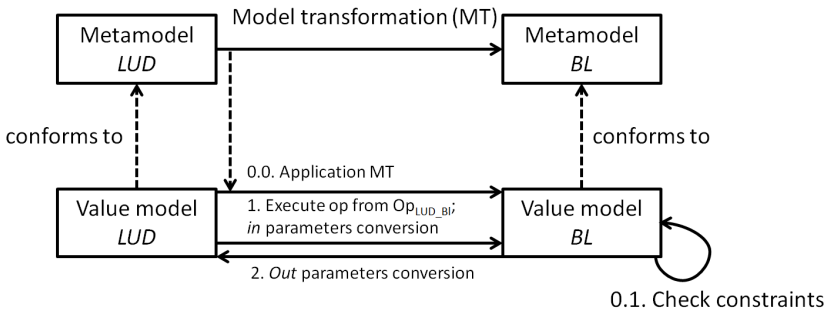


Figure 8.7: High level description of the execution process of the delegated operations.

– Indicate delegated operation from *BL*;
– Indicate conversion operations for *in* parameters;
– Indicate conversion operations for *out* parameters.

To understand the approach of delegated operations, it is also important to look at its execution steps. The execution of our approach is done at the value model level. The execution steps are as follows (note that all these steps are executed automatically):

- The model transformation is applied on the value model of the LUD;
- The resulting BL value model representation is checked for validity;
- The execution of an operation $op \in Op_{LUD\_BL}$ is delegated to the BL value model representation;
- The *in* parameters of operation *op* are converted;
- The operation *op* is executed on the BL value model representation;
- The *out* parameters that resulted from the execution of operation *op* are converted and returned to the LUD value model.

Note that the model transformation is executed once for the LUD value model, and all the operations use the resulting base value model representation. That is why *the BL value model representation is read-only for the LUD context.* That means, modifications made by operations in Op_{LUD_BL} on the base value model are dropped in the calling context, that of LUD.

Moreover, there is a validity check executed on the BL value model representation immediately after the transformation. That happens because the operations associated to a metamodel are implemented with the assumption that the value models they are executed on are structurally valid (they conform to the metamodel) and they fulfill the user-defined constraints (extra constraints that are not captured in the metamodel). That is why, the resulting value model of the transformation must be valid with respect to two types of constraints: structural and user-defined.

In the following subsections, we focus on the two steps that the DSL engineer needs to undertake to define the approach.

8.3.1 Model transformations

The first step in our approach definition is to define a correspondence between the two DSLs. This is achieved by defining a model transformation from the *LUD* metamodel to the *BL* metamodel.

We give a classification of the model transformation required by our approach, based on the categories specified by Mens et al. [99]. The model transformation that one needs to define for the approach:

- is *exogenous* because the two DSLs are different (e.g. from the simple state machine DSL to the graph DSL);
- is *vertical* because usually the *LUD* has more details than the *BL* (e.g. the simple state machine DSL has more details than the graph DSL);
- is performed in the *same technological space* (e.g. at the moment, in MetaMod);
- has *small complexity*, in general, because the *BL* metamodel should be logically similar to part of the *LUD* metamodel, although not necessarily structurally (e.g. the simple state machine metamodel logically contains the graph metamodel);

Usually, model transformation languages have different types of transformations [99]. There is a distinction between one-to-one transformations and other types of transformations. This distinction is important for the next step of the approach definition, and it will be illustrated in Section 8.4.

8.3.2 Operation signatures for OP_{LUD_BL}

The DSL engineers can create *LUD* operations that they delegate to *BL* operations. This is done by creating an operation signature for the *LUD* operation. In the signature, they also need to specify the *BL* operation to which the *LUD* operation delegates and the conversion functions to and from *BL* concepts for the parameters of the *LUD* operation. For *in* parameters, one needs to specify conversion functions that take a *LUD* value concept and convert it to a *BL* value concept. For *out* parameters, one needs to specify conversion functions that take a *BL* value concept and convert it to a *LUD* value concept.

For instance, take the operation *fanIn* (see Figure 8.2) from the graph DSL. The *node* parameter has type *Node* from *BL*. Assume that the DSL engineer wishes to reuse this operation in the simple state machine. That means she needs to write the operation signature using *LUD* type concepts in the signature instead of the *BL* type concepts. For that, she needs to specify a counterpart in the simple state machine for type concept *Node*. In this case, the counterpart is *State*. The DSL engineer also needs to specify how a value concept of type *State* converts into a value concept of type *Node* because *node* is an *in* parameter (see Figure 8.8). This conversion function is used by our approach when delegating an operation call with a specific parameter of type *State* to the *BL* value model, to find out the corresponding value concept of type *Node* in the *BL* value model representation.

Not all operations from the *BL* need to be reused in the *LUD*. For instance, in the case of the simple state machine DSL, the DSL engineer could be interested in the reuse of only a high-level model checking operation. In turn, this operation could contain calls to many other model checking operations for the simple state machine. Because not all operations from the *BL* are reused in the *LUD*, we say that we reuse fragments (parts) from the auxiliary DSL aspects of *BL*.

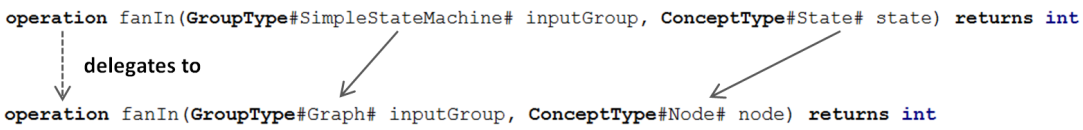


Figure 8.8: Visual description of operation delegation.

To answer the last two questions in Section 8.2.4, in our approach, the signature of the operations is handled by the DSL engineers (although at least part of the signature can be derived automatically; see Section 8.4). On the other hand, the semantics of an $op \in Op_{LUD\_BL}$ (operation in the list of LUD operations delegating to BL) is given by the transformations of the parameters of op and the semantics of the delegated operation from BL.

8.4 The approach of delegated operations in MetaMod

In this section we demonstrate the approach of delegated operations in MetaMod using the examples of the graph DSL, the simple state machine DSL and the composite state machine DSL.

Before we dive into any further details, remember that all the operations associated to a metamodel in MetaMod are a variation of Java methods. Thus, *in* parameters are the parameters of an operation and *out* parameters are the returned elements and all the elements wrapped in an object given as a parameter.

An overview of the steps for defining our approach and that are completed by either the DSL engineer or MetaMod, follows:

- Define model transformations (DSL engineer);
- Define signatures of delegating operations (DSL engineer and MetaMod, partially):

8.4.1 Model transformations

The model transformation language that we have designed for MetaMod is an imperative transformation language. A model transformation is created per source and target metamodel, and it consists of a set of operations. The model transformations are applied on value models that conform to the source metamodel of the model transformation. Moreover, there is a main transformation operation from which the entire transformation process starts.

There are three types of operations in the model transformation language of MetaMod:

- The transformation operation (see Figure 8.9); this is a one-to-one transformation. That means that it can have one and only one LUD type concept as a parameter and it can return one and only one BL type concept. This operation can be used for converting the *in* parameters of the delegating LUD operations as well.
- The helper operation (see Figure 8.10); this is a many-to-many transformation.
- The conversion operation (see Figure 8.11); this is an operation that is not used as a transformation. It is used for the conversion of *in* and *out* parameters of the delegating LUD operations.

All the operations in the model transformation language have at least two fixed parameter types. These are the first two parameters and represent the input type group (metamodel) and the output type group (metamodel). Thus, if transforming from a simple state machine DSL to a graph DSL, the first parameter type of an operation would be a *SimpleStateMachine* type group and the second parameter type would be a *Graph* type group.

The body of the operations of the model transformation can contain any computations, but an important component is a simple API to create model elements (besides the one for querying model elements that was described in Section 4.2). A few elements of the API can be seen in Figure 8.9. The figure depicts the transformation operation for transforming a state from the simple state machine DSL into a node from the graph DSL. In the body of the transformation, a new node is created and it is given the same name as the state.

The *transformation operation* (the one-to-one transformation) is a non-injective and non-surjective function. That is because there can be multiple *LUD* value concepts mapped to the same *BL* value concept (thus, the non-injectivity), and the same *BL* value concept type can be created by multiple transformation operations (thus, the image of the transformation operation is smaller than its co-domain, implying non-surjectivity). Moreover, the signature of the transformation operation is fixed. Its parameter types are the input type group, the output type group, and the *LUD* type concept. The returned type of the operation is the *BL* type concept.

For efficiency reasons, we cache the results of the transformation operations. That is because the same model transformation can be called multiple times in the course of performing a model transformation. Moreover, they can also be called when converting parameters and return values of the delegating *LUD* operations, and we do not want to modify the output value models anymore at that point.

```
transformation State2Node(GroupType#SimpleStateMachine# inputGroup, GroupType#Graph# outputGroup,
    ConceptType#State# state) returns ConceptType#Node# {
  ConceptType#Node# node = create ConceptType<Node>;
  node.set name(state.strValue);
  outputGroup.add to contents(node);

  return node;
}
```

Figure 8.9: A state from the simple state machine DSL transforms into a node in the graph DSL. This is an example of a transformation operation. One can notice API functions of MetaMod for creating model elements: *create ConceptType##*, *set name* on model elements and *add to contents* on groups.

There are cases when model transformations cannot be completed only with one-to-one transformations. For instance, a composite state machine transition transforms to a series of other transitions in the simple state machine. That is, each transition with a composite state as a source transforms into multiple transitions, one for each state in the source composite state. For these cases, we have created the *helper operations*. These can have any number of parameters. The signature of a helper operation for the creation of a transition is depicted in Figure 8.10. Again, the first two parameters of the operation are fixed and are the input and output type groups.

The third type of operation for model transformations, the *conversion operation*, is created specifically to convert parameters of the delegating *LUD* operations, both *in* and

```
helper CompTransAndStates2Trans(
    GroupType#CompositeStateMachine# inputGroup, GroupType#SimpleStateMachine# outputGroup,
    ConceptType#CompTransition# compTrans, ConceptType#State# sourceState, ConceptType#State# targetState) returns
    ConceptType#Transition#
```

Figure 8.10: A transition and two states in the composite state machine DSL transform to a transition in the simple state machine DSL. This is the signature of a helper operation.

out parameters. The conversion operation can be used to convert multiple parameters of the delegating *LUD* operation into one parameter of the *BL* operation, for instance. An example of a conversion operation is shown in Figure 8.11.

```
conversion Node2State(GroupType#Graph# inputGroup, GroupType#SimpleStateMachine# outputGroup, ConceptType#Node# node)
    returns ConceptType#State# {
  for (ConceptType#State# state : outputGroup.conceptsOfType(ConceptType#State#)) {
    if (node.strValue.equals(state.strValue)) { return state; }
  }

  return null;
}
```

Figure 8.11: A node from the graph DSL is converted into a state from the simple state machine DSL. This is an example of a conversion operation. One can notice API functions and MetaMod types we discussed in Section 4.2.

When conversion functions are called, in the conversion of *in* and *out* parameters of the delegating *LUD* operations, the output group (value model) is already available. That means that one can query the output value model from the conversion operation. This is what is happening in the example in Figure 8.11 as well. The states and the nodes in the input and output value model have the same name (because of the transformation operation in Figure 8.9), and that means that a node originates from a state with the same name. This is correct under the assumption that different nodes have different names in the simple state machine.

8.4.2 Operation signatures for Op_{LUD_BL}

An operation signature for delegating operations in MetaMod has other sections besides the formal parameters and return value. Firstly, there is a section, *@delegate*, where the delegated operation is specified. Secondly, there is a *@conversions in* section where the conversion operations are specified for the *in* parameters and a *@conversions out* section where the conversion operations are specified for the *out* parameters. Lastly, there is a *@precondition* section where constraints that need to hold on the actual parameters are specified.

A concrete example of how we handle operation signatures for Op_{LUD_BL} is shown in Figure 8.12. The figure defines how the operation *touchedStates* in the simple state machine DSL is delegating to the operation *touchedNodes* in the graph DSL. We see in the conversion section of the operation signature how the *in* parameter, *startState*, is transformed into *startNode*. The transformation operation itself is used for conversion. The output group, *outputGroup*, is used for type-checking only because the *State2Node* operation will not modify the output group, it will only return the node from the cache. Moreover, the *alreadyTouched* list is used both as *in* and as *out* parameter. For the *in* conversion, the *State2Node* transformation operation is used, while for the *out* conversion, the *Node2State* conversion operation is used.

```
@delegate touchedNodes(outputGroup, startNode, alreadyTouchedNodes)
@conversions in [ GroupType#Graph# outputGroup;
                  ConceptType#Node# startNode = State2Node(inputGroup, outputGroup, startState);
                  list<ConceptType#Node#> alreadyTouchedNodes = { =>
                    foreach state in alreadyTouched {
                      yield State2Node(inputGroup, outputGroup, state);
                    }
                  }().toList; ]
@conversions out [ alreadyTouched.clear;
                   for (ConceptType#Node# node : alreadyTouchedNodes) {
                     alreadyTouched.add(Node2State(outputGroup, inputGroup, node));
                   } ]
@precondition [ ]
operation touchedStates(GroupType#SimpleStateMachine# inputGroup, ConceptType#State# startState,
    list<ConceptType#State#> alreadyTouched) returns void
```

Figure 8.12: The *touchedStates* operation from the simple state machine DSL delegates to the *touchedNodes* operation from the graph DSL. The conversion of the parameters is handled in the *@conversions in* and *@conversions out* sections.

Note that in MetaMod, the conversion of parameters can be partially automated (not implemented currently) when the number of parameters, their multiplicity and their order coincides in the *LUD* operation and the reused *BL* operation. In that case, the type information of both the *BL* and *LUD* operations is known and MetaMod can look up transformation operations that take the *LUD* type and transform it to a *BL* type for *in* parameters. Only transformation operations are searched because they are one-to-one transformations. Moreover, transformation operations can be looked up for *out* parameter conversions as well. That is because the results of these transformation operations are cached and the source *LUD* value concept for a given *BL* value concept can be returned. If there are multiple source *LUD* value concepts that created the *BL* value concept, the first one could be returned.

Given that the output model of the transformation needs to fulfill the structural and user-defined constrains (as discussed in Section 8.3), MetaMod automatically checks these constraints on the output model. If the output model is not valid with respect to these constraints, the model transformation needs to be modified such that it creates a valid output model.

8.4.3 Implementation in MPS

The version of MetaMod where we implemented this reuse mechanism can be found on our Github project on branch *ReuseMethod2*. In the latest version of MetaMod, in the querying functions, the value model appears as a parameter, while in the version for delegated operations, the value model is not a parameter. The value model is considered fixed and it is either the input value model or the output value model. Moreover, the multi-operation feature is not present in this version of MetaMod.

The way we tackled delegated operations in the implementation is the following. The body of the delegating operation first checks whether the precondition holds, and then performs the conversion of the *in* parameters, according to the *@conversions in* section of the delegating operation. Afterwards, the call to the delegated operation with the converted parameters is made. The last step is to convert the *out* parameters and return the result, if any. These conversions are also taken from the *conversions out* section of the delegating operation.

The previous steps are implemented using the template generation language of MPS. The way that the call to the delegated operation is made in this template is depicted in Figure 8.13. Depending on whether the return type of the operation is void or not, one or the other statement in the figure is generated.

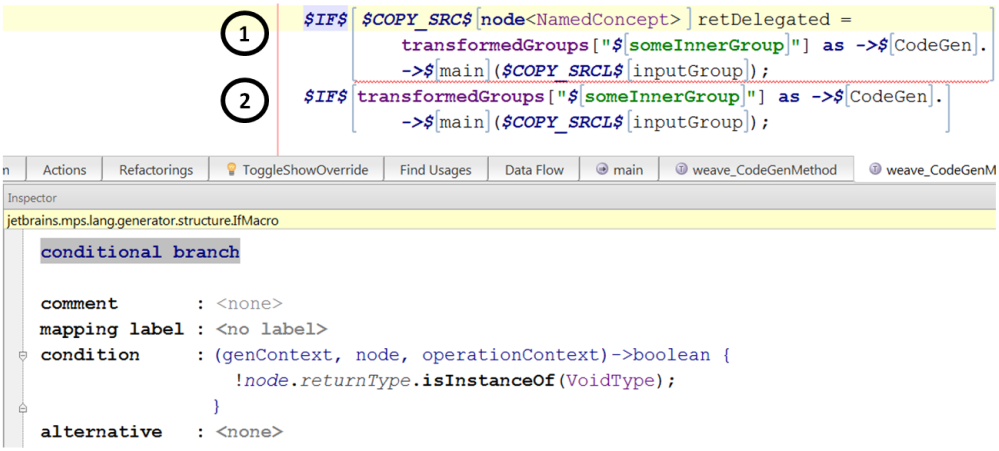


Figure 8.13: Template for placing a call to the delegated operation in the body of the delegating operation. The condition for generating the first statement (1) is that the return type of the delegating operation is not void. For the opposite condition, when the return type is void, the second statement (2) is generated.

8.5 Discussion

In this section we discuss trade-offs, observations, tool support, advantages and limitations of our approach.

Trade-offs Because the model transformation should have a small complexity, the benefit of reusing the operations from *BL* should outweigh the effort of defining model transformations and writing operation signatures (which can be partly automated). Translating concepts from one language fragment to the other should be easier than implementing algorithms for a non-trivial problem domain. Take, for instance, the case of simple state machines and composite state machines. The model checking operations defined for the simple state machines are not trivial, but the transformation from composite state machines to simple state machines is straightforward. Moreover, the transformation from a simple state in the composite state machine to a state in the simple state machine is one-to-one, easing also the conversions of the parameters, as most parameters in model checking operations are states. Thus, using our approach required less effort than developing everything from scratch in this case.

Although our approach introduces a new level of indirection, again, the benefit of reusing the operations from *BL* should outweigh the performance penalty. The application of the model transformation is done once, and used by all the operations. Moreover, the fact that the computations are done at the right level of abstraction should outweigh the extra calls to the delegated operation and the conversion functions (some of which can

be handled from cache). For instance, model checking operations made directly on the composite state machine would need to keep track of the composite states at all steps, adding overhead.

Observations One of the highlights of our approach is the fact that reused operations from the base DSL, although executed on the *BL* value model representation, are called from the *LUD* value model. Therefore, there is an interaction between the *BL* value model representation and the *LUD* value model. The operations from Op_{LUD_BL} are called on concepts from the *LUD* value model, and delegated to the *BL* value model, which also needs to return the *out* parameters back to the *LUD* value model.

Another observation is that the approach presented in this chapter would benefit greatly if the underlying language workbench had a way to modularize the metamodels. That would result in using smaller metamodels in the construction of the language, and, as a result, smaller metamodels would be available for use in our approach. If base DSLs are big and monolithic, and if one needs only a part of that DSL, transforming to the entire DSL is unnecessary, if not also impossible (think of a complex expression language with all possible kinds of expressions in it, e.g. bitwise operations, set operations and interval operations).

In addition, our approach is generic because it could be implemented in other language workbenches as well. As long as there exists a way to represent metamodels in the language workbench, and a model transformation language, this approach can be implemented there as well.

Tool support Tool support plays an important role in our approach. The tools should offer a convenient way of defining the signatures of the delegating operations, together with the conversions. In MetaMod, we did that by creating dedicated sections for the conversions, one for defining the delegated operation and one for preconditions.

Advantages and limitations The major advantage of the approach is that it leverages previously developed auxiliary DSL aspects. One can treat a stable DSL (even legacy DSL) as a base DSL. Moreover, by transforming to a base DSL and executing operations on the *BL* value model representation, operations are executed at the right level of abstraction. Take, for instance, the simple state machine and the composite state machine. The model checking operations can be performed irrespective of the composite states. Finally, the metamodels of the base DSL and the DSL under development can vary significantly. This is important for DSLs, as they are designed for specific usage scenarios, and capturing that in the DSL, and ultimately in the metamodel, results in variations of the metamodels even for DSLs in the same domain.

Furthermore, we have discovered cases of reuse that, to the best of our knowledge, cannot be covered by other approaches for reuse. For instance, reusing the operations of the simple state machine DSL in the composite state machine DSL is not possible with other approaches for reuse (see Section 8.6).

One concern that might arise with the approach is that it introduces a new dependency in *LUD*, that to *BL*. That can lead to problems if *BL* changes or if *BL* has errors. That is why, one should use a stable language as *BL*.

Another concern with the approach arises when an operation executed on the *BL* value model updates the *BL* value model. Such an update is not propagated to the *LUD* value model. That means that one cannot reuse *BL* operations that modify the *BL* value

model if one needs those modifications visible in the *LUD* value model. On the other hand, if the updates do not need to be propagated back, this is possible.

8.6 Related work

There are a multitude of approaches for reuse in software, in general, and in DSLs, in particular. In this section, we focus on a few major directions in the related work: transformations to semantic domains, genericity, templating, inheritance, composition, model transformation reuse, and more general works. We treat model transformation reuse separately because it represents a considerable part of the related work.

Transformations to semantic domains The work on semantic anchoring of Chen et al. [21] is using model transformations from a DSL under development to a *semantic unit*, a formalism that has a formal and precise semantics. This transformation is called semantic anchoring because the resulting value model can be interpreted based on the operations defined by the semantic unit. Compared to our approach, there is no interaction between operations defined directly on the DSL under development and the semantic domain to which is transforms. The operations defined on the semantic domain are executed in isolation from the DSL under development.

Genericity One relevant approach is the work on generic types in metaDepth by Juan de Lara et al. [33]. In their approach they use so-called *concepts* to define requirements for metamodels. They have two types of concepts: structural concepts and hybrid concepts. The structural concepts define structural requirements only and can be bound to metamodels that embed the same structure as the structural concept. On the other hand, in a hybrid concept, they delay some structural decisions by means of defining operations that need to be implemented by the user of the hybrid concept. In the end, both types of concepts have code generation, interpretation or simulation attached to them, which can be reused once the concepts have been bound to a metamodel. In contrast with our mechanism, any of the structural elements of the hybrid concept have to appear without modification in the binding metamodel. The DSL engineer needs to anticipate what elements will vary and what elements will be fixed in the binding metamodel. In our case, any of the structural elements can have different representations in the *LUD*. Furthermore, with our reuse mechanism, one can treat legacy DSLs as base DSLs because the DSLs do not need to be specially designed to be base DSLs. Another difference with metaDepth is that a hybrid concept is not an actual DSL unit because one cannot attach an editor to it, since some structural decisions are encoded in operations (and thus, one cannot visualize the *LUD* value models as *BL* value models, for instance). On the other hand, our mechanism does not permit to have operations that modify the base value models if we want the modifications propagated back to the *LUD* value model.

Templating In the UML world, one of the most related works is that on the templating mechanism [115]. The template contains a signature where formal template parameters are specified and the substitution of the formal template parameters by actual template parameters is called binding. After the binding is made, all different variants are going to reuse the operations associated to the metamodel. This is one other way of approaching reuse, not only of operations, but also of structure. On the other hand, that means there

cannot be any structural differences among the two DSLs, which is important given that DSLs are created for specific usage scenarios.

Inheritance There are also approaches that use inheritance mechanisms in the context of domain-specific languages. In Monticore, Krahn et al. [84] introduce language inheritance and embedding. With these two mechanisms, besides the extension of the concrete and abstract syntax, also the operations of the DSL are going to be reused in the resulting DSL. Another work that uses both specialization and templates is that on the Meta Modeling Language [25]. Furthermore, Varro et al. [153] define inheritance relationships not only for classes, but also for associations and packages. There are also language workbenches that approach reuse via inheritance. For instance, Jetbrains MPS allows to extend another language and also to extend separate concepts from the extended language [159]. In all these cases, the structural variations among the parent DSL and the corresponding part in the child DSL cannot be too sizeable because the child is inheriting the structure of the parent and can modify it to a very small extent.

Composition The work on Melange by Degueule et al. [35] focuses on reusable and modular DSLs. They achieve this with a set of assembly operators (merging and weaving) and a set of customization operators (slicing, merging and inheritance). For all these operators, the operational semantics of the composed languages is based on the component languages. Unlike in our case, they reuse the operations associated to the exact components that go into the resulting language.

An interesting mechanism, described by Berg et al. [14], is focused on the composition of separate metamodels with the goal of reusing the operational semantics of the component metamodels from the composition. They do so by creating an unification model for the metamodels of the DSLs involved in the composition and by creating a linking model for the models of the DSLs. In their mechanism, proxy classes are added to the composition of the metamodels. This way, the operational semantics of the two languages can interact at runtime. One limitation that we see is that linking on the model side can be tedious for large models.

The current language workbenches offer various mechanisms for composition of the different aspects of a DSL (concrete syntax, transformations, etc.) [47]. Again, the structural variations possible between the original DSL and the DSL in the composition is much more restrictive than in our approach.

Model transformation reuse Another set of related work is that on model transformation reuse. There are many types of reuse for model transformations [88], and the type that we are covering with our approach is that on generic transformations with the source metamodel as the generic part.

One work on generic meta-model transformations is that of Varro et al. [154]. They define generic meta-model transformations (higher-order model transformations - HOTs) by using type variables instead of concrete model types. Then, by using meta-transformations, they transform the HOT into a first-order transformation by replacing type variables with concrete model types. The structural variability between the metamodels cannot be that big in this case.

Another work on generic model transformations is that of Cuadrado et al. [30]. They use the notion of a structural concept from de Lara et al. [33], but they allow features in bindings to be bound to OCL expressions. This reuse mechanism is more limited than

ours, because it requires one-to-one mappings for all elements in the base language, for instance.

The work on model typing of Steel et al. [134] is relevant to us as well. They define model types as a set of object types, or, more precisely, as a set of MOF classes and the references that they contain. These model types can be used as types in model transformations, thus making the transformations applicable to any of the object types in the set of the model type. An object type in this context is a metamodel. A more recent work by Chechik et al. [20] generalizes the model typing work by introducing coercive subtyping which allows reuse for transformations that tolerate specific violations of the model subtyping. Our mechanism allows for more variation among the reusing DSL fragments, as long as a model transformation resulting in a valid *BL* value model can be created.

Another work that builds on model typing and that treats the reuse of model transformations is that of Moha et al. [108]. They first extend the notion of model subtyping to make it more inclusive, and present a mechanism of model transformation reuse. This mechanism requires building model transformations for a generic metamodel that contains only the necessary information for the model transformations. Then aspects and derived properties are weaved into metamodels that reuse the model transformations such that they become a subtype of the generic metamodel. In comparison, our approach does not require interfering with the structure of the reusing DSL metamodel, making it less invasive.

General A more general work on reuse is that of Kienzle et al. [76]. They note that reuse of artifacts always involves a combination of one or more of the following interfaces: variation (available variants that the artifact encapsulates), customization (how to tailor a generic artifact to a specific application) and usage (structural and behavioural elements accessible from the artifact). They also apply this work to DSLs [125], but again, the reused behaviour is applicable on structures identical to the reused structure.

8.7 Conclusions

In this chapter, we have presented another approach for the reuse of fragments of DSL aspects, called the approach of delegated operations. The definition of the approach consists of model transformations from a DSL under development to a base DSL and of delegating operations for the DSL under development to the base DSL. The advantages of our approach are the following: it leverages previously developed DSLs, the metamodels of the two DSLs can differ significantly, and the operations are executed at the right level of abstraction. There is also a limitation of the approach: updates that are made on the base value model representation by reused operations are not propagated back to the value model of the DSL under development.

We also presented an implementation of this approach in MetaMod together with a few examples of DSLs that made use of the approach. One could see how the approach helped with reusing DSL operations associated to the graph DSL and the simple state machine DSL, respectively.

Our approach opens up new possibilities of reuse, because the corresponding parts of the metamodels of the two DSLs can differ structurally more than in other approaches. This allows us to handle reuse situations that are impossible to handle with other approaches.

Chapter 9

Evaluation

We evaluated MetaMod with the help of various DSLs. These DSLs play a double role in the evaluation, that of highlighting features for modularity and reuse of MetaMod and that of demonstrating the feasibility of building these DSLs with MetaMod. The DSLs range from small sizes to large sizes, where the size of a DSL is given by the number of concepts and relations in the metamodel and the number of operations in the processing unit. They are also at different levels on the scale from declarative to imperative, and they target different domains.

The chapter is organized around the DSLs that we created in MetaMod. We chose this organization because each DSL shows different features for modularity and reuse, and each DSL touches on slightly different aspects of the feasibility of building a DSL with MetaMod. For each DSL, we present the design decisions we made for the DSL, we highlight the main features of MetaMod that the DSL has made use of and we reflect on the main advantages and challenges we encountered in the process of implementing the DSLs. We conclude the chapter by reflecting on all of our DSL designs and implementations, and we highlight one commonly occurring pattern in the reusable DSLs, that of a hot-spot DSL. We noticed this is one of the most helpful patterns when building reusable DSLs in particular, and we assume that it can be useful for the creation of most reusable DSLs as well.

9.1 Kaja language

The Kaja language was presented in Chapter 6. Kaja has a mix of declarative and imperative constructs, but it is more inclined towards imperative constructs (e.g., the while loop, the repeat loop and the procedure). This language is a translation of the Kaja language from the samples project of MPS into MetaMod. The implementation of this DSL in MetaMod can be found in the Github repository in Solution *Kaja*, model *kaja*. In the next three paragraphs, we give three reasons why we re-implemented Kaja.

The main purpose of implementing this language in MetaMod was to highlight MetaMod's support for modularity and reuse at the level of value models. For instance,

there was no need to introduce definitions of routines and calls to routines, and definitions of libraries and imports of libraries in the implementation in MetaMod. To support these generic modularity mechanisms at the value model level, we made use of the fragment abstractions and applications, and the group organization of the models.

At the same time, Kaja is also an example of a language that offers basic imperative language constructs, such as while loops, if statements, procedures, and libraries. This gets us to the auxiliary purpose of implementing Kaja, that of evaluating MetaMod using a DSL with an imperative flavor to it.

Incidentally, because this DSL was implemented at a time when we did not have multi-operations and overriding in MetaMod, it also shows the difficulties that the lack of these features bring. The lack of multi-operations and overriding made the implementation very tedious, because in the processing units of MetaMod, whenever we needed to call the operations to evaluate commands or logical expressions, we had to write a switch on the type of command or logical expression. For that, we wrote an operation called *dispatchCommand* and one called *dispatchLogicalExpression*, that checked the runtime type of the command and that of the logical expression parameter and called the respective operations. This solution brought new problems though, because in this situation we had to implement operations that would be fit for reused groups in the top-most group *Kaja*. For instance, we had to write the operation evaluating the *and* logical expression in the processing unit of *Kaja* because *and* calls the dispatch operation for logical expressions on both the left-hand side expression and the right-hand side expression, and only the dispatch operation of Kaja knows about all the logical expressions. Not being able to write operations in the appropriate group hinders modularity and reuse to a great extent.

The difficulties in implementation stemming from not having the multi-operations and the possibility to override them has also shaped the organization of the DSL units of Kaja in the implementation in MetaMod. The DSL units that form Kaja are depicted in Figure 9.1. For instance, we did not reuse *Commands* in *GridRobotCommands* (commands to move the robot and to pick marks), *ComputationalCommands* (commands like *while* or *if*), or *GenericBuilderCommands* (commands to build the grid). These commands all come together under *Kaja*. If, on the other hand, we had multi-operations and overriding, we would have reused *Commands* in all previous groups related to commands, so that we can override an execute command operation, for instance. Then, *Commands* would have been a central DSL unit of Kaja, because many other groups from Kaja would use model elements from it and would override operations of this DSL unit.

To give you an idea of the scale of the Kaja DSL, here are a few statistics (pertaining to the metamodels only, and not the value models):

- Number of concepts: 39
- Number of relations: 41
- Number of operations: 71
- Number of DSL units: 8

The next section is going to be about the reusable expression language we implemented in MetaMod. The Kaja DSL itself makes use of a subset of logical expressions, and we implemented this subset together with Kaja. That is because at the moment we implemented Kaja, the expression DSL was not yet implemented.

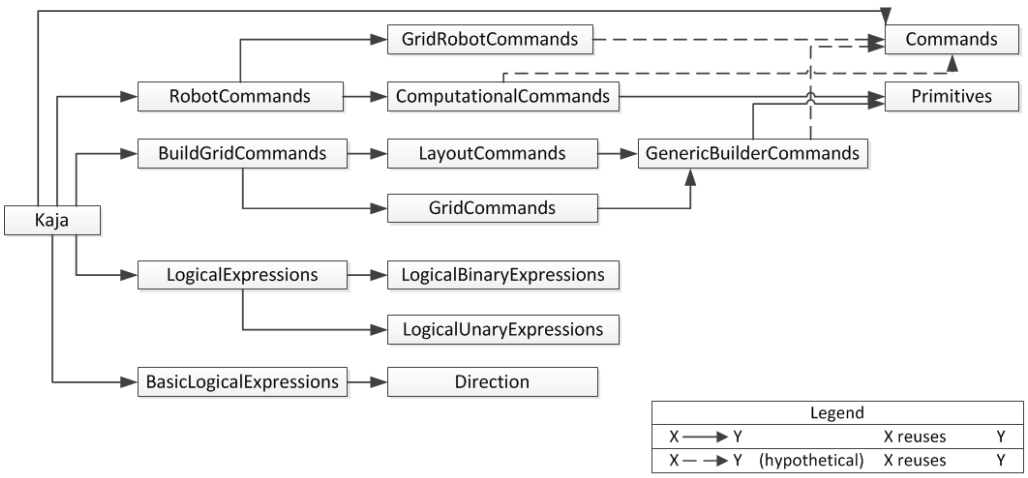


Figure 9.1: The organization of DSL units in Kaja. The dashed arrows represent the arrows that would be present if the implementation had multi-operations and overriding.

9.2 Expression language

In this section we describe the iets3 expression DSL, its implementation in MPS by the mbeddr team and the implementation of a subset that we did in MetaMod. Examples from this DSL are also used in Chapter 5 and they highlight the use of features such as multi-operations, multiple dynamic dispatch, and inheritance of concept types and group types. Here we make a more high-level presentation of the expression DSL. This is also an example of a DSL with an imperative flavor. The MetaMod reimplementation can be found in the Github repository in Solution *Expressions*, model *iets3ExpressionsV1*.

9.2.1 The iets3 expression DSL

The iets3 project [71] is a specification environment for software systems built by the mbeddr team. On top of this specification environment, DSL engineers build concrete tools, where the components of iets3 are reused. One of the core components of iets3 is its expression DSL. The iets3 expression DSL consists of seven language modules: *base*, *simple types*, *lambda*, *collections*, *path*, *repl*, *tests* and *toplevel*. This is a comprehensive expression DSL because it covers features encountered in a large pool of programming languages (among the less mainstream features we mention path expressions, option types, attempt types, closures, higher-order functions, unit tests, etc.).

The iets3 project was designed for reuse in real-world DSLs that the mbeddr team is building for their clients. As a consequence, the iets3 expression DSL was designed with reuse in mind. The DSL offers an implementation for most important language aspects; structure, behavior, editor, constraints, type system (a strict one to ensure safety) and interpretation.

Many of these DSL units are coarse grained and also depend on each other (some dependencies also give rise to cycles), essentially meaning that when you want to reuse one part of a DSL unit, you have to reuse the entire DSL unit, plus the dependencies that it has. For instance, the *base* DSL unit contains many other DSL units, such as alternatives expressions, binary arithmetic expressions, unary arithmetic expressions, attempt types,

contracts, option types, validity types, etc. The simple types, on the other hand, contain only three DSL units, the *String*, *Numeric* and *Boolean*, but they have a dependency on the *base* DSL unit, which means that reusing the *Boolean* DSL unit would result in reusing all the simple types and the base expressions. Nonetheless, the undesired concepts from the dependencies can be forbidden in models by imposing additional constraints, for instance. The dependencies between the simple types, the lambda and the base expression DSL units is shown in Figure 9.2.

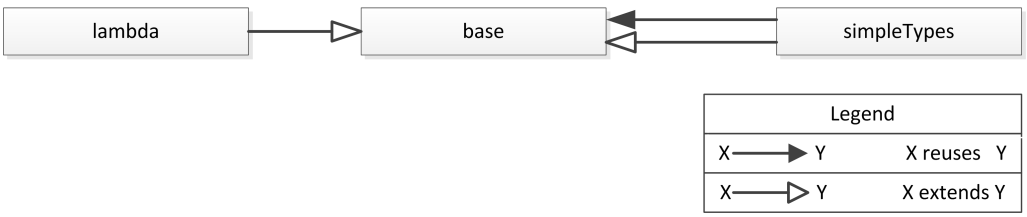


Figure 9.2: The organization of three expression DSL units in MPS. Reusing and extension in MPS were introduced in Chapter 2.

9.2.2 The MetaMod expression language

Our goals with the reimplementation of the iets3 expression language in MetaMod were (1) to provide smaller units that can be reused in isolation (without surreptitiously bringing in lots of dependencies), and (2) to allow reuse of combinations of those smaller units (by predefining such combinations; cf. Figure 9.6). Moreover, we wanted to see how larger implementations from MPS could be captured with MetaMod and to what degree modularity features of MetaMod ease the creation process.

We considered the base, simple types and lambda DSL units to be the most relevant DSL units. We have completely translated the concepts from the simple DSL units and the lambda DSL unit, but we have translated only part of the concepts of the *base* DSL unit. We selected those concepts that we encountered in many other DSLs we worked with. We have reimplemented all the language aspects from MPS, with the exception of the editor (because this would require a dedicated meta-language), and the interpreter for the lambda DSL unit (because of time constraints). From a visual inspection on the expression language in MPS, the selection of concepts and operations we made for the translation to MetaMod covers the most important MPS features used in the creation of the expression language in MPS.

We tried to stay as close as possible to the intention of the original (in most cases we have one-to-one translation of concepts), and we restricted ourselves to reorganization into smaller DSL units. The additional DSL units that are part of the three top-level DSL units (base, simple types and lambda) can be seen in Figures 9.3 through 9.5. Note that we kept the top-level DSL units from the iets3 expression language, but we have further divided their content into additional DSL units.

In MetaMod, most operations for behavior, constraints, type system and interpretation are captured in the top-level DSL units: base, simple types and lambdas. On the other hand, there are still important operations of the type system and interpretation aspects captured in the combinations of the base expressions with the simple types. The features of MetaMod that allowed redefining the type system and the interpretation in the combinations are explained in Chapter 5. The interesting combinations that lie at the

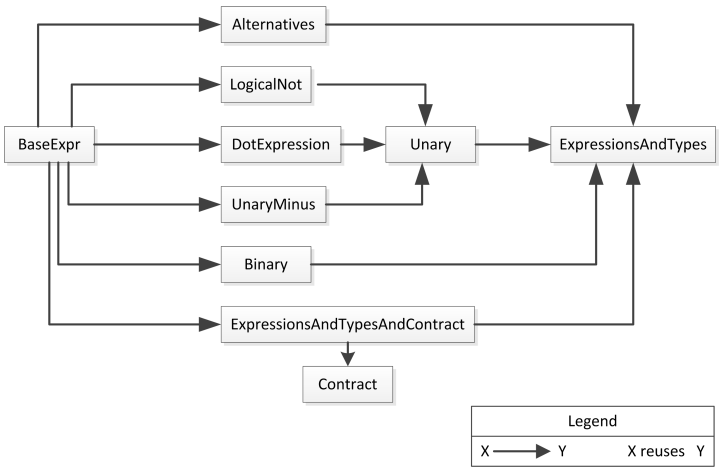


Figure 9.3: The DSL units forming the DSL unit for base expressions in the MetaMod implementation.

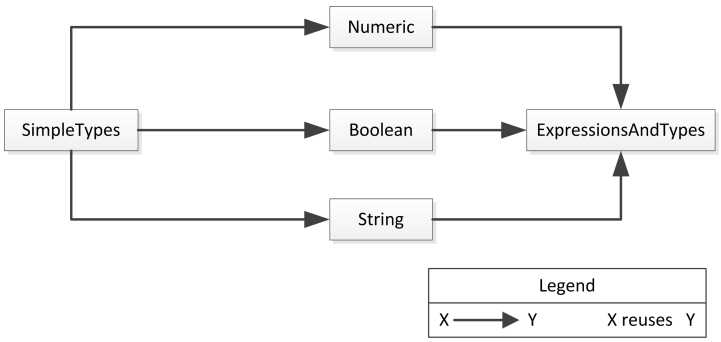


Figure 9.4: The DSL units forming the DSL unit for simple types in the MetaMod implementation.

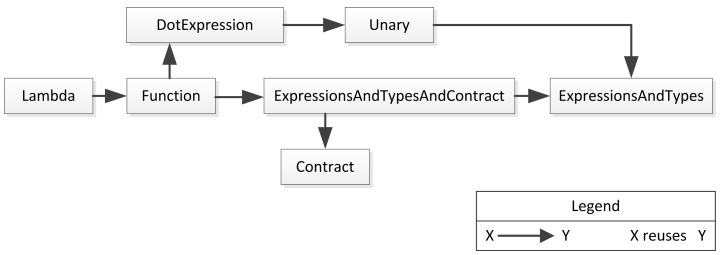


Figure 9.5: The DSL units forming the DSL unit for lambda expressions in the MetaMod implementation.

intersection of base expressions and simple types are captured in Figure 9.6. For an example of a metamodel, the metamodel of the top-level group, *BaseExprAndSimpleTypes*, is depicted in Figure 5.5 in Chapter 5.

To give you an idea of the scale of the rewritten expression DSL we obtained in

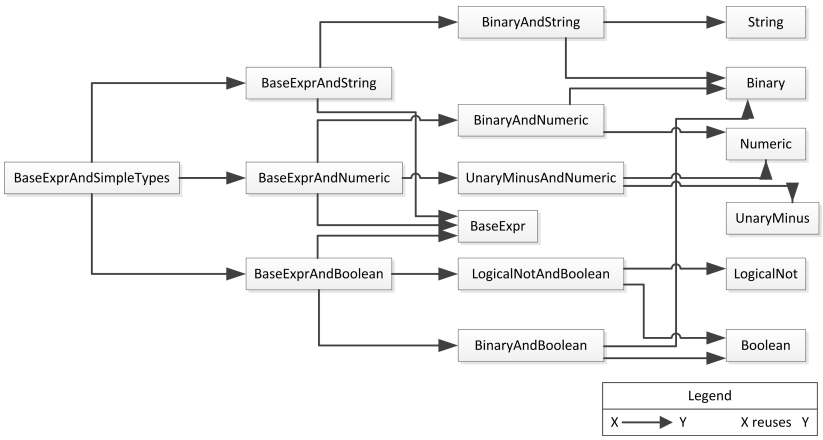


Figure 9.6: The most relevant combinations among base expressions and simple types are captured in separate DSL units and united under DSL unit *BaseExprAndSimpleTypes*. This is part of the MetaMod implementation.

MetaMod, here are a few statistics:

- Number of concepts: 90
- Number of relations: 41
- Number of operations: 300
- Number of DSL units: 34

The expression language itself was built in a modular way, through reuse of DSL units. In Figures 9.3, 9.4, 9.5 and 9.6, one can see for each DSL unit what other DSL units it reuses. Thus, the expression DSL is an example of building a modular language in MetaMod. Moreover, these DSL units have been built with the goal to be reusable in other DSLs. Thus, one can see examples of how to design reusable DSL units with the expression language.

Modularity Modularity comes from decomposing the three original DSLs into smaller DSL units. Most of the DSL units communicate through the core expressions and types, in the sense that they reuse the core *ExpressionsAndTypes* group and they override operations from the core. Whenever these operations are called, the appropriate override available in the context of the processed model is called. For instance, the type of an expression is defined in an operation in the core expressions and types group. Whenever extending an expression, this operation can be overridden for the new expression and whenever we reuse the expression in a new group, the operation can be changed again; cf. Section 5.1.5.2.

Reuse We have a core DSL unit, *ExpressionsAndTypes*, that is being reused by almost all other DSL units, directly or indirectly. The DSL engineer interacts with the reused core DSL unit by creating DSL units that extend the reused DSL unit. The extension takes the form of (1) reusing groups that add relations to the core group, and (2) processing units that override operations or add new operations to the processing units of the core

group. Both of these happen in the context of the extending DSL units. In that respect, the core DSL unit behaves like a hot-spot or a nucleus.

Trade-offs Smaller DSL units that are built for reuse also come with trade-offs. One question is how many combinations of smaller DSL units should the DSL engineer create? Some could argue that predefining certain combinations is just a convenience, since the DSL engineers could also do that afterwards. One reason is that a handful of similar combinations are most likely going to be created by many DSL engineers, so it is more efficient to provide such combinations in a central place. Another reason is that often, operations can be written only by combining groups. For instance, the combination of numerics and binary operations needs to type check and interpret the effects of adding two numerics, subtracting two numerics, etc.

Because of these two reason, the DSL engineer of the reusable DSL units needs to tackle the hardest combinations to implement, together with the overall combination. This way, the reusing DSL engineers know how the combinations can be tackled in case they want to reuse other parts that are not covered by smaller combinations. For instance, in the combination between base expressions and simple types, we decided to combine separately the binary DSL unit with numerics, the unary DSL unit with numerics and so on. We think these are the most common situations occurring in practice, and they are also the hardest combinations to implement because many type checking and interpretation operations can only be written in these combinations. Moreover, we also create the overall combination, that in group *BaseExprAndSimpleTypes*, where we cover everything that was not covered by the smaller combinations.

Discussion If an organization similar to the one of MetaMod would be chosen for the original iets3 expression DSL implementation in MPS, the DSL engineers would need to introduce concepts only for implementation purposes to create the combinations in MPS, as opposed to the implementation in MetaMod. For instance, consider the combination between alternatives and simple types in *BaseExprAndSimpleTypes*. The *AltOption* concept (representing an option of the alternatives operation), that was initially defined in the *Alternatives* group, needs operation *getAllOtherwiseSiblings* in the combination, as part of its behavior aspect. The problem in MPS is that one can not add operations to the behavior aspect of a reused concept unless a new concept extending the reused concept is created; thus, this results in adding a concept only for implementation purposes to the structure. In MetaMod, on the other hand, there is no need to create this new concept. The operation *getAllOtherwiseSiblings* can be simply added to the processing unit of *BaseExprAndSimpleTypes* in MetaMod.

A more detailed discussion on the benefits that MetaMod offers in this re-implementation of the iets3 expression language can be found in Section 5.1.6.1. There, we make a comparison between the implementations in MetaMod and in MPS.

One challenge that we faced in this exercise, was a performance one, both in the time to generate code from the processing units and the time to run the processing units on the value models. There are many parts of the current MetaMod implementation that can be optimized (e.g., the way we query the models is not efficient), and we detail some of the possible optimizations in future work, Section 10.3.

9.3 Bootstrapping

In this section, we present how we created DSL units that define a subset of MetaMod in MetaMod itself. We had four reasons to do this exercise. These reasons also point out what does bootstrapping bring to the evaluation. Firstly, bootstrapping is a good form of testing the bootstrapped application. It demonstrates that MetaMod is mature enough to implement a non-trivial application, MetaMod itself. Secondly, it is an exercise where we highlight the multilevel nature of MetaMod. In this exercise we define a subset, $MetaMod_b$, of MetaMod in MetaMod (we subscript with b the bootstrapped subset of MetaMod and its elements); we define a DSL in $MetaMod_b$, called *DSLMetaMod*; and we define a model in *DSLMetaMod*. Thirdly, it is an intellectual challenge as well because it is non-trivial to design and implement in MetaMod the different meta-languages of MetaMod. Fourthly, this exercise is a first step towards making MetaMod 'self-supporting', that is, a step towards using MetaMod outside of MPS.

Before we go on to describe the steps we took in the implementation, we make three remarks. Firstly, what is special about the implementation of the bootstrapping exercise is that we are able to build not just $MetaMod_b$ (a subset of MetaMod in MetaMod), but also a DSL and a model of this DSL using $MetaMod_b$. With all the other DSLs we present in this chapter, one can define conforming models. That is also the case for $MetaMod_b$, but those models are actually also metamodels, and hence they can have conforming models as well. Other platforms would require that one first loads $MetaMod_b$ into the environment [82], and then she would be able to build the DSL and the model. This is an advantage of having multilevel meta-tools. Secondly, $MetaMod_b$ is not the full implementation of MetaMod, it is just a subset. Through this exercise, we show what it would take to implement MetaMod in MetaMod, but we do not perform it completely, and we use just a subset of elements from MetaMod. Thirdly, MetaMod itself is a collection of meta-languages, which means that the implementation of MetaMod in MetaMod is an example of implementing a modular language.

Next, we define the high-level process of bootstrapping MetaMod. Because DSL technology takes its roots from programming languages [48], we are going to make a parallel to bootstrapping a compiler.

T-diagrams The process of implementing a compiler in the language that the compiler knows how to compile, is called bootstrapping a compiler [144]. We now illustrate traditional bootstrapping schemes using T-diagrams. We depict a T-diagram in Figure 9.7. The T-diagram represents a compiler for the left-hand side language (called the source language, S), written in the bottom language (called the implementation language, I) and generating the right-hand side language (called the target language, T). We will refer to this T-diagram as *SIT*. A traditional bootstrapping scheme can be explained with T-diagrams and is depicted in Figure 9.8. In a traditional scheme, we want to get to the situation in step three, where we have a full language implementation F, written in a native language, N, and producing native language code, N. We get to this result in two steps. In step one, we write a compiler in N for a subset of the language, S, that produces N. In step two, we write a compiler for the full language F in the subset S, generating code in N. Feeding *FSN* (step two) to *SNN* (step one), we get *FNN* (step three). After step three, FNN is a 'fixpoint' of FNN, meaning that this compiler can be used to maintain itself (without having SNN and FSN).

Using a process that resembles that of bootstrapping a compiler, we defined the meta-languages of MetaMod in MetaMod itself. Figure 9.9 depicts the two steps that we

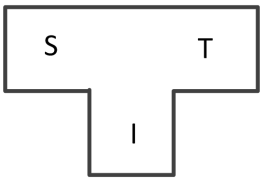


Figure 9.7: A T-diagram from compiler technology.

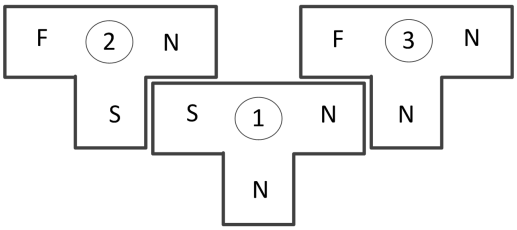


Figure 9.8: The three T-diagrams involved in the traditional bootstrapping scheme. The number inside the circle in the middle of the diagram represents the step number.

undertook for the bootstrapping and the resulting third step. In step one, we fully built MetaMod using MPS (as explained in the previous chapters of the thesis), by generating Java from the input MetaMod DSL definitions and conforming models. Our first step differs from traditional bootstrapping schemes in two ways. Firstly, we did not build a subset of MetaMod in MPS, but we built it fully in MPS. This is because we only want to demonstrate bootstrapping, as opposed to implementing it for practical reasons. Secondly, we did not use Java as an implementation language, but instead used the MPS platform (which in the end transforms everything to Java). Now, going back to step two in the bootstrapping process, the step illustrated in this chapter, we built a subset of MetaMod using MetaMod itself, by generating Java from the input subset of MetaMod. Again, here we built only a subset of MetaMod to demonstrate that bootstrapping is possible with MetaMod. Finally, step three is the output of feeding step two to step one. We obtained a compiler for a subset of MetaMod as a source language, and Java as the target language, by using Java as an implementation language. This process can be continued by adding features to the subset of MetaMod we already created, MM_b.

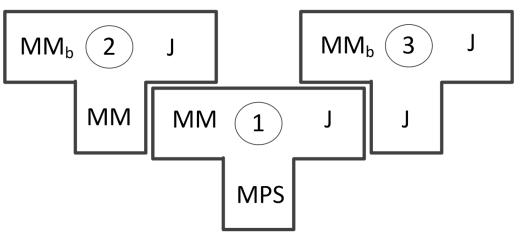


Figure 9.9: The three T-diagrams involved in the bootstrapping of MetaMod. The number inside the circle in the middle of the diagram represents the step number.

Subset of MetaMod The subset, $MetaMod_b$, of MetaMod we implemented in step two of the bootstrapping process (denoted as MM_b in Figure 9.9) consists of two models, $MetaModModels_b$ and $MetaModPUs_b$. Model $MetaModModels_b$ contains the elements of the group extension of the core meta-metamodel of MetaMod (see Section 4.1 and Figure 4.9). Thus, we do not implement the fragment abstractions and applications in this subset. Model $MetaModPUs_b$, on the other hand, represents a subset of the constructs in the processing unit of MetaMod. Figure 9.10 shows model $MetaModPUs_b$ and Figure 9.11 shows the groups that make up $MetaModPUs_b$. One can notice that we did not introduce aspects in $MetaModPUs_b$. So, we constructed a minimalistic version of the processing units. Operations are also minimalistic because they have parameters, a body made of statements, and a return type only. We did not implement the multi-operation or the overriding annotations, for instance. This minimalistic version of the processing units looses modularity and reuse features, but that is not of concern for the purpose of this exercise. Furthermore, we introduced two types of statements, a logging statement (for printing strings on the console) and an expression statement (a statement made of an expression). We also introduced three types of expressions, an operation that returns value concepts that are of a given type (the $conceptsOfType_b$ operation), an operation that can be applied on operation $conceptsOfType_b$ and that returns the first value concept, and an operation call expression. These are the basic constructs from the processing units of MetaMod that we used in step two of the bootstrapping process.

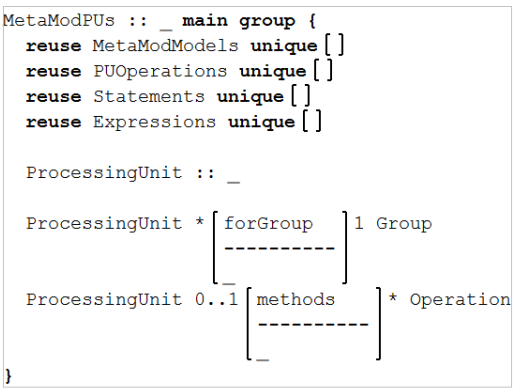


Figure 9.10: Definition of $MetaModPUs_b$, as a group.

MetaMod generates Java code from the processing units, and so $MetaMod_b$ needs to generate Java code as well from the processing units of the DSLs implemented in $MetaMod_b$. The combination of "quotations" and "openapi" from MPS with the processing units (we discussed this combination in Section 4.2.5) allowed us to easily write code generation using the processing units, while at the same time having the modularity mechanism provided by the processing units. Figure 9.12 shows an operation that generates an expression for a compact function invocation, which is an invocation of a closure literal. This is what MetaMod generates from the *conceptsOfType* operation.

Multilevel nature of MetaMod Now that we have built a subset, $MetaMod_b$, of MetaMod in MetaMod, we also construct a DSL in $MetaMod_b$, *DSLMetaMod*, and a model in this DSL, *ProgramDSLMetaMod*. As mentioned in the motivation, the bootstrapping example also showcases the multilevel nature of MetaMod; there are three

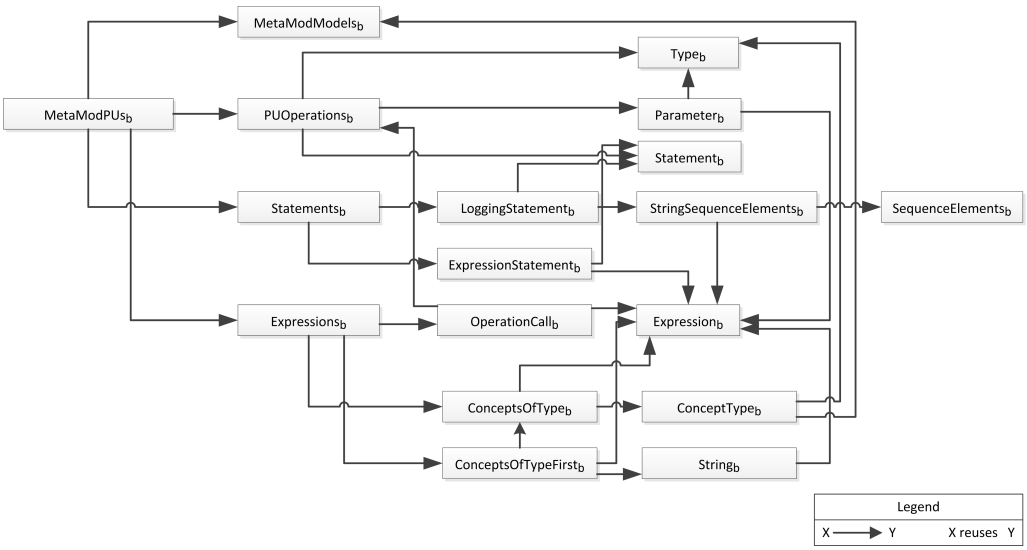


Figure 9.11: All groups that play a role in $MetaModPUs_b$, directly or indirectly, via reuse relationships.

```
PU_CreateConceptsOfType_ConceptsOfType
for group ConceptsOfType
aspect CreateConceptsOfType
       reuses CreateExpression

@override createExpression
operation createExpression(GroupType#ConceptsOfType# inputGroup, ConceptType#ConceptsOfType# conceptsOfType,
    node<ClassConcept> languageMetamodel, TransformationMaps transfMap) returns node<CompactInvokeFunctionExpression>
    {
  node<CompactInvokeFunctionExpression> compactInvocation = <{ =>
        yieldAll inputGroup.getRepresentativesNamedConcepts().
          where({~it => it.conformsTo.isInstanceOf(RefToNamedConcept) &&
          it.conformsTo : RefToNamedConcept.ref.name.equals(
          "$(  conceptsOfType.@src#argument# in (inputGroup).first.@src#concept# in (inputGroup).first.strValue
          )$"); }); }()>;

  return compactInvocation;
}
```

Figure 9.12: An example of code generation in MetaMod with processing units that use quotations from MPS.

levels of conformance between $MetaMod_b$, *DSLMetaMod*, and *ProgramDSLMetaMod*. In a non-multilevel language workbench, one would be able to write just $MetaMod_b$ and *DSLMetaMod*.

Levels of conformance Figure 9.13 defines the three levels of conformance involved in the example: level two ($MetaMod_b$), level one (*DSLMetaMod*), and level zero (*ProgramDSLMetaMod*). As already discussed, there are two main parts to level two: $MetaModModels_b$ and $MetaModPUs_b$. Then, at level one we build a value model of the type model from level two. This has also two main parts, a DSL unit defining the concepts and relations of *DSLMetaMod*, called *DSLMetaModModels*, and a DSL unit defining a particular processing unit instance, called *DSLMetaModPU*. The final level, level zero, defines a value model, *ProgramDSLMetaMod*, which conforms to *DSLMetaMod*. The processing

unit defined at level one will be applicable on this value model, *ProgramDSLMetaMod*.

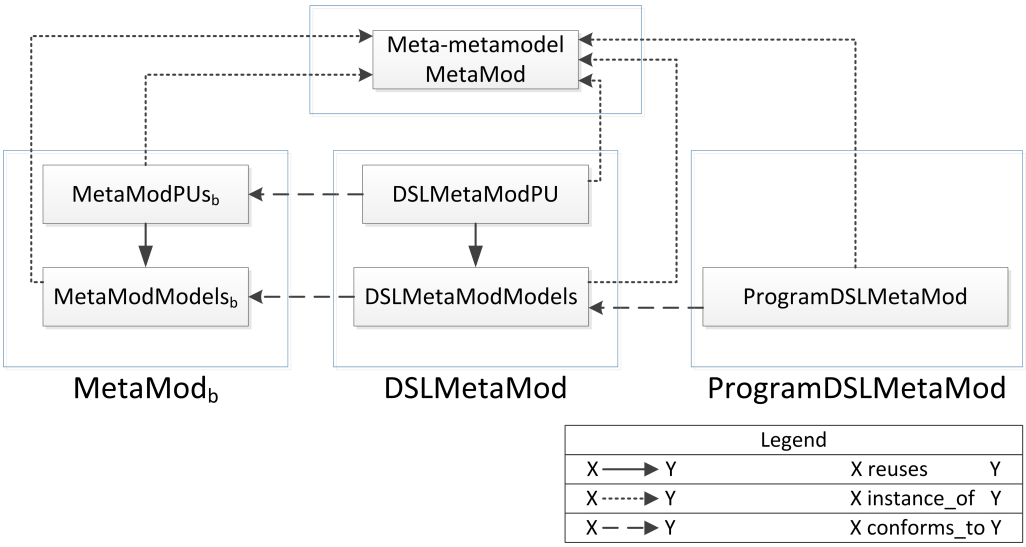


Figure 9.13: The conformance relations among the three levels modeled with MetaMod. The models at all three levels are instances of the meta-metamodel of MetaMod.

We now discuss the characteristics of *DSLMetaMod* and *ProgramDSLMetaMod*.

Level one At level one, we created a value model that conforms to type model *MetaModModels$_b$*, called *DSLMetaModModels*, and that contains only two concepts, *C0* :: *Concept* and *C1* :: *Concept* (see Figure 9.14). We also created a value model (conforming to type model *MetaModPUs$_b$*), *DSLMetaModPU*, that contains two methods, *metLog* and *metCallingMetLog*. Figure 9.15 defines this model. These two models are value models for level two and type models for level zero.

```
DSLMetaMod :: MetaModModels group {
  dslInMetaMod :: Group
  C0 :: Concept
  C1 :: Concept

  dslInMetaMod [----------] C0
               [contains ]
  dslInMetaMod [----------] C1
               [contains ]
}
```

Figure 9.14: Value model *DSLMetaMod* with two concepts.

We applied the processing units defined at level two on *DSLMetaModPU*. The code generated as a result of processing *DSLMetaModPU* is depicted in Figure 9.16. This code can further be used for level zero. That is why we create a small processing unit at level one (see Figure 9.17) that simply calls *metCallingMetLog* on value models at level zero.

```
DSLMetaModPU :: MetaModPUs group {
  reuse MetCallingMetLog unique [ ]
  reuse MetLog unique [ ]

  dslInMetaModPU :: ProcessingUnit

  dslInMetaModPU [----------] dslInMetaMod
                 [forGroup  ]
  dslInMetaModPU [----------] metLog
                 [methods   ]
  dslInMetaModPU [----------] metCallingMetLog
                 [methods   ]
}
```

Figure 9.15: Value model of *DSLMetaModPU* with two methods and associated to group *dslInMetaMod* from Figure 9.14.

```
public class dslInMetaMod {
  public node<NamedGroup> inputGroup;
  public dslInMetaMod(node<NamedGroup> inpGroup) {
    inputGroup = inpGroup;
  }
  public void metCallingMetLog() {
    metLog({ => yieldAll inputGroup.getRepresentativesNamedConcepts().
            where({~it => it.conformsTo.isInstanceOf(RefToNamedConcept) &&
            it.conformsTo : RefToNamedConcept.ref.name.equals("C0"); }); }().first);
  }
  public void metLog(node<NamedConcept> param0) {
    info "Log message!";
  }
}
```

Figure 9.16: The Java class generated from *DSLMetaModPU*. Method *metLog* has a parameter of type concept *C0* (which was generated into a *node⟨NamedConcept⟩*) and a body made of one logging statement. Method *metCallingMetLog* has no parameters and calls method *metLog* with one parameter, the first value concept from the value model at level zero that has type concept *C0*.

```
PU_RunDSLMetaMod_DSLMetaMod
for group DSLMetaMod
aspect RunDSLMetaMod
       reuses << ... >>

operation main(GroupType#DSLMetaMod# inputGroup) returns void {
  dslInMetaMod dMM = new dslInMetaMod(inputGroup);
  dMM.metCallingMetLog();
}
```

Figure 9.17: Processing unit making use of the generated class *dslInMetaMod*.

Level zero At level zero, we created a very simple value model (conforming to *DSLMetaMod*) that contains only one concept, *co* :: *C0* (see Figure 9.18). Moreover, we also applied the processing unit *PU_RunDSLMetaMod_DSLMetaMod* from level one on this value model and we got the logging statement in the console.

```
ProgramDSLMetaMod :: DSLMetaMod group {
  c0 :: C0
}
```

Figure 9.18: A value model of *DSLMetaMod* that contains only one concept.

Some statistics To give you an idea of the scale of the bootstrapping example, here are a few statistics:

- Number of concepts: 42
- Number of relations: 57
- Number of operations: 17
- Number of DSL units: 23

Conclusions This example has illustrated the multilevel nature of MetaMod through three levels of conformance in MetaMod, where level one represents a type model for level zero and a value model for level two. The fact that an element can be both a type and a value is made possible by the multilevel nature of MetaMod. This has enabled us to both create a subset of MetaMod in MetaMod and to test it. The example can be found on the Github repository of MetaMod in Solution *Tests*, model *bootstrapCore*.

With this example, we also demonstrated that bootstrapping can be achieved with MetaMod. Nonetheless, at this stage, through this bootstrapping scheme, we cannot escape out of MPS because we use it to create the models (that are persisted as XML files). Note that this also corresponds to the traditional situation of compiler bootstrapping, where one would not automatically get an editor for the language. The editor has to be written separately.

An interesting exercise would be to create MetaMod with even more features as a DSL in $MetaMod_b$. This would require four levels of conformance, and it would also be a very challenging intellectual exercise (more than a practical exercise).

We conclude with two remarks on the bootstrapping exercise, remarks that are important for the discussion in this chapter. Firstly, the bootstrapping example is a combination of an imperative language (the part on processing units) and of a declarative language (the part on models). Secondly, we want to point the central role that group $MetaModModels_b$ and group $Expression_b$ play (see Figure 9.11). That is because almost all other groups in the figure reuse these two groups, directly, or indirectly.

9.4 Other DSLs

In this section, we review a few DSLs that we already mentioned in different chapters and that were built either for a course or workshop, or for showcasing advantages and disadvantages of a mechanism.

The shapes language One DSL with a declarative flavor, the shapes language, was already covered in Section 4.3. The focus in that section was on highlighting all the steps of building a DSL, from type models, value models, and processing units, to documentation,

fragment abstractions, fragment applications, and DSL extensions. This DSL can be found on the Github repository in Solution *CaseStudies*, model *Courses/OOTICourseShapes*.

This DSL is an excellent demonstration of the feasibility of the ideas we have implemented in MetaMod. The DSL is small enough to easily follow the implementation and big enough to showcase most features of MetaMod. One noticeable benefit of implementing the shapes language with MetaMod, was the ease of creating the second extension, the one where we added a color to the shape (see Section 4.3.3). This would not be possible in MPS, unless one would create a new type of shape, where we add the color property. This, in turn, would require creating a new circle, rectangle and square, that inherit from the new shape.

In models of this DSL we used the fragment abstractions and applications because we had fragments of models that could be nicely generalized into a fragment abstraction (such as the *3ConcentricCircles* in Section 4.3). Applying such an abstraction a few times in a model paid off in terms of time invested in writing the abstraction and conciseness. Moreover, the fragment applications give a more compact and 'domain-specific' look to the models, that do not have a custom syntax otherwise. Another observation we had during this exercise is that the generational concepts are very useful in our case to create, for instance, distinct circles in the *3ConcentricCircles* example. On the other hand, the generated names are not visually appealing or do not have relevant names, but this is the price we have to pay for generational concepts.

In this language, we noticed the central role that the *Shapes* group is playing. Many other groups in this language depend on it, reuse elements from it, and override operations associated to it.

The route language The route DSL is a DSL for describing routes made of forward and turn commands. This DSL is very similar to the shapes DSL in that it also generates a Java frame to draw the route and it has a similar structure to that of the route, except that in this case the central group is represented by the commands. Moreover, it also has similarities to Kaja, because they both describe routes, but the route DSL is much simpler. That is why we do not describe it in the thesis. The DSL can be found on the Github repository in Solution *CaseStudies*.

Graphs, state machines and Petri nets We have also designed various graph DSLs, state machine DSLs and Petri net DSLs. These DSLs were built for the two mechanisms that do not require structurally identical metamodels in the reused DSL and the reusing DSL: the reuse mapping and delegated operations from Section 7 and Section 8, respectively. Their implementation can be found on the Github repository on branches *ReuseMethod1* and *ReuseMethod2*, respectively.

These were very small DSLs, and there were no particular central groups noticeable. These DSLs have highlighted the feasibility of the reuse mapping and that of delegated operations, but also the limitations of the reuse mapping in contrast to delegated operations.

9.5 Testing

We defined around 30 DSLs (each formed of several DSL units) that represent regression tests for MetaMod. These DSLs have as a sole purpose to test that certain features of MetaMod behave as expected. They verify that either constraints (structural or

user-defined) hold on more intricate models, that invalid models report errors, or that the processing of certain models yields the expected results. These tests can be found on the Github repository under Solution *Tests*. Although these tests are not necessarily interesting for analysis, they are important to maintain the quality of the meta-tools of MetaMod.

We have used the 'unique' field of group reuse most intensively in the testing section. Our experience with it was in general positive, but we also had abstract examples where things were a bit confusing. Because of the way the algorithm for equivalence classes works (see Section 4.1.4), it is sometimes hard to understand what elements are from the same equivalence class. Fortunately, in the tool, auto-completion and the possibility to visualize the equivalence classes were of great help. Moreover, the difficulties could have also been caused by the abstract nature of the examples themselves, because the concepts in these examples did not mean anything in particular.

9.6 Discussion

In most DSLs we built, we noticed the formation of one or more core DSL units. In the context of a DSL consisting of a collection of DSL units, a core DSL unit is one that most other DSL units reuse, directly or indirectly. More specifically, the other DSL units use model elements and override operations from these core DSL units. We call a core DSL unit a hot-spot DSL unit. For instance, in the case of the expression language, we placed most of the operations that were later overridden by other DSL units, in the processing units of group *ExpressionsAndTypes*. In the case of bootstrapping with $MetaMod_b$, the hot-spot DSL units were $MetaModels_b$ and $Expression_b$; and in the case of the shapes language, the hot-spot DSL unit was the *Shapes* DSL unit. One bigger DSL that was built without a hot-spot group is the Kaja DSL. As already argued in Section 9.1, this hindered its modularity because of not being able to place the operations in the appropriate DSL units.

There are two other important remarks about the hot-spot DSL unit structure. Firstly, consider DSL D that consists of a collection of DSL units, *Units*, and that has a hot-spot DSL unit, H. The fact that concepts from *Units* are subtyping concepts from H, and that operations from *Units* override operations from H, leads to the DSL units in *Units* to interact indirectly in D. Secondly, the hot-spot DSL unit is usually the main part of the domain (the commands were the most important in Kaja, although this was not visible from the reuse relation among groups; the shapes were the most important in *Shapes*; expressions and types were the most important in *Expression*; and the models were the most important in the bootstrapping example, together with expressions for the operations).

Phenomenons with core artifacts were noticed in other branches of software development as well. If we take a look at the feature-oriented software development community in particular, we see that there are usually hot-spot features in a program [129]. These features interact with many other features. At the same time, work on between-module connections in OOP [143] has shown the same trend, where a few modules are highly connected with most other modules. The hot-spot DSL unit is also in line with the idea of building a "core, elegant language" [46], and then extending it with extra features.

A hot-spot DSL unit should not be confused with a "God DSL unit" though. A "God DSL unit" is a paraphrasing of the God class [116]. A God class is an anti-pattern in OOP design, where a class gets burdened with too many responsibilities (methods, attributes).

Most complexity of the application lies in this class, and it delegates messages to other, less complex classes. The previous sentences could easily be translated to DSLs if we changed the term "class" to the term "DSL unit". The God DSL unit is not applicable to a hot-spot DSL unit, because the hot-spot DSL unit does not know of the other DSL units that extend it, and it does not delegate anything to them. Moreover, the complexity of the DSL does not lie in the hot-spot DSL unit, but it is distributed among all the DSL units.

Another observation we have from implementing these DSLs is that their implementation is an incremental process, especially where modularity is concerned. The process of designing the appropriate groups takes increments. Moreover, the appropriate group organization should be established before attaching processing units to the groups. The domain itself and its organization should drive the processing, and not the other way around.

We also noticed that the fragment abstractions and applications were particularly useful for the value models. They helped both in adding a custom look-and-feel to the value models, but also helped with avoiding duplication in the value models.

Note that the mechanisms for modularity and reuse of DSLs that we built for MetaMod and that we used in this evaluation, are only enablers, and not panaceas. One must use these mechanisms appropriately and judiciously (which involves trade-offs). During the creation of the evaluation DSLs, we have devised a few guidelines, and we have mentioned them in the chapters where we describe the mechanisms of MetaMod.

The last observation we make is that some of the DSLs we developed, were developed at a time that certain features of MetaMod were still missing. That taught us valuable lessons as well. For instance, this is how we discovered that not having multi-operations and overriding hinders modularity and makes extension harder.

9.7 Conclusions

In this section, we reflected on the design of a few DSLs we implemented with MetaMod, and we also highlighted the features of MetaMod that allowed us to achieve modularity and reuse of the DSLs. We looked at small-sized and big-sized DSLs, and at DSLs with different degrees of imperative or declarative constructs. These DSLs demonstrated the feasibility of the ideas underlying MetaMod and also showed cases where the implementation of a DSL is eased with MetaMod (as opposed to other language workbenches, such as MPS). In particular, the reimplementation of DSLs originally built with MPS, such as Kaja and the expression language, helped to a great extent to highlight useful features of MetaMod.

Thus, in this section, we offer an answer to research question RQ_6.

> **RQ_6:** *How can modularity and reuse features of language workbenches be evaluated?*

Chapter 10

Conclusion

In this chapter we conclude the thesis by revisiting the research questions from Chapter 1. Moreover, we discuss the main contributions and we sketch directions for future research.

10.1 Contributions

The work in this thesis was motivated by the visions of model-driven engineering (MDE) and that of language-oriented programming (LOP). Both these methodologies heavily use domain-specific languages (DSLs) in the software development process. Thus, the ease of developing, reusing, and applying DSLs is paramount to accomplishing the visions of MDE and LOP. Ideally, when creating a DSL, the DSL engineer should be concerned only with the complexity of the application domain he is capturing. Tools to create DSLs should not stand in their way. With the advent of language workbenches, and projectional language workbenches in particular, the creation of DSLs has been considerably simplified. Nonetheless, the creation of non-trivial languages, such as mbeddr (see Section 3.1) for instance, requires good support for modularity and reuse. For this reason, we set out to explore modularity and reuse of DSLs from the core, the metamodel. While exploring, we investigated all kinds of mechanisms for the design, implementation, and use of domain-specific languages. These explorations were made under the guidance of our main research question.

> **RQ:** *What are effective ways to achieve modular and reusable definition, implementation, and application of domain-specific languages?*

The main research question was split into six other research questions. Among these, the first research question set the context. This research question was targeted at motivating modularity and reuse for the creation of DSLs. Furthermore, the first research question also looked at requirements of language workbenches for supporting modularity and reuse of DSLs. Thus, we asked

> **RQ_1:** *What are reasons and requirements for modularity and reuse in language workbenches?*

To address this question, we first looked at modularity and reuse in product design, and in software development, in particular (see Chapter 3). DSLs are, in the end, implemented by software; thus, DSL implementation is a special kind of software development, viz. software development focused on implementing DSLs. So, the benefits and challenges of modularity and reuse from software development should apply to DSLs too. Thus, benefits such as interchangeability of modules, ease of updating and maintenance, ease of design and testing, parallel development, improvements in time-to-market, etc., are to be expected. On the other hand, we should be aware of the challenges brought by modularity and reuse in general software development, such as more upfront deliberation and even an increase in complexity if used inappropriately. Moreover, the need for extensibility in general programming languages offers an insight into why we need modularity and reuse of DSLs. Extensibility in programming languages was deemed useful for a variety of reasons, such as security, optimization, static checking, etc. [165]. Besides the advantages we foresee being carried over from general software development to DSLs, one of the biggest reasons for modularity and reuse of DSLs probably comes from the use of DSLs in LOP. Initially, when applying LOP, one might think that one only needs to develop a single DSL while building a particular software application. However, in practice, it turns out that usually one ends up defining multiple DSLs that need to work together. The evolution of DSLs is another case for modularity and reuse. Modular DSLs ease the process of evolution, i.e., it is easier to evolve modular DSLs than monolithic DSLs. Furthermore, due to these advantages, modularity and reuse in the creation of domain-specific languages lead to accomplishing the visions of MDE and LOP.

We return to our observation that a domain-specific language is, in the end, a software application. Thus, the criteria for the extensibility of software (where extensibility relates to modularity and reuse) can be reformulated to criteria for modularity and reuse of domain-specific languages. That is why we took the criteria for the extended expression problem (see Section 3.4), that assess the extensibility power of a programming language, and we translated it to language workbenches. That is, we translated it to criteria that assesses the modularity and reuse powers of a language workbench when it comes to the creation of domain-specific languages.

After we have established the need for modularity and reuse of DSLs, and we have formulated requirements for that, we looked into the specifics of achieving modularity and reuse of DSLs. Specifically, we asked

> **RQ_2:** *How can we organize metamodels of the DSLs such that we facilitate modularity and reuse of DSLs?*

We addressed this research question through the design of the meta-metamodel of MetaMod in Chapter 4 and Chapter 5. Besides modularity and reuse, we also had simplicity as a goal for the meta-metamodel, which is important to mention here because simplicity shaped features we added for modularity and reuse as well (for instance, the simple grouping mechanism). The meta-metamodel consists of three parts: a core part, a group extension, and a fragment abstraction and application extension. The group extension, and the fragment abstraction and application extension are introduced for modularity and reuse reasons. Groups, together with properties such as group reuse, and concept and relation sharing, are used for decomposing metamodels and for reusing metamodels. They lead to incremental definitions and compositions of metamodels. One important difference with many other formalisms is that the groups that reuse elements from elsewhere allow reused concepts to be augmented in the reusing context. Fragment abstractions and applications, on the other hand, are used to factor out structures with

placeholders in the metamodels. This extension combines lambda calculus with modeling elements, which, to the best of our knowledge, is novel. Moreover, there is a feature for reuse in the core part of the meta-metamodel as well, and that is the subtype relationship. The subtype relationship allows reusing the relations of the super-concept in the sub-concept. Although this feature exists in many OOP-like mechanisms, we use it differently. We use concepts and their subtype hierarchy in the type system of the processing units, but we do not create classes from the concepts themselves in the generated code of the processing units. The code is generated per group instead of per concept, which ultimately complements the augmentation of concepts in reusing groups.

Although the central part, the metamodel is only one aspect of a DSL definition; so, we looked at how to organize the other aspects of a DSL as well. Thus, we asked

> **RQ_3:** *How can we organize processing units of the DSLs and the operations in the processing units such that we facilitate modularity and reuse of DSLs?*

We have addressed the third research question in Chapter 5. One of the biggest advantages of introducing modularity and reuse early, already in the metamodel, is that we can take advantage of it in the processing units. Firstly, the metamodels give rise to two hierarchies: the group hierarchy by group reuse and the concept hierarchy by concept subtyping. These hierarchies are exploited by the type system of the processing units. Secondly, processing units are organized around a group and a language aspect. Besides that, operations from processing units of reused groups and sub-aspects are available in processing units of the group and the aspect. These two features of the processing units were influenced directly by the organization of the metamodels. Moreover, reuse of operations is also facilitated by the fact that processing units are stateless. Another feature of the processing units is the multiple dynamic dispatch on the concept types, group types and raw Java types. This allows the adaptation of operations from reused groups in the reusing groups. All the features of the processing units allow the addition of operations to reused groups and the adaptation of operations from reused groups. This is valuable because operations often need to be added and adapted in new contexts, that of the reusing DSLs.

The previous research questions established modularity and reuse features when two DSLs have parts of their metamodels almost identical, and one of the metamodels is reused in the other metamodel. We now asked a different question.

> **RQ_4:** *How can we facilitate reuse of operations despite structural differences among domain-specific languages?*

The fourth research question departs from this direction, and asks how to reuse only the processing units associated to a metamodel in another DSL, when the two DSLs are conceptually similar. To address this question, we created two mechanisms: that of *reuse mapping* and that of *delegated operations*. The first mechanism, based on reuse mappings, was treated in Chapter 7, and it expresses queries from the base DSL unit in terms of queries on the reusing DSL unit. Once this mapping is defined, operations from the processing units of the base DSL can be used in the processing units of the reusing DSL. Although this mechanism can be applied for non-trivial cases, there are a few constraints that need to be satisfied in order to be able to apply the mechanism. These constraints become limiting in some cases as was explained in Chapter 7.

The second mechanism, based on delegated operations, was treated in Chapter 8, and it does not suffer from some of the limitations of the reuse mapping mechanism. For

instance, the reuse of operations defined on a simple state machine in a composite state machine is possible with delegated operations (as opposed to reuse mappings), but the reuse of operations that modify the value model is still not possible. This approach is based on defining a model transformation between the reusing DSL and the base DSL, and defining the operation signature of the reused operations in the reusing DSL. This approach is able to cope with bigger differences between the metamodel of the base DSL and that of the reusing DSL than the mechanism of reuse mapping.

So far, the first four research questions concentrated on the modularity and reuse features of metamodels. Incidentally, these features can be used for value models as well. For this reason, we have briefly looked at modularity and reuse at the value model level as well. Here, we asked

> **RQ_5:** *What modularity and reuse mechanisms can be applied to models, irrespective of the DSL?*

We addressed this research question mostly in Chapter 4 and Chapter 6. The multilevel nature of MetaMod, created by incorporating the conformance relationship in the meta-metamodel (as opposed to having separate definitions of metamodel, models, and their conformance), allows that features we discussed for RQ_2 be applicable to value models as well. That is, the group mechanism, and the fragment abstractions and applications are applicable at the value model level as well. We do not consider the subtype relationship for value models, because this relationship is worth using only if the value model is also a type model. To understand the benefits that grouping, and fragment abstractions and applications bring, we have implemented the Kaja language from the samples projects of MPS in Chapter 6. In Kaja, the modularity and reuse mechanisms that MetaMod offers for value models come in handy. We used groups and group reuse to replace libraries and library import, and we used fragment abstractions and applications to replace procedures and procedure calls. The development effort for Kaja was thus decreased with the help of these modularity and reuse mechanisms for value models.

Now that we have introduced the mechanisms for the design and implementation of DSLs, a natural question would be how to assess these mechanisms. More specifically, we asked

> **RQ_6:** *How can modularity and reuse features of language workbenches be evaluated?*

We addressed this question in Chapter 9 through presenting an analysis of the domain-specific languages implemented in MetaMod with respect to their modularity and reuse characteristics. We have implemented DSLs that range from a small size to a large size, and from a declarative flavor to an imperative flavor (see Chapter 9). In the process of analyzing these DSLs, we have highlighted features of MetaMod that eased the creation of DSLs, and we mentioned the main design decisions we took. We also touched on the evaluation in Chapter 5, where we compared our implementation of the expression language with that from MPS. Although MPS is superior in tool maturity and ease of use, MetaMod does offer some extra features. The extension mechanism in MetaMod is uniform across all language aspects, and MetaMod allows to augment a concept in a reusing DSL, unlike in MPS. These two aspects could lead to a smaller learning curve for the tools and to fewer concepts created only for implementation purposes in the metamodels (see Section 9.2).

10.2 Discussion

The design and implementation of MetaMod was a real exploration. The rapid prototyping that Jetbrains MPS enables once you become familiar with it, empowered us to implement each idea easily. Once implemented, we would play with the idea and decide whether to continue with it or not. It was important to have a short cycle from idea to implementation, because many ideas seem good at first, but in practice fall short of bringing real value. Moreover, MetaMod has taken us through all stages of design and implementation of a language workbench: from the meta-metamodel to the processing units. We spent a considerable amount of time on the design of the meta-metamodel (which defines the structure of metamodels and models), because the metamodel is the central part of the DSL, and the metamodel can shape the types of features one can add to the other DSL aspects.

We think that another big contribution we make is the complete implementation (`https://github.com/farcasia/MetaMod`) of the mechanisms described in this thesis using MPS. The implementation accompanies the description in the thesis, and the complete details of the mechanisms can be viewed there.

We can categorize our explorations of modularity and reuse in the context of DSLs using the three principles mentioned in the introduction: do not use only inheritance-like mechanisms (that is, inheritance and overriding), treat modularity and reuse starting from the meta-metamodel itself (the modeling formalism) and leverage characteristics of DSLs in the features of MetaMod (that is, the central role played by the metamodel and the hierarchies created by model elements). We expand on these in the next three paragraphs.

In the first principle, we say that we also use modularity and reuse mechanisms that are not based on inheritance-like inheritance and overriding mechanisms. We back this up with features for modularity and reuse that we introduced in both the metamodels and the processing units. The metamodel uses a form of modularity with groups, group reuse, and sharing of relations and concepts. The processing units use a form of modularity and reuse where operations from reused groups and reused aspects can be accessed by the current processing unit. Moreover, processing units are made only of operations, and no state. Furthermore, we introduce two reuse mechanisms for reuse of operations only (reuse mappings and delegated operations), without the reuse of the metamodels. These two mechanisms are also different from mechanisms encountered in OOP languages. It is at finer-grained levels that we make use of OOP mechanisms as well. We leverage the hierarchies created by the groups and the subtype relationships in the type system of MetaMod and we allow overriding of multi-operations based on these hierarchies.

In the second principle, we target modularity and reuse from the core of a DSL, the metamodel. These features are captured in the design of the meta-metamodel, and include the groups with their features, the fragment abstraction and application, and the subtype relationships. We did not take an existing formalism and added operations on top of it to cater for the missing modularity. Although there are good reasons for solving modularity outside of the modeling formalism (e.g., legacy models, or existence of other tools), this would not be a deep exploration of modularity.

In the third principle, we also make use of the characteristics of DSLs in the explorations; the ingredients that are constituents of a DSL make the DSL application unique among other software applications. Firstly, we keep the metamodel separate from the processing units; only the processing unit has a dependency to the metamodel, and not the other way around. This is unlike in the MOF world, for instance, where the metamodels have

also operation signatures assigned to them. Because we keep the metamodel and the processing unit separated, there can be various alternatives for processing units associated to a metamodel. Secondly, processing units themselves are organized like the groups in the metamodels. Thirdly, we also leverage the grouping mechanism and the subtype relationships from the metamodels at the processing unit level. Lastly, we leverage the multilevel nature of the meta-metamodel through the use of the same modularity mechanisms for the value models, as in the type models.

In the previous three paragraphs, we have presented the results we have obtained in the explorations with MetaMod; each paragraph discussed the results according to one guiding principle. During these discussions, we also touched on the collection of mechanisms we developed with MetaMod: metamodels with groups sharing concepts and relations; metamodels with lambda abstractions and applications; operations from processing units with multiple dispatch; metamodel hierarchies used in the type system for processing units; stateless processing units; reuse of processing units; reuse mappings; and delegated operations. We have revisited these mechanisms through the lenses of the guiding principles.

We conclude this discussion with the remark that modularity and reuse are difficult topics. No feature is a panacea for all problems related to modularity and reuse. There are plenty of features introduced in programming languages that help with the issue of modularity and reuse, each offering an answer to this issue to a certain degree. On the other hand, none of these features is considered to be "the way". Any feature that one introduces for modularity and reuse needs to be balanced with other qualities, such as ease of use, understandability or performance. We noticed this with the introduction of every new feature in MetaMod.

10.3 Future Work

In this section, we mention some of the main directions for future research that are interesting to pursue with MetaMod.

Having good language mechanisms for modularity and reuse is not by itself a guarantee for the success of these mechanisms. Using the mechanisms well is another discussion. That is why, one needs guidelines on how to use them. Although we have hinted at guidelines in the chapters discussing our mechanisms (e.g., see discussion on number of combinations of smaller DSL units for reusable DSLs in Section 9.2.2), we did not compile an extensive list of guidelines to use the mechanisms. Take, for instance, grouping and group reuse. Group reuse allows the DSL engineer or user to construct arbitrary graph structures (without cycles), but it must be used with care: low fanout and deep hierarchies are hard to understand, as are large fanout and very shallow hierarchies. Somewhere there is a sweet spot, and that brings in cognitive psychology, because it is about making designs understandable and adaptable by humans, with all their cognitive limitations.

Some of the guidelines we develop could also be integrated in the language workbenches themselves. Language workbenches could be equipped with means to advice users (on request, or autonomously) on ways to split their metamodels or processing units, if these get over a certain threshold. This advice could be given in the form of warnings generated in some IDEs, like IntelliJ IDEA [64], for code smells.

One of the most important future work direction is to decrease the generation time of the code created from the processing units and to decrease the run time of this generated code. This relates to the performance problem we mentioned in Section 9.2.2. Although

there are probably many opportunities to optimize the generation and the run times, we have two ideas that we give priority to. In regard to the generation time, the first thing we want to implement is the incremental generation we mentioned in Section 5.2. After a modification in a group or an associated processing unit, only the reusing groups and their corresponding processing units should be regenerated. This alone would reduce the generation time significantly and would allow for a smoother development process. Note that modularity in itself is helpful here; without modularity (so, with monolithic designs), incremental generation is virtually impossible. Thus, a modular design can be a facilitator for improved performance. In regard to the runtime of the generated code, we noticed that the bottleneck can be the querying of the models (see Section 9.2.2). One possibility is to make the navigation queries faster. For that, we could, for instance, persist the metamodel in a graph database like Neo4J [61], and then we could query the models with tools like Cypher [61].

One more interesting direction of future work is to extend the API that navigates and queries the models to provide information on reused models, and fragment abstraction and applications. As commented in Chapter 6, this would enable developers to customize the modularity mechanism for the value models in the processing units of the type models.

Another future work direction is to specialize the processing unit for editor construction. Building editors for the newly created DSLs should be well supported in any language workbench. MetaMod is not a complete (stand-alone) language workbench, because it lacks support for DSL-specific editors, and that is why we say MetaMod is a collection of meta-tools. This would be a first step towards a more audacious future work direction, that of making MetaMod ready for large-scale use. Putting MetaMod in the hands of DSL engineers, would give us more insights into the benefits and challenges of the features of MetaMod.

Something that might also be valuable to pursue is to study the qualities of a DSL in more depth. In Chapter 3, we have presented qualities of DSLs as a paraphrasing of qualities in software systems. In the process, we have noticed that the DSL qualities are more nuanced because of the various language aspects involved in a DSL: the metamodel and the variety of processing units. We think that a more in depth investigation is worthwhile.

10.4 Concluding remarks

We hope that our explorations of features for modularity and reuse have brought the vision of MDE and LOP closer to completion, and that these explorations will be of inspiration for the development of future language workbenches.

We conclude the thesis with the following quote, which is attributed to Larry Wall, known as the creator of Perl [161]. “Computer languages differ not so much in what they make possible, but in what they make easy.” The word ‘easy’ makes the link between a computer language and humans. It is about making things easier for humans, that have cognitive limitations. This further connects to why modularity and reuse are important not only to DSLs (benefiting the DSL users), but also to language workbenches with their meta-languages (benefiting the DSL engineers).

Bibliography

[1] Graphviz - Graph Visualization Software. `http://www.graphviz.org/`. Accessed: 2017-03-13.

[2] *AOM '09: Proceedings of the 13th Workshop on Aspect-oriented Modeling*, New York, NY, USA, 2009. ACM.

[3] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[4] O. Alam, J. Kienzle, and G. Mussbacher. Concern-oriented software design. In *Model-Driven Engineering Languages and Systems*, volume 8107, pages 604–621. Springer, 2013.

[5] Amalio, N. and de Lara, J. and Guerra, E. Fragmenta: A Theory of Fragmentation for MDE. In *Proc. MODELS*, pages 106–115. IEEE, 2015.

[6] Artemis. Crystal - Critical Systems Engineering Acceleration. `http://www.crystal-artemis.eu/`. Accessed: 2017-03-13.

[7] U. Aßmann. *Invasive software composition*. Springer, 2003.

[8] C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[9] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[10] C. Y. Baldwin and K. B. Clark. Managing in an age of modularity. *Managing in the modular age: Architectures, networks, and organizations*, 149:84–93, 2003.

[11] C. Y. Baldwin and K. B. Clark. *Modularity in the Design of Complex Engineering Systems*, pages 175–205. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[12] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. Van Der Storm, and J. Vinju. Modular language implementation in Rascal–experience report. *Science of Computer Programming*, 114:7–19, 2015.

[13] D. Batory. A theory of modularity for automated software development (keynote). In *Companion Proceedings of the 14th International Conference on Modularity*, pages 1–10. ACM, 2015.

[14] H. Berg and B. Moller-Pedersen. Towards non-intrusive composition of executable models. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 1–11. IEEE, 2015.

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

[16] J. Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[17] J. T. Boyland. Remote attribute grammars. *Journal of the ACM (JACM)*, 52(4):627–687, 2005.

[18] G. Bracha and W. Cook. Mixin-based inheritance. *ACM Sigplan Notices*, 25(10):303–311, 1990.

[19] G. Brooch. The Promise, the Limits, the Beauty of Software. Turing Lecture, 2007.

[20] M. Chechik, M. Famelis, R. Salay, and D. Strüber. Perspectives of Model Transformation Reuse. In *International Conference on Integrated Formal Methods - Volume 9681*, pages 28–44. Springer, 2016.

[21] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 115–129. Springer, 2005.

[22] W. Chen. A Theory of Modules Based on Second-Order Logic. In *SLP*, pages 24–33, 1987.

[23] B. H. Cheng, B. Combemale, R. B. France, J.-M. Jézéquel, and B. Rumpe. Globalizing Domain-Specific Languages (Dagstuhl Seminar 14412). 2015.

[24] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[25] T. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise meta-modelling approach. In *Fundamental Approaches to Software Engineering*, pages 159–173. Springer, 2002.

[26] T. Clark, A. Evans, and S. Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46(5):566–577, 2003.

[27] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM Sigplan Notices*, volume 35, pages 130–145. ACM, 2000.

[28] L. L. Constantine. Segmentation and design strategies for modular programming. In *Modular Programming: Proceedings of a National Symposium, Cambridge, MA*, 1968.

[29] N. Cowan. The magical mystery four: How is working memory capacity limited, and why? *Current directions in psychological science*, 19(1):51–57, 2010.

[30] J. Cuadrado, E. Guerra, and J. De Lara. Generic model transformations: write once, reuse everywhere. In *International Conference on Theory and Practice of Model Transformations*, pages 62–77. Springer, 2011.

[31] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[32] J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Model Driven Engineering Languages and Systems*, pages 16–30. Springer, 2010.

[33] J. de Lara and E. Guerra. From types to type requirements: genericity for model-driven engineering. *Software & Systems Modeling*, 12(3):453–474, 2013.

[34] J. de Lara, E. Guerra, and J. Cuadrado. Reusable abstractions for modeling languages. *Information Systems*, 38(8):1128–1149, 2013.

[35] 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 International Conference on Software Language Engineering*, pages 25–36. ACM, 2015.

[36] E. W. Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall Englewood Cliffs, 1976.

[37] DiverSE team from Inria. Kermeta 3 - Executable Meta-Modeling. `http://diverse-project.github.io/k3/`. Accessed: 2017-03-13.

[38] S. Dmitriev. Language Oriented Programming - The Next Programming Paradigm. `http://www.onboard.jetbrains.com/articles/04/10/lop/`. Accessed: 2017-03-10.

[39] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2):1–13, 2004.

[40] D. D'souza and A. Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[41] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.

[42] Eclipse Team. Eclipse. `https://eclipse.org/`. Accessed: 2017-03-10.

[43] T. Ekman and G. Hedin. The JastAdd extensible java compiler. *ACM Sigplan Notices*, 42(10):1–18, 2007.

[44] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, pages 7:1–7:8. ACM, 2012.

[45] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, volume 46, pages 391–406. ACM, 2011.

[46] S. Erdweg and F. Rieger. A framework for extensible languages. In *ACM SIGPLAN Notices*, volume 49, pages 3–12. ACM, 2013.

[47] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[48] J. Estublier, G. Vega, and A. D. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *International Conference on Model Driven Engineering Languages and Systems*, pages 69–83. Springer, 2005.

[49] J.-M. Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer, 2004.

[50] S. I. Feldman. Make - A program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

[51] M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005. Accessed: 2017-03-10.

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

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[54] R. Garud, A. Kumaraswamy, and R. Langlois. *Managing in the modular age: architectures, networks, and organizations*. John Wiley & Sons, 2009.

[55] J. Gershenson, G. Prasad, and Y. Zhang. Product modularity: definitions and benefits. *Journal of Engineering design*, 14(3):295–313, 2003.

[56] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. In *Transactions on Aspect-Oriented Software Development VI*, pages 39–82. Springer, 2009.

[57] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Model-based language engineering with EMFText. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 322–345. Springer, 2011.

[58] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck. *On the Extent and Nature of Software Reuse in Open Source Java Projects*, pages 207–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[59] J. Henriksson, J. Johannes, S. Zschaler, U. Aßmann, et al. Reuseware-Adding Modularity to Your Language of Choice. *Journal of Object Technology*, 6(9):127–146, 2007.

[60] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[61] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[62] B. G. Humm and R. S. Engelschall. Language-Oriented Programming Via DSL Stacking. In *ICSOFT (2)*, pages 279–287. Citeseer, 2010.

[63] Intentional Software Team. Intentional Software, Language Workbench Competition 2011 Entry. `http://www.intentsoft.com/wp-content/uploads/2011/08/lwc11.pdf`. Accessed: 2017-03-10.

[64] D. Jemerov. Implementing refactorings in IntelliJ IDEA. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 13. ACM, 2008.

[65] Jetbrains MPS Team. Meta Programming System - DSL Development Environment. `https://www.jetbrains.com/mps/`. Accessed: 2017-03-10.

[66] Jetbrains MPS Team. MPS's user guide. `https://confluence.jetbrains.com/display/MPSD34/MPS+User's+Guide`. Accessed: 2017-03-10.

[67] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

[68] A. Johnstone, E. Scott, and M. van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23–43, 2014.

[69] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[70] L. C. Kats and E. Visser. *The Spoofax language workbench: rules for declarative specification of languages and IDEs*, volume 45. ACM, 2010.

[71] F. Keller, M. Völter, A. van Hoorn, and K. Birken. Leveraging Palladio for Performance Awareness in the IETS3 Integrated Specification Environment. 2016.

[72] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ a fully configurable multi-user and multi-tool case and came environment. In *International Conference on Advanced Information Systems Engineering*, pages 1–21. Springer, 1996.

[73] S. Kelly and J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[74] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[75] J. Kienzle, W. Al Abed, and J. Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98. ACM, 2009.

[76] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. Deantoni, J. Klein, and B. Rumpe. VCU: The Three Dimensions of Reuse. In *International Conference on Software Reuse*, pages 122–137. Springer, 2016.

[77] D. E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[78] D. E. Knuth and D. Bibby. *The texbook*, volume 3. Addison-Wesley Reading, 1984.

[79] D. Kolovos, R. Paige, and F. Polack. The epsilon object language (EOL). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.

[80] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[81] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 2013.

[82] G. Konat, L. E. de Souza Amorim, E. Visser, and S. Erdweg. Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench. 2016.

[83] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *International Conference on Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.

[84] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, 2010.

[85] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[86] I. Kurtev, J. Bézivin, and M. Akşit. Technological spaces: An initial appraisal. 2002.

[87] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616. ACM, 2006.

[88] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, 14(2):537–572, 2015.

[89] S. Laurence and E. Margolis. Concepts and cognitive science. *Concepts: core readings*, pages 3–81, 1999.

[90] J. Lilius and I. Paltor. Formalising UML state machines for model checking. In *International Conference on the Unified Modeling Language*, pages 430–444. Springer, 1999.

[91] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[92] D. H. Lorenz and B. Rosenan. Cedalion: a language for language oriented programming. In *ACM SIGPLAN Notices*, volume 46, pages 733–752. ACM, 2011.

[93] D. H. Lorenz and B. Rosenan. Code reuse with language oriented programming. In *International Conference on Software Reuse*, pages 167–182. Springer, 2011.

[94] D. H. Lorenz and B. Rosenan. CEDALION's Response to the 2016 Language Workbench Challenge. 2016.

[95] mbeddr Team. The mbeddr.platform. `http://mbeddr.com/platform.html`. Accessed: 2017-03-10.

[96] D. D. McCracken and E. D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

[97] J. D. McGregor. Complexity, it's in the mind of the beholder. *Journal of Object Technology*, 5(1):31–37, 2006.

[98] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, (5):472–482, 1981.

[99] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[100] M. Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451–2464, 2013.

[101] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An interactive environment for programming language development. In *International Conference on Compiler Construction*, pages 1–4. Springer, 2002.

[102] M. Mernik, V. Žumer, M. Lenič, and E. Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices*, 34(6):68–75, 1999.

[103] B. Meyer. *Object-oriented software construction*, volume 2. Prentice-Hall New York, 1988.

[104] B. Meyer. The many faces of inheritance: A taxonomy of taxonomy. *Computer*, 29(5):105–108, 1996.

[105] Microsoft. Overview of Domain-Specific Language Tools. `https://msdn.microsoft.com/en-us/library/bb126327.aspx`. Accessed: 2017-03-13.

[106] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[107] T. J. Misa and P. L. Frana. An interview with Edsger W. Dijkstra. *Communications of the ACM*, 53(8):41–47, 2010.

[108] N. Moha, V. Mahé, O. Barais, and J.-M. Jézéquel. Generic model refactorings. In *International Conference on Model Driven Engineering Languages and Systems*, pages 628–643. Springer, 2009.

[109] D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2):91–105, 1985.

[110] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[111] Object Management Group. Documents associated with Meta Object Facility 2.0, Query/View/Transformation (QVT), v1.2. `http://www.omg.org/spec/QVT/1.2/`. Accessed: 2017-03-13.

[112] Object Management Group. Meta Object Facility (MOF), Version 2.5. `http://www.omg.org/spec/MOF/2.5/PDF`. Accessed: 2017-03-13.

[113] Object Management Group. Object Constraint Language, Version 2.4. `http://www.omg.org/spec/OCL/2.4/PDF/`. Accessed: 2017-03-13.

[114] Object Management Group. UML Infrastructure Specification, Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF`. Accessed: 2017-03-13.

[115] Object Management Group. UML Superstructure Specification, Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`. Accessed: 2017-03-13.

[116] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[117] T. Parr. *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.

[118] R. Pike. Go at Google: Language Design in the Service of Software Engineering. `https://talks.golang.org/2012/splash.article`. Accessed: 2017-03-13.

[119] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open multi-methods for C++. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 123–134. ACM, 2007.

[120] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[121] J. Sanchez Cuadrado, E. Guerra, and J. De Lara. A component model for model transformations. *Software Engineering, IEEE Transactions on*, 40(11):1042–1060, 2014.

[122] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[123] E. Schindler, K. Schindler, F. Tomassetti, and A. M. Şutîi. Language Workbench Challenge 2016: the JetBrains Meta Programming System. 2016.

[124] D. C. Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[125] M. Schöttle, O. Alam, G. Mussbacher, and J. Kienzle. Specification of domain-specific languages based on concern interfaces. In *Proceedings of the 13th workshop on Foundations of aspect-oriented languages*, pages 23–28. ACM, 2014.

[126] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

[127] P. Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation*, pages 420–435. Springer, 2002.

[128] M. Shaw. Writing good software engineering research papers. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 726–736. IEEE, 2003.

[129] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177. IEEE, 2012.

[130] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, pages 398–399, 1995.

[131] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *ACM SIGPLAN Notices*, volume 41, pages 451–464. ACM, 2006.

[132] R. Solmi. Whole Platform. PhD thesis. `http://www.cs.unibo.it/~solmi/papers/Sol05_PhDthesis.pdf`. Accessed: 2017-03-10.

[133] S. Sparks, K. Benner, and C. Faris. Managing object oriented framework reuse. *Computer*, 29(9):52–61, 1996.

[134] J. Steel and J.-M. Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

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

[136] B. Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

[137] A. M. Şutıi. MetaMod: a modeling formalism with modularity at its core. 2015.

[138] A. M. Şutîi, T. Verhoeff, and M. van den Brand. Modular modeling with a computational twist in MetaMod. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 4–7. ACM, 2016.

[139] A. M. Şutîi, T. Verhoeff, and M. van den Brand. Modular multilevel metamodeling with MetaMod. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 212–217. ACM, 2016.

[140] A. M. Şutîi, T. Verhoeff, and M. van den Brand. Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod. *Computer Languages, Systems & Structures*, 2017.

[141] A. M. Şutîi (Farcaşi). Improving modularity in GLL. Master thesis. http://alexandria.tue.nl/extra1/afstversl/wsk-i/farcasi2013.pdf. Accessed: 2017-03-10.

[142] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[143] C. Taube-Schock, R. J. Walker, and I. H. Witten. Can we avoid high coupling? In *European Conference on Object-Oriented Programming*, pages 204–228. Springer, 2011.

[144] P. D. Terry. Compilers and compiler generators: an introduction with C++. 2000.

[145] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110–2126, 2013.

[146] TypeFox. Xtext - Language engineering for everyone. https://eclipse.org/Xtext/. Accessed: 2017-03-13.

[147] E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[148] M. G. van den Brand, A. van Deursen, J. Heering, H. De Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, et al. The ASF+SDF meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, 2001.

[149] A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[150] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.

[151] E. Van Wyk, O. De Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *International Conference on Compiler Construction*, pages 128–142. Springer, 2002.

[152] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *European Conference on Object-Oriented Programming*, pages 575–599. Springer, 2007.

[153] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software and Systems Modeling*, 2(3):187–210, 2003.

[154] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *International Conference on the Unified Modeling Language*, pages 290–304. Springer, 2004.

[155] T. Verhoeff. Personal Communication.

[156] E. Visser. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, pages 299–315. Springer, 2002.

[157] 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.

[158] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software & Systems Modeling*, pages 1–46, 2017.

[159] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Third International Conference on Software Language Engineering (SLE 2010)*, 2010.

[160] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[161] L. Wall, T. Christiansen, and J. Orwant. *Programming perl*. O'Reilly Media, Inc., 2000.

[162] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[163] C. Wende, N. Thieme, and S. Zschaler. A role-based approach towards modular language engineering. In *Software Language Engineering*, pages 254–273. Springer, 2010.

[164] O. Werner. Sapir-Whorf Hypothesis. *Concise Encyclopedia of Philosophy of Language*, pages 76–83, 1997.

[165] G. V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, 2004.

[166] P. H. Winston and B. K. Horn. Lisp. 1986.

[167] E. Yourdon and L. L. Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.

[168] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proceedings of Workshop on Foundations of Object-Oriented Languages, FOOL*. 2005.

Appendix A

Code generated from the shapes example

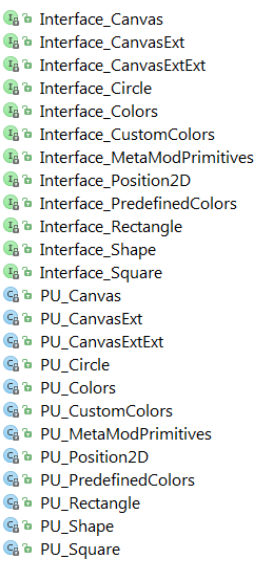


Figure A.1: The Java interfaces and classes generated from the metamodels in the example DSL with shapes presented in Section 4.3.

```
public interface Interface_Shape extends Interface_Position2D {
  void multi_drawShape_GroupTypeShape_ConceptTypeShape_Graphics(
      node<NamedGroup> inputGroup, node<NamedConcept> shape, Graphics graphics);
}
```

Figure A.2: The interface generated for group *Shape*, *Interface_Shape*. This interface extends interface *Interface_Position2D*, because group *Shape* directly reuses group *Position2D* (see Figure 4.20). Interface *Interface_Shape* also contains the signature of a method originating from a multi-operation for drawing shapes, because multi-operation *drawShape* was defined in a processing unit for group *Shape* (see Figure 4.24).

```
public class PU_Square extends CodeGen implements Interface_Square {
  public PU_Square(node<NamedGroup> group, node<NamedGroup> inputGroup, MetamodelClass callingContext) {
    super(group, inputGroup, callingContext);
  }

  @Override
  public boolean checkAllConstraints() {
    if (!innerGroups["MetaModPrimitives"].checkThisConstraints()) { return false; }
    if (!innerGroups["Position2D"].checkThisConstraints()) { return false; }
    if (!innerGroups["Shape"].checkThisConstraints()) { return false; }
    if (!innerGroups["Rectangle"].checkThisConstraints()) { return false; }
    if (!innerGroups["Square"].checkThisConstraints()) { return false; }

    return true;
  }

  @Override
  public boolean checkThisConstraints() {
    if (!LengthAndWitdhEqual(inputGroup)) { return false; }

    return true;
  }

  protected void initResolveMaps() {...}

  public void multi_drawShape_GroupTypeShape_ConceptTypeShape_Graphics(
      node<NamedGroup> inputGroup, node<NamedConcept> shape, Graphics graphics) {
    if (Utils.allTrue(inputGroup.metamodel.
        getAllSuperConceptsPlusSelf(inputGroup.metamodel.getRepresentativesNamedConcepts().
          where({~it => UtilityMethods.areTheSameConcept(shape.conformsTo : RefToNamedConcept.ref, it,
        inputGroup.metamodel); }).first, inputGroup.metamodel).
        where({~it => UtilityMethods.areTheSameConceptWithString("Rectangle.Rectangle", it, inputGroup.metamodel); }).
        size != 0, graphics instanceof Graphics)) {
      {
        innerGroups["Rectangle"] as PU_Rectangle.
            drawShape_GroupTypeRectangle_ConceptTypeRectangle_Graphics(inputGroup, shape, graphics as Graphics);
        return;
      }
    }

    innerGroups["Shape"] as PU_Shape.drawShape_GroupTypeShape_ConceptTypeShape_Graphics(inputGroup, shape, graphics);

  }
  public boolean LengthAndWitdhEqual(node<NamedGroup> inputGroup) {...}
}
```

Figure A.3: The processing unit generated for group *Square*, *PU_Square*. We folded some methods for readability. This processing unit shows that the method for checking constraints, *checkAllConstraints* does so in the order of the reused groups (from the inner-most group to the outer-most group; see Figure 4.20). The processing unit also shows that a method corresponding to a multi-operation, in this case the one for drawing shapes, checks the type of the shape to choose the overridden method to call. Because *Square* only knows about *Rectangle* and *Shape*, it first checks whether the shape is a rectangle, and, otherwise, it calls the generic method for *Shape*.

```
public class PU_CanvasExtExt extends CodeGen implements Interface_CanvasExtExt {
  public PU_CanvasExtExt(node<NamedGroup> group, node<NamedGroup> inputGroup, MetamodelClass callingContext) {...}

  @Override
  public boolean checkAllConstraints() {...}

  @Override
  public boolean checkThisConstraints() {...}

  protected void initResolveMaps() {...}

  public Color multi_getColor_GroupTypeColors_ConceptTypeColor(node<NamedGroup> inputGroup, node<NamedConcept> color)
      {...}
  public void multi_drawShape_GroupTypeShape_ConceptTypeShape_Graphics(
      node<NamedGroup> inputGroup, node<NamedConcept> shape, Graphics graphics) {...}
  public void setColor(node<NamedGroup> inputGroup, node<NamedConcept> shape, Graphics graphics) {...}
  public JPanel createPanelWithColor(final node<NamedGroup> inputGroup) {...}
  public void main(node<NamedGroup> inputGroup) {

    if (!checkAllConstraints()) {
      error "Some constraints do not hold on the model!";
      return;
    }
    JFrame frame = new JFrame(inputGroup.getRepresentativesNamedConcepts().
          where({~it => it.conformsTo.isInstanceOf(RefToNamedConcept) &&
          Utils.isConceptOrSuperConcept(it.conformsTo : RefToNamedConcept.ref, "Canvas.Canvas", inputGroup.
          metamodel); }).toList.first.name);
    final JPanel panel = innerGroups["CanvasExtExt"] as PU_CanvasExtExt.createPanelWithColor(inputGroup);
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Figure A.4: The processing unit generated for group *CanvasExtExt*, *PU_CanvasExtExt*. We folded some methods for readability. All the operations that were defined for this group, such as *createPanelWithColor* (see Section 4.3.3), appear in this class as methods. Moreover, one can see that the first instruction in the main method is to check the constraints.

Index

For indexing, we use the same conventions as Donald Knuth in *The TexBook* [78]. We underline the page numbers that contain a main description or the definition of the indexed term. We italicize the page numbers that contain the most relevant examples of the indexed term.

Summary

Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod

The history of software engineering is one of continuously raising the abstraction level in programming languages. The abstraction level has been raising towards using concepts from the problem space into the programming languages themselves. The ideal situation would be to have programming languages and associated interactive development environments that allow a domain expert to create software applications by instructing the computer using concepts from her own domain. This situation would also require a different software development process, with at least two main role players: language engineers that develop and maintain the domain-specific languages (DSLs), and domain experts that use these domain-specific languages by creating models in them.

This is advocated by new methodologies, such as model-driven development (MDD) and language-oriented programming (LOP). Although MDD and LOP have different focuses (MDD focuses on the models, and LOP focuses on the programming languages), they both use domain-specific languages to achieve their visions. A big leap in easing the creation of DSLs occurred with the introduction of language workbenches; nonetheless, developing non-trivial DSLs is still a daunting task. As already hinted by the discussion in this paragraph, we have concentrated on the part of easing the development of DSLs in this thesis. We have done so by employing modularity and reuse in the creation of domain-specific languages. We have taken the path of modularity and reuse because DSLs, and especially DSLs from the same domain, have common fragments, or DSL units. One example is the use of arithmetic expressions, which is common among many DSLs. This DSL could live in a separate DSL unit for arithmetic expressions. Then, language engineers should be able to seamlessly integrate the arithmetic expressions DSL unit into their own DSL.

There are two main parts to the development of a DSL, the metamodel and the associated processing units. We consider the metamodel to be the core of a DSL because this is where the main concepts and the relations among them are made explicit; moreover, the processing units (interpreters, code generators, editors, etc.) are querying and navigating this metamodel. Getting the metamodel right is essential for both the expressivity of the DSL and the ease of developing the processing units. Thus, much of the development of a DSL relies on the metamodel.

This thesis contributes to the community of DSL development with mechanisms and meta-tools focused on modularity and reuse in the creation of domain-specific languages. We have approached modularity and reuse in the creation of domain-specific languages

starting from the core, the metamodels, and finishing with the processing units. These mechanisms are also accompanied by a prototype meta-tool, MetaMod.

First, we investigated reasons and requirements for modularity and reuse in the creation of domain-specific languages. To this end, we went over advantages and disadvantages of modularity and reuse in other fields, and especially in software engineering. Advantages such as increased productivity and ease of updating, or disadvantages such as more upfront deliberation, are to be expected when carrying over modularity and reuse in the creation of DSLs. Besides these, LOP itself offers a reason for modularity, because in LOP, one needs to combine different DSLs in the creation of a software application. As for requirements, given that DSLs are, in the end, software applications, we have paraphrased criteria for the extensibility of software to criteria for modularity and reuse of DSLs, with slight modifications.

Second, we looked at how to organize metamodels of the DSLs to facilitate modularity and reuse of DSLs. We did that through the design of a new meta-metamodel for MetaMod. This meta-metamodel introduced the following elements for modularity and reuse of DSLs: groups with group reuse, and concept and relation sharing, that allow decomposing and reusing metamodels; fragment abstractions and applications, that allow factoring out common metamodel structures with placeholders; and a subtype relationship that allows reuse of relations. This mix of elements allows the creation of modular and reusable metamodels.

Third, we addressed how to organize processing units of the DSLs and the operations in the processing units to facilitate modularity and reuse of DSLs. For that, we took advantage of the organization in the metamodels by leveraging two hierarchies from the metamodel (the subtype hierarchy and the group hierarchy) in the type system of the processing units. Moreover, we organized the processing units around groups and aspects. What further facilitated modularity and reuse, was the reuse of operations in agreement with the reuse of groups and aspects, and the introduction of multiple dynamic dispatch on the concept types, group types and raw Java types. All these features allow the reuse and adaptation of operations from reused DSL units into reusing DSL units.

Fourth, we investigated the reuse of operations in spite of structural differences among domain-specific languages. That is because, although logically similar, the metamodels of two DSLs can be significantly different; nonetheless, it is of great value to allow the reuse of operations defined for one DSL in the other DSL. For this, we created two mechanisms, the mechanism of reuse mappings and that of delegated operations. The mechanism of reuse mappings is based on expressing queries from the base DSL in terms of queries in the reusing DSL, so that operations from the base DSL can be reused in the reusing DSL. On the other hand, the mechanism of delegated operations is based on creating a model transformation from the metamodel of the reusing DSL to the metamodel of the base DSL, and defining the signature of the operations from the base DSL in the reusing DSL. The latter mechanism is an improvement over the former because it imposes less constraints on the form of the metamodels of the base DSL and reusing DSL.

Fifth, we addressed mechanisms that can be applied to models, irrespective of the DSL. At this point, we used the multilevel nature of MPS, that allows to use mechanisms developed for the metamodel in the models as well. That is, the groups (with group reuse, and relation and concept sharing), and the fragment abstraction and application can be used in the models as well. These mechanisms are available for any model, irrespective of the DSL to which they conform.

Last, we have evaluated our approach, starting with the development of the meta-tools of MetaMod. We have then created DSLs ranging from small sizes to big sizes, and from

imperative-flavoured to declarative-flavoured. Most notably, we have re-implemented a considerable part of a non-trivial expression language built in Jetbrains MPS. This gave us confidence that the meta-tools and the ideas behind them are powerful enough to tackle real-life problems. Moreover, it has also allowed us to make a more direct comparison with a seasoned language workbench. As a result of the comparison, we have discovered that MetaMod leads to more conceptually cohesive DSLs because it avoids the introduction of some implementation-oriented concepts. Moreover, the modularity of MetaMod allowed us to create smaller-sized DSL units, that are reusable in separation, but that are part of the overall expression language DSL.

We think that the ideas that we have introduced in this thesis get us a step closer to the accomplishment of the visions of MDE and LOP, and that they will speed up the development of domain-specific languages.

Curriculum Vitae

Personal Information

Name: Ana Maria Şutîi
Date of birth: September 11, 1987
Place of birth: Bacău, Romania

Education

MSc. Computer Science and Engineering *2011–2013*
Eindhoven University of Technology
Eindhoven, the Netherlands

BSc. Computer Science and Engineering *2007–2011*
Politehnica University from Bucharest
Bucharest, Romania

Professional Experience

PhD candidate *2013–2017*
Eindhoven University of Technology
Eindhoven, the Netherlands

Software Engineer Intern *2012*
Google
Munich, Germany

IPA Dissertation Series

Titles in the IPA Dissertation Series since 2014

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes.* Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart*

Cards. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty

of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects.* Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence.* Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare.* Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance.* Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices.* Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems.* Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs.* Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement.* Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security.* Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning.* Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections.* Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors.* Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).* Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09