Applications of Graph Transformation in Tools for Domain-Specific Modeling Languages

Torbjörn Lundkvist

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3–5 A, 20520 Turku, Finland

2011

Supervisor

Professor Iván Porres
Department of Information Technologies
Åbo Akademi University
Joukahasenkatu 3–5 A, 20520 Turku
Finland

Reviewers

Associate Professor Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics
Magyar Tudósok körútja 2
1117 Budapest
Hungary

Professor Jon Whittle
Department of Computing
Lancaster University
Bailrigg
Lancaster, Lancs LA1 4YW
United Kingdom

Opponent

Associate Professor Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics
Magyar Tudósok körútja 2
1117 Budapest
Hungary

ISBN 978-952-12-2601-4
Painosalama Oy, Turku, Finland, 2011

Abstract

The use of domain-specific languages (DSLs) has been proposed as an approach to cost-effectively develop families of software systems in a restricted application domain. Domain-specific languages in combination with the accumulated knowledge and experience of previous implementations, can in turn be used to generate new applications with unique sets of requirements. For this reason, DSLs are considered to be an important approach for software reuse.

However, the toolset supporting a particular domain-specific language is also domain-specific and is per definition not reusable. Therefore, creating and maintaining a DSL requires additional resources that could be even larger than the savings associated with using them.

As a solution, different tool frameworks have been proposed to simplify and reduce the cost of developments of DSLs. Developers of tool support for DSLs need to instantiate, customize or configure the framework for a particular DSL. There are different approaches for this. An approach is to use an application programming interface (API) and to extend the basic framework using an imperative programming language. An example of a tools which is based on this approach is Eclipse GEF. Another approach is to configure the framework using declarative languages that are independent of the underlying framework implementation. We believe this second approach can bring important benefits as this brings focus to specifying *what* should the tool be like instead of writing a program specifying *how* the tool achieves this functionality.

In this thesis we explore this second approach. We use graph transformation as the basic approach to customize a domain-specific modeling (DSM) tool framework. The contributions of this thesis includes a comparison of different approaches for defining, representing and interchanging software modeling languages and models and a tool architecture for an open domain-specific modeling framework that efficiently integrates several model transformation components and visual editors. We also present several specific algorithms and tool components for DSM framework. These include an approach for graph query based on region operators and the *star operator* and an approach for reconciling models and diagrams after executing model transformation programs.

We exemplify our approach with two case studies MICAS and EFCO. In these studies we show how our experimental modeling tool framework has been used to define tool environments for domain-specific languages.

Sammanfattning

Domänspecifika språk har föreslagits som en metod för att kostnadseffektivt utveckla familjer av mjukvarusystem inom ett begränsat applikationsområde. Domänspecifika språk i kombination med tidigare samlad erfarenhet och kunskap om applikationsområdet kan i sin tur användas för att generera nya applikationer med unika specifikationer. Av denna anledning anses domänspecifika språk vara en viktig metod för att återanvända mjukvara.

Även verktygsstödet som utvecklats för ett visst applikationsområde är per definition domänspecifikt och kan inte återanvändas inom andra applikationsområden. Följaktligen kräver utveckling och underhåll av ett nytt domänspecifikt språk tilläggsresurser som kan vara större än de inbesparningar som användningen av detta kan inbringa.

För att förenkla utveclingsarbetet för nya domänspecifika språk och följaktligen minska kostnaderna för detta, har ramverk för ändamålet framtagits. Fortfarande bör utvecklare som handhar utvecklingen av verktygsstöd för domänspecifika språk skräddarsy det valda ramverket för det domänspecifika språket. Det finns olika metoder för detta. En metod är att använda ramverkets programmeringsgränssnitt (API) och således utöka ramverket med program skrivna i ett imperativt programmeringsspråk. Exempel på detta är Eclipse GEF. En annan metod är att skräddarsy ramverket med hjälp av deklarativa språk som är oberoende av ramverkets implementation. Vi anser den andra metoden vara fördelaktigare eftersom utvecklaren då kan fokusera mera på *vad* verktyget skall utföra, än att skriva program som beskriver *hur* verktyget kan uppnå en viss funktionalitet.

I denna avhandling studerar vi den andra metoden. Vi använder graftransformationer som utgångspunkt för att skräddarsy ett ramverk för domänspecifik modellering. Denna avhandling bidrar med en jämförelse över olika metoder att definiera, representera och utbyta modeller och modelleringspråk och en arkitektur för ett öppet ramverk för domänspecifik modellering som effektivt integrerar flera modelltransformationskomponenter och visuella editorer. Vi presenterar även flera specifika algoritmer och verktygskomponenter för bruk inom ramverk för domänspecifik modellering. I dessa ingår en metod för förfrågning av grafer baserade på operatörer som verkar på regioner samt en metod för att uppdatera diagram efter att modelltransformationsprogram exekverats.

Vi exemplifierar vår metod med två fallstudier, MICAS och EFCO. I dessa studier visar vi hur vårt experimentella ramverk för domänspecifik modellering använts för utvecklingen av konkreta verktygsmiljöer för domänspecifika språk.

Acknowledgements

It is a pleasure for me to take this opportunity to express my deepest gratitude to those who made this thesis possible.

First of all, I would like to thank my supervisor Professor Iván Porres for having confidence in me and for supporting and guiding me in this work. It has been a privilege to me to have had the opportunity to work closely with Iván in many interesting projects.

I also wish to thank Associate Professor Dániel Varró and Professor Jon Whittle for reviewing this thesis and for providing me with useful comments. I would like to thank Dániel for kindly accepting the task of being my opponent at the public defense.

I would especially like to thank my co-authors Marcus Alanen, Johan Lilius, Johan Lindqvist, Dragoş Truşcan, Tomas Lillqvist, Johan Ersfolk, Asim Zaka, Glenn Sveholm, Ian Oliver and Kim Sandström. I have had many interesting discussions with Marcus that have influenced this work, and I am grateful for his help. I would like to thank Johan Lilius and Dragoş for interesting discussions that have broadened my perspective on modeling embedded systems. Tomas, Asim and Glenn for working closely with me in the MICAS project. I had very much fun working with Tomas. Johan Ersfolk for explaining me the CoFluent Studio. Johan Lindqvist for his special contributions in the CQuery project. I am also grateful for the help in implementing the research ideas from undergraduate student working on the Coral projects.

I would like to thank the department and the Graduate School on Software Systems and Engineering (SoSE) for providing me with funding for the research work and conference travel. A special thanks go to Kai Koskimies and Maarit Harsu for organizing the superb seminars in Lapland. At these semainars and at conference trips I have had the opportunity to meet many fellow PhD students and researchers to share ideas with. The discussions with, and the feedback from Jim Coplien have been very interesting and useful to me. I also appreciate the long discussions on graph transformation with Reiko Heckel.

I also thank Bernhard Schätz for the opportunity to visit TU München and to work in his research group. A special thanks go to Florian, Daniel, Marco, Markus, Silke and Marina.

Thanks also go to the colleagues, administrative personnel and technical personnel at the Department of Information Technologies for providing an enjoyable and fun place to work. In particular I would like to thank Johannes, Mats, Ali, Miki, Marta, Jeanette, Irum, Adnan, Espen, Fredrik D., Fredrik A., Pontus, Leonidas, Linda, Kim, Larissa, Viorel, Luigia, Patrik, Ye, Jerker, Andreas, Juuso, Jani, Sébastien, Marina, Anna, Karl, Hannu, Åke, Luka, Andrea and Piia for many interesting discussions in almost any topic. A special thanks go to Ralph-Johan Back, Barbro Back, Patrik Sibelius, Kaisa Sere, Ion Petre and Jan Westerholm for their encouragement throughout this work. Thanks to Christel for keeping the department running, Britt-Marie for invaluable advice and to Tiina for sorting out the tricky travel and funding related questions I have had.

I am honored and grateful for the generous scholarhips received from the Finnish Foundation for Technology Promotion, the Nokia Foundation, Hans Bang Stiftelsen, Ulla Tuomisen Säätiö and Harry Elvings Legat.

I would like to thank all my friends for all the support and encouragement. Finally, I want to thank my wife Desirée, my daughter Mathilda and all my family and my family-in-law for all their love and support.

Turku, May 2011

Torbjörn Lundkvist

Contents

Part I

Research Overview

Chapter 1

Introduction

1.1 Research Objectives

Developing large software systems is a complex process that involves the use and integration of many different artifacts. A rigorous software engineering process can, in addition to source code, produce artifacts such as requirement documents, design specifications, test cases, build configurations, documentation et cetera, which are linked in complex relationships. Managing these artifacts and their relationships in large systems is a difficult task and requires tool support. With increasing processing capabilities at decreasing hardware costs, software has a tendency to also become more complex. To manage the increasing complexity of software, techniques such as software composition and abstractions are necessary. By raising the level of abstraction of software, the perceived complexity can be reduced, and consequently help understanding and reasoning about software.

Software modeling is about creating abstractions of software at different levels. Hence, a software model is an abstraction of software. The level of abstraction is a trade off of the level of detail.

The abstractions used in software models are defined by modeling languages, which aim to present a meaningful, particular aspect of software. For example, a software system can be seen from a structural perspective, describing the relationships of object classes of the software. A concrete example of this is the UML class diagram [74], essentially a refinement of the ideas by Rumbaugh [87]. Software can also be seen from a behavioral perspective, describing the interaction of the different components of software. A notable example is the state machine, which models a software system as an abstract machine with states and transitions [39]. Formalisms such as these are currently part of the UML standard [74], which has reached *de facto* status in the software industry. UML is an example of a *general-purpose modeling language*, which can be applied to many types of applications.

Another approach of software modeling is to view the software from the point-of-view of the software problem domain. For a single application implementing its own unique set of requirements, the problem domain presents a high-level description of what is relevant for modeling a solution for that particular problem. However, if the problem domain is slightly expanded to include a set of similar or a *family* of software problems, abstractions of a larger *application domain* can be created, where a subset of the available concepts are combined to form a set of application specific requirements. Such languages are called *domain-specific modeling languages*. Domain-specific modeling languages (DSMLs) can be used to model applications in a well-known, restricted application domain, in which there may exist many implementations. DSMLs are often seen as domain-specific programming or specification languages [93], based on graphs and often having a visual syntax. One of the main characteristics of domain-specific (modeling) languages is the idea of relating the different concepts used for defining requirements to an implementation and consequently to the idea of generating entire applications [21, 26] based on models [47, 48, 70]. Often, this is listed as a main advantage of DSMLs, as this can reduce the cost and effort of implementing new applications.

However, although there exists many successful domain-specific languages, implementing them for a new or existing application domain can be challenging, as it implies the development of new, custom application development tools. To develop and maintain such tools requires additional resources, which can in turn imply higher costs than the projected savings of using a development process based on DSMLs. To overcome these additional costs, different domain-specific modeling (DSM) frameworks have been proposed. However, developers of tool support for DSMLs need to instantiate, customize or configure the framework for a particular DSML. There are different approaches for this. Perhaps the most straightforward approach is to use a framework application programming interface (API) and to extend the basic framework using a standard programming language. Another approach is to extend the framework using declarative languages that are independent of the underlying framework implementation. This can have important benefits as this brings focus to specifying *what* should the tool be like instead of writing a program specifying *how* the tool achieves this functionality.

In this thesis, we explore the second approach. We use graph grammars and graph transformation in the development of a framework for the development of tools for domain-specific, visual modeling languages for software and system engineering. We have used these techniques to explore the definition of DSML tool environments based on declarative specifications expressed as graphs and graph transformations. We do this in the context of industry standards from the Object Management Group (OMG).

Conceptually, a DSM framework consists of many tool components with different responsibilities. In this thesis, we also explore and present approaches for the efficient organization of such tool components in a DSM framework, while maintaining a low coupling between tool components. We study this especially in the context of graph transformation.

We continue in the next section by presenting the concrete contributions of this thesis.

1.2 Contributions of this Thesis

In this thesis several contributions related to the use of graph transformation in the context of framework development for tools for domain-specific modeling languages (DSMLs) are presented. These contributions are presented in detail in Publications I–VII, which are included in Part II of this thesis. A summary of the contributions is presented below:

Comparison of modeling frameworks. A modeling framework is at the core of modeling tools and decides how information is defined, represented, manipulated and interchanged. Currently, there are several modeling frameworks which can be used to describe information especially in the context of modeling tools and these frameworks share many commonalities. The focus of this contribution is to present and compare several modeling frameworks and discuss the main differences of each of these approaches and to discuss benefits and drawbacks of these approaches. This contribution is presented in Publication I.

An editor architecture for a domain-specific modeling framework. A DSM framework is essential for cost-efficient implementation of a DSML application. A DSM framework often consists of several individual components which are configured to obtain a tool environment which supports the development of applications in a restricted application domain. We present a tool architecture for an open DSM framework that efficiently integrates several model transformation components and visual editors, realized by maintaining a strong separation of the abstract and concrete syntax and by the integration of change propagation components in a tool environment. We present an experimental DSM framework which realizes this approach. This contribution is discussed in detail in Publication IV.

The contributions of this research work also consists of several specific algorithms and tool components for a DSM framework:

Graph query based on region operators. Graph query is used to find parts of graphs that fulfill some given constraints. Graph query is an important operation in software modeling tools, and is fundamental in graph transformation based on algebraic approaches. We present in this thesis a query language based on declarative patterns consisting of regions denoting the scope of a matching operator. In the context of this query language we present a matching operator called the *star operator*, which can be used to match hierarchical and recursive structures in graphs. We also present a matching algorithm, which extends existing approaches for graph matching using subgraph isomorphism, for matching patterns with regions against a target graph. We present this contribution in Publication II and discuss a graph transformation engine based on the double-pushout approach with support for the star operator in Section 3.

Diagram reconciliation. In this thesis we propose an approach for maintaining consistency between the abstract and concrete syntax after executing model transformations. This approach is based on a mapping language between the abstract syntax and the Diagram Interchange (DI) standard, called the Diagram Interchange Mapping Language (DIML). Based on this mapping language, we present an approach to decouple the changes made in models from the related updates in diagrams. We also present algorithms for creating new diagrams where previous do not exist and updating existing diagrams when the underlying abstract model has changed, and present a proof-of-concept implementation in the context of a modeling tool. We also show that this approach is beneficial, as it allows model transformation components to deal with the abstract syntax alone while diagrams are still maintained. We present this contribution in Publication III.

1.3 Validation of the Research Work

We have validated our research work in the context of an experimental tool called the Coral Modeling Framework. The Coral Modeling Framework is an experimental DSM framework developed as a demonstrator of research ideas in the context of various research projects at the Department of Information Technologies at Åbo Akademi University, including these presented in the context of this thesis.

This framework provides a proof-of-concept implementation of the editor architecture presented as a contribution of this thesis in Publication IV. This editor architecture consists of many different editor components, where update responsibilities are delegated to a set of dedicated change propagation components. Among these change propagation components is the diagram reconciliation component based on DIML presented in Publication III. We have used this component to further support the integration of tool components in a tool environment, based on the abstract syntax alone. We have validated in practise that by using this approach we can reduce the coupling between editor components by decoupling model transformations and diagram updates.

We have implemented a graph query engine called CQuery based on the query language with the star operator and the matching algorithm presented in Publication II. This work is a further development of the work by Lillqvist [56]. This query engine consists of a graphical editor allowing the definition of queries using the concrete syntax of the target modeling language. This query engine is an integral part of the Coral Modeling Framework and has successfully been used to define queries in many modeling languages like UML 1.4 and 2.0 and in many domain-specific languages. The query engine has also been used as a part of an experimental transformation engine based on the double-pushout approach. We have evaluated the use of the star operator in rule-based model transformation in a component called DPOTrans, especially in the context of the Coral diagram editor, which uses double-pushout transformation rules to define editor actions for all modeling languages. CQuery has also been evaluated in the context of a constraint evaluation engine initially developed by Lillqvist [56] and further adapted to support the star operator. There is also a model-to-text transformation engine based on CQuery developed by Nyman [67].

Using the Coral Modeling Framework, we have implemented diagram editors for UML 1.4 and 2.0, implementing most of the diagrams described in the standards. We have implemented editors for many of the domain-specific languages used for developing editors, components and tool environments based on Coral. For example, we have implemented editors for the DIML and CQuery languages presented in this thesis. We have also validated the approaches presented in this thesis in the context of the domain-specific tool environments MICAS (Publication V and VI) and EFCO (Publication VII).

1.4 List of Original Publications

This thesis is based on the following publications numbered I to VII. For each publication the contribution of the author is stated. In the European research communities in Computer Science, authors are listed alphabetically and thus no distinction is made between the first author and other authors. It must be noted that the authors are here listed in alphabetical order with the exception of Publication VI.

I Marcus Alanen, Torbjörn Lundkvist and Ivan Porres. Comparison of Modeling Frameworks for Software Engineering. Nordic Journal of Computing, vol: 12, num: 4, page(s): 321–342, Winter, 2005.
Author's contribution: The author participated in the preparation of this paper and specially contributed in Section 2 of the paper.

II Johan Lindqvist, Torbjörn Lundkvist and Ivan Porres. A Query Language with the Star Operator, In proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), volume 6(2007) of Electronic Communications of the EASST, Braga, Portugal, March 2007. EASST
Author's contribution: The author developed the original idea of the paper and lead the development of the proof-of-concept implementation.

III Marcus Alanen, Torbjörn Lundkvist and Ivan Porres. Creating and Reconciling Diagrams After Executing Model Transformations. Science of Computer Programming, Elsevier, 68(3), 2008.

This journal article is based on the following conference publications:

- Marcus Alanen, Torbjörn Lundkvist and Ivan Porres. Reconciling Diagrams After Executing Model Transformations. In Proceedings of the

21st Annual ACM Symposium on Applied Computing, Track on Model Transformation, Dijon, France, April, 2006.

- Marcus Alanen, Torbjörn Lundkvist and Ivan Porres. A Mapping Language from Models to DI Diagrams. In Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, 2006.

Author's contribution: The original idea of these papers was jointly developed by Alanen and the author. The author also developed the diagram reconciliation algorithms and the proof-of-concept implementation.

IV Torbjörn Lundkvist and Ivan Porres, Coordination of Model Transformation Engines and Visual Editors. TUCS Technical Report number 974, April 2010.
Author's contribution: The idea of this paper was developed jointly by the author and Prof. Porres.

V Johan Lilius, Tomas Lillqvist, Torbjörn Lundkvist, Ian Oliver, Ivan Porres, Kim Sandström, Glenn Sveholm and Asim Pervez Zaka. An Architecture Exploration Environment for System on Chip Design, Nordic Journal of Computing, vol: 12, num: 4, page(s): 361–378, Winter, 2005
Author's contribution: The author contributed with ideas related to the definition of the domain-specific language, transformations and developed tool support for MICAS. The author also developed parts of the Coral Modeling Framework in parallel.

VI Dragos Truscan, Torbjörn Lundkvist, Marcus Alanen, Kim Sandström, Ivan Porres and Johan Lilius, MDE for SoC Design. Innovations in Systems and Software Engineering, Springer, 5(1):49–64, March 2009.
Author's contribution: The author contributed with ideas related to the definition of the domain-specific language, transformations and developed tool support for MICAS. The author also developed parts of the Coral Modeling Framework in parallel.

VII Johan Ersfolk, Johan Lilius, Torbjörn Lundkvist and Ivan Porres. A Tool for Efficient Combination and Evaluation of Reusable Design Assets. TUCS Technical Report number 975, April 2010.
Author's contribution: The author contributed in the study and specially authored Section 5 of this paper.

Chapter 2

Background

2.1 Modeling Languages in Software Engineering

Software models and modeling languages are used to create abstractions of a software system, to capture the most important design decisions of the software system and allow the reasoning about a system before it is built. Software models are used to create software designs and document, analyze, verify and validate them before implementing a solution in a programming language.

A software modeling language aims to provide developers and stakeholders using it with a set of relevant concepts that can be used to to create these abstractions. Different stakeholders may be interested in different views depending on their role in the development and use of the system. This has given rise to different modeling languages, each focusing on describing different aspects of the system. Examples of such are the structure, which models how a system is composed and structured, and behavior, which models how a system reacts to input under given conditions. Users of a modeling language need to choose an appropriate modeling language and use it to create meaningful models or views of the system.

In general, there are two main families of approaches for software modeling. One approach is to use a modeling language which is applicable to many software design problems in many different application domains. We refer to these languages as general-purpose modeling languages. General-purpose modeling languages are languages developed for describing systems using a general set of software concepts which apply to most software systems. Perhaps the most notable general-purpose modeling language is the Unified Modeling Language (UML) [74], standardized by the Object Management Group (OMG) [68]. The UML has been widely adopted in software development and can be considered *de facto* standard for software modeling. UML can be used to describe software using general object-oriented concepts, independently from the programming language eventually used to implement the software. There exists numerous tools for creating, editing and analyzing UML models, for example Rational Rose [42], Gentleware Poseidon [37], Magic Draw [59] and Eclipse Model Development Tools [2].

While general-purpose modeling languages aim to generalize software concepts in order to apply them to many software domains, another approach is to use a modeling language developed for a particular application domain. *Domain-specific modeling (DSM)* [47] brings the focus of the modeling to one particular, restricted application domain. A *domain-specific modeling language (DSML)* has a limited scope, often to a particular type of software based on a particular software framework, and are therefore by design not reusable in a different domain. On the other hand, as the domain is restricted, a higher expressivity within this domain can be achieved instead. However, as these languages are highly specific-purpose, the level of abstraction and modeling concepts used can be defined to more closely correspond to the problem domain in question, creating a more problem-oriented abstraction of the system under development.

Perhaps the most important characteristics of DSMLs is that they can often be viewed as both specification languages and as high-level programming languages. Combined with the use of domain concepts in the language, this can effectively hide the complexity of the actual implementation from the users of this language [93]. That is, a user of a DSML will not necessarily need to know how to implement the solution in a general-purpose programming language to be productive; ideally sufficient knowledge about the problem domain is enough. This can in some cases allow domain experts to develop their own applications despite lack of skills to implement the same program using general-purpose programming languages [61].

To obtain a deliverable or an executable program, the specification expressed in the DSML is processed further by a compiler. Typically the solution is either generated using an *application compiler* or *application generator* [21, 26] application to a general-purpose programming language, or executed using a language interpreter. This application contains the mapping from the domain concepts to code needed to produce the final artifact or execute the program. Program generation [26] can facilitate the reuse of optimized solutions to specific problems. For these reasons, domain-specific languages are considered a viable form of software reuse [53].

Domain-specific modeling languages can be considered to be a form of domain-specific languages (DSLs) [93], and the term DSL is therefore often used to denote a DSML. There exists numerous domain-specific languages used for a variety of purposes. For example, SQL for querying and updating data from relational databases [22], YACC [43] for generating parsers based on BNF grammars [5], TEX [50] for typesetting documents and many more are discussed in [89]. Among the above mentioned examples, the application compilers of YACC outputs the executable code of a parser, TEX outputs a device independent file that can be rendered to for example PDF or HTML, while SQL program statements embedded in an application are passed to an interpreter, which executes these statements and returns the results to the embedding application for further processing.

Perhaps the most significant difference between the above mentioned DSLs and DSMLs, is that DSMLs are represented as graphs defined using graph grammars or metamodels, often represented with a visual notation to allow graphical editing. Otherwise the usage patterns are very similar. However, DSMLs can also have other notations such as textual representations.

By using abstractions, the same artifacts are more likely to be used to communicate the system among stakeholders (customers, domain experts, requirements engineers, designers, programmers, testers etc.), as well as using them for testing, analysis, model checking as well as generating the full system code. This allows for a higher degree of automation in software development, which can have a positive effect on productivity and reduce the time required to specify, design, implement and test an application [47].

Although there exists many documented and successful DSMLs, most are developed in-house by software companies to bring benefits to their own development [93, 47]. This can make it difficult to find existing reusable solutions. In addition, existing solutions will need to be a superset of the requirements of the product to be built using DSM. If there is no available third party DSML to fit the problem domain, software companies will need to develop their own solution in order to benefit from using DSML. However, developing a new language and efficient use of it also implies a need to develop tool support for application development.

The development of a new DSML involves several tasks which can require a substantial effort to complete. These major tasks are defining the appropriate scope and abstraction of the domain, defining domain-constraints and DSML and tool-chains, mapping the DSML to executable code for application generators and maintaining the solution. This effort needs to be weighed against the effort required to develop a one-off solution in a conventional general-purpose programming language. On the other hand, if it is likely that many similar applications will be developed with slightly different requirements, this effort can be distributed over several application development projects. Studies by Weiss and Lai [98] suggest that for software product lines, three or more variants need to be built before a DSM solution can become cost-effective. The extra effort invested in finding a DSM solution will probably require more resources to complete than a one-off solution.

In the context of this thesis we will focus on tool support for creating DSM solutions, in particular tool infrastructure, language and editor definition and the support for model transformation in DSM tools. Out of scope of this thesis are the DSM related activities of domain engineering, such as domain analysis [10, 26] and how to define program generators. Program generation based on a graph defined by a DSML is a specific instance of model-to-text transformation, and an example of how this has been implemented can be found in [67]. In the next section we will provide a brief overview of the main development activities in DSM.

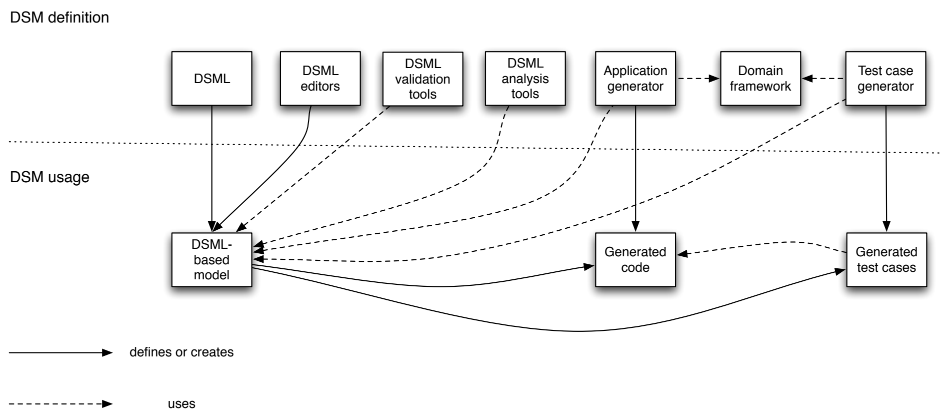

Figure 2.1: A basic DSM architecture; definition and usage.

2.2 Domain-Specific Modeling Language Development

2.2.1 DSM Architecture

Software development based on domain-specific languages is often based on a two-level application structure, which has a clear distinction between *definition* and *usage* of DSM, each with its own set of tools and key developer roles.

DSM definition involves defining and developing tool components needed to develop the domain-specific language and related tool support, such as language and editor definition, application generator or interpreter, domain-specific libraries and framework and development utilities (testing, analysis etc.). Each of these components can in turn be based on reusable framework specialized for developing one or more of these tool components, etc. languages or code generators. The process to systematically model and develop domain-specific solutions is referred to as *domain engineering* [26].

DSM usage is, as implied, the every day usage of the tools defined for the DSM to create concrete applications with their own set of requirements. In software development based on DSM, models remain the primary artifact for developing new applications. Thereby, DSM usage is primarily modeling activities such as creating and editing models and using these as input for code generators or other utilities.

A typical DSM architecture is depicted in Figure 2.1. On the top of this figure are the components defining the DSM application architecture and on the bottom are the artifacts involved in the usage of DSM and their relations. In this setting, the following components are defined during DSM definition: The domain-specific modeling language (DSML), the DSML editors, the DSML validation tools, the DSML analysis tools, the application and test case generators and finally, the domain framework.

The domain-specific modeling language contains the abstractions of the problem domain and describes the concepts and their relationships. The editors, here shown as a single component, are used to create and maintain models. Examples of such editors are diagram editors and form-based editors. The validation tools are used to check whether DSML models constructed using the editors are valid models. Examples of validation tools are well-formedness checks and simulation. Well-formedness checks [17, 18] are used to validate models against additional design constraints. Such checks can be carried out while the model is being created, or as a separate step before proceeding to e.g. code generation. Using simulation, it can be possible to determine whether an application based on a DSML model satisfies some given requirements, e.g. performance criteria, in parallel with the development of the application. The analysis tools can consist of a variety of different tools which can be used as part of the verification of DSML models. Important examples of such can be found in the field of model checking [20]. Models can also be analyzed to collect metrics [64].

The application generator is a program which task is to translate all valid models to executable systems. Essentially, the application generator contains a mapping of the problem domain to code and a set of rules how the output code objects and statements are glued together based on the input model. The complexity of the application generator will most certainly vary between different domains. Application generators can also use domain-specific framework, which can consist of ready-made libraries with domain-specific components and functions. This framework can be used for various purposes such as to simplify the generated code or to map generated code to slightly different target platforms (for example different hardware).

Similarly as an application is generated from a model expressed in a DSML, the same models can sometimes be used to generate test cases for the application, based on some additional heuristics. In [15], various approaches for model-based testing are discussed.

On the bottom of the figure are the main artifacts involved in DSM usage, produced using the components of the DSM tool. Ideally, the developer only needs to construct specifications of the system under development as models based on the DSML and execute the code generator to obtain the generated code (or more generally, the deliverable) and the test case generators to generate tests for the application. When interpretation is used instead of code generation, the models are loaded and executed, either as is or via an intermediate format, in an application interpreter.

2.2.2 DSM Development Activities

The use of a particular domain-specific language is in practise often limited to a setting consisting of one single software company. Yet, developing applications using DSM can be beneficial as development is done using high-level domain concepts rather than general-purpose programming languages, which in turn can increase productivity. To benefit from DSM, a software company will often need to develop tool support in-house, which is likely to be a long-term investment.

Domain-specific modeling solution development is in general divided into three major phases: Analysis, implementation and usage [93].

The analysis phase consists mainly of activities related to defining the actual DSML. These activities are defining the domain and domain boundaries, collecting relevant information about the domain and defining the concepts needed to describe applications in this domain. As an output of this process, a DSML for describing applications in this domain can be defined. This phase is perhaps the most demanding in DSML development, as it requires a high degree of maturity in the application domain as well as sufficient knowledge and experience with the application domain [47].

The implementation phase in turn consists of the activities related to implement the tool support required to start developing applications using the DSML. This involves the development of domain-specific application libraries. In this phase tool support for constructing applications based on the DSML is also developed. This tool support in the context of model-based approaches typically include the development of editors, e.g. diagram editors and form-based editors. Depending on the chosen approach to actually create executable programs or the deliverable, typically either a DSML interpreter or a code generator program is also developed.

In all, the outcome of this phase is a tool environment supporting development of new applications in the given domain. Consequently, the third and final phase is the usage of these tools.

2.3 Domain-Specific Modeling Tool Frameworks

Domain-specific modeling is often heavily based on tools as increased automation is one of the main drivers of adopting DSM. Examples of tools used in the DSM development process are tools to define and edit modeling languages, model analysis tools, verification and validation tools and application generators or interpreters. These tools are used together as part of a custom tool chain. These tools can be developed in-house specifically to the domain or acquired from a third party. As these tools can support application development for a narrow-focused domain, such tools are per definition domain-specific and not reusable as such for other future application domains. Hence, it is likely that the effort of reusing existing domain-specific applications when creating similar tool sets for new application domains is high. Despite the fact DSM solutions are developed to facilitate reuse in software development, the initial cost of development of such tools can still be high. These initial costs can be prohibitive for software companies exploring the possibility of using DSM in software development.

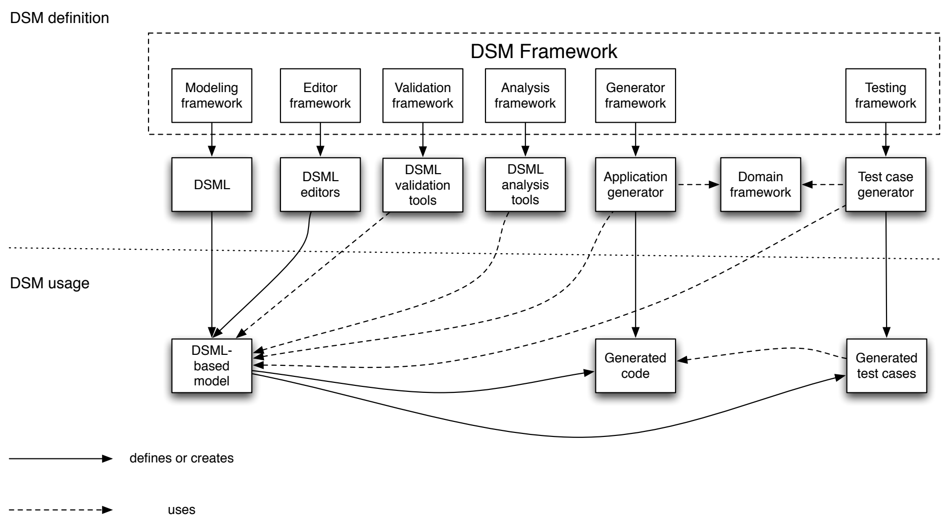

Figure 2.2: A basic DSM architecture, based on a reusable DSM framework

Despite DSM solutions are often developed as long-term investments, these will require maintenance over time, for example if the domain evolves or if increased understanding of the domain is gained. Still, there are no guarantees the lifetime of the applications will be high. As there can be benefits with DSM, developers implementing tool support for a DSL needs to take into account gains of developing new applications rapidly, towards the effort of developing the tool support for the DSL.

To reduce the effort of developing tool support for a DSML, numerous DSM tool frameworks have been developed. Examples of such frameworks are Eclipse Graphical Modeling Framework (GMF) [3], GME [55], Metacase MetaEdit+ [49], Atom^3 [28], Microsoft Visual Studio DSL Tools [25] and the Coral Modeling Framework (Publication IV).

DSM frameworks generally consist of a set of tools and utilities frequently needed to develop customized tools supporting the development of applications using DSM. An example of such a setting is shown in Figure 2.2. In this case, a generic DSM framework (framed in the figure) is customized for a specific domain using tool-specific extensions to produce similar components as previously shown in Figure 2.1. A highly customizable tool is likelier to be reusable in several application domains, but the effort involved in customizing the tool can be high, and how exactly the customization is carried out can vary significantly. For an organization developing many DSM solutions, reusing the same general tools for many domains can bring benefits as the initial effort can possibly be reduced to involve only the customizing of the general DSM framework.

While there exists various philosophies of organizing DSM tool frameworks, many consist of one tool set including the tools required to construct domain-specific tools (DSM definition) and another consisting of runtime components used when particular applications are developed (DSM usage). Sometimes, both definition and usage is carried out in the same tool. Central in these frameworks is the support for a *modeling framework*. A modeling framework defines how models and modeling languages are defined, represented and interchanged uniformly. We discuss modeling frameworks more in Section 2.5 and Publication I. DSM tool frameworks provide facilities to define basic tool support for model manipulation. Typically, this tool support includes visual language editors, allowing models to be manipulated visually and various form-based editors allows more data-oriented editing approaches. Support for automation is often included, as it is an essential element in domain-specific modeling. Examples of such components are various analysis tools used in e.g. constraint validation [96, 79] and metrics calculation [62], model query and transformation components and code generators.

2.4 Graphs and Graph Transformations

Software modeling tools can benefit from the fact that software models can be considered as graphs. The theoretical and mathematical foundation of graphs is given by graph theory [29]. One of the most notable benefits of applying graph theory to software modeling are in the field of visual language theory [60] and graph transformation [85, 31].

In this section we will define the most important concepts of graphs and graph transformations we use in this thesis. The following definitions are based on [31]:

In the context of this thesis we will use solely directed graphs. A directed graph $G = (V, E, s, t)$ consists of a set V of nodes (vertices), a set E of edges, and the functions $s, t : E \rightarrow V$, where s is the source and t is the target functions.

In graph theory, graph morphisms are used to relate two graphs to create a mapping. A graph morphism is defined as follows: Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ where $i \in \{1, 2\}$, a graph morphism $f : G_1 \rightarrow G_2, f = (f_V, f_E)$ consists of two functions $f_V : V_1 \rightarrow V_2$ and $f_E : E_1 \rightarrow E_2$ that preserve the source and target functions. A graph morphism f is injective or surjective if both f_V, f_E are injective or surjective respectively. A graph isomorphism is defined when f is bijective (injective and surjective).

Graphs can be classified as typed, labeled (also called attributed) and hierarchical graphs, each providing new characteristics to the graph.

A type graph defines a set of types, which are assigned to the nodes and edges of a graph. Graphs are typed by defining a graph morphism between the graph and the type graph. The definition is as follows: A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called the node and edge type alphabets. A typed graph is defined as a tuple $(G, type)$ of a graph G together with the morphism $G \rightarrow TG$.

Typed graph morphisms can be used to define graph morphisms such that the morphism between typed graphs are consistent with the type graph: Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$.

Labeled graphs [32] are used to add additional labeled information to nodes and edges of a graph as key–value pairs. It is one of the fundamental mechanisms of carrying data in graphs. Labeled graphs are defined as follows: A label alphabet $L = (L_V, L_E)$ consists of a set L_V of node labels and a set of L_E edge labels. A labeled graph $G = (V, E, s, t, l_V, l_E)$ consists of an underlying graph $G^0 = (V, E, s, t)$ together with label functions $l_V = V \to L_V$ and $l_E = E \to L_E$. Type graphs can also be labeled. In this case there is a bijective correspondence between the labels of the typed graph and type graph.

There are also hierarchical graphs [30], where there are a special type of edges called frames. These frames can contain a subgraph, which in turn can be a hierarchical graph. While hierarchical graphs are useful for dealing with composition in graphs, we will not deal with this category of graphs in this thesis.

A *graph grammar* can be used to restrict which graphs that can be constructed and is therefore a fundamental mechanism to define graph-based languages. A graph grammar consists of the set of productions or rules which represents the possible derivation steps which can be applied to construct all valid graphs. A production p can be defined as $p = (LHS, RHS)$ where LHS describes the left-hand side and RHS the right-hand side. The LHS describes a graph for which the rule is valid to be applied on a target graph, while the RHS describes a graph after the rule is applied. The actual application of the rule involves replacing an occurrence of the LHS by an occurrence of the RHS.

A graph grammar is used in a *graph transformation system* and defines the rules and the theoretical aspects of applying graph grammar productions on graphs. Different solutions of how the application of productions are applied has lead to a variety of graph transformation approaches. A detailed overview of the main approaches can be found in [85, 31] .

Perhaps the most widely used graph transformation systems are based on the algebraic approaches, of which the *double-pushout approach* is the most notable. The concept of a pushout is used to describe how the gluing of graphs occurs, that is, how nodes and edges are connected in a target graph. In the double-pushout approach a production is described as $p = (L, K, R)$, where L and R describes the left- and right-hand sides and K is the intersection $L \cap R$. K can also be referred to as a mapping between L and R. The productions are also declarative, L and R can also be seen as pre- and post-conditions. The application of this production (i.e. a transformation) can be understood as $L \backslash K$ being the part to be deleted, and $R \backslash K$ the part to be created.

In the context of this thesis, we are interested in how these productions can be used to obtain a new graph. In Figure 2.3 the main idea of the application of a production in the double-pushout approach is shown. The production is here written in the form $p = (L \leftarrow K \rightarrow R)$. The two squares (PO1) and (PO2) illustrate the two pushouts, and the vertical arrows describe morphisms between the production and the graphs; L and G (the source graph), R and H (the derived graph) as well as between the mapping structures K and D (the context graph) which in turn must be valid graphs. This figure shows the conditions which must hold in the application of this rule: (PO1) the gluing of $L\backslash K$ and D must result in G and (PO2) the gluing of $R\backslash K$ and D must result in H. These gluings form two pushouts, hence the name of the approach.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(PO1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(PO2)"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 2.3: The double-pushout approach

The actual application of the rule to obtain the derived graph H, is to find a morphism (match) of $m : L \rightarrow G$ and construct a graph $D := (G\backslash m(L)) \cup m(K)$, and remove $L\backslash K$ from G (PO1). Then, in a second step (PO2), $R\backslash K$ is glued onto D, resulting in the derived graph H.

We have now presented the most fundamental concepts of graphs and graph transformations used in this thesis. We will discuss graph query and graph transformations further in Chapter 3.

2.5 Software Models as Graphs

In the context of our work, we consider software modeling to be the manipulation, specification, analysis and the transformation of many related graphs at different levels of abstraction. Graph theory allows the definition and execution of these activities to occur with formal precision.

Software models are no longer only used for communicating the systems among humans, they are also used by software programs to perform automated tasks, hence software models needs to be based on formal languages [40].

The widespread use of models in software development has given rise to the need of standardization of modeling languages and their technical space. The Object Management Group (OMG) [68] is a non-profit, vendor-neutral software consortium setting standards in the area of object-oriented computing. Perhaps the most known OMG standard is the Unified Modeling Language (UML) [74].

Standards such as UML have not been specified in isolation. The OMG manages a process where standardization of technologies can be proposed and proposals for standards are invited. The members of OMG with an interest in the technology

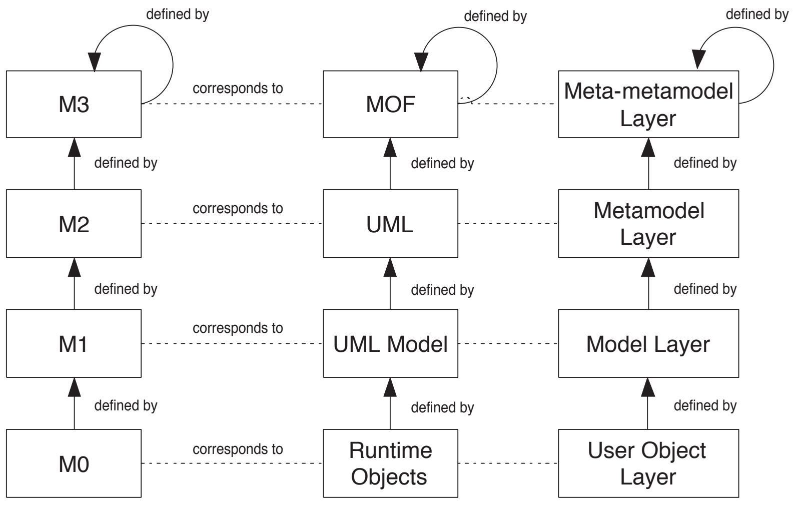

Figure 2.4: The OMG Four-Layer Metadata Architecture

can contribute by submitting their own proposal and collaborate to refine the proposal towards a new standard. All OMG standards are freely available from the OMG website. The OMG manages a large collection of standards, of which UML, MOF (Meta Object Facility), CWM (Common Warehouse Metamodel) [72] and XMI (XML Metadata Interchange) [76], are the most notable, forming the flagship standard Model Driven Architecture (MDA) [82].

The OMG standards are based on a four-layered meta-data architecture coined by Kotteman and Konsynski [51], defining a hierarchical relationship of models and modeling languages, depicted in Figure 2.4. The relationship between two layers in this architecture is analogous to that of the relationship of classes and instances in object-oriented software development. Hence, an instance of the concepts available on a certain level will form the concepts at the next layer. This formalism of defining languages is referred to as metamodeling. We consider metamodeling to be an alternative approach to the definition of graph grammars in the context of graphs.

The four-layered meta-data architecture is based on the notion of a modeling language with the expressive capability of defining all modeling languages, including itself. This language is placed at the M3 layer. In the OMG standards, this layer is represented by MOF or the UML Infrastructure [73]. This layer is referred to as the meta-metamodel or metalanguage layer. The language on the M3 layer is per definition in the four-layered architecture self-defining, and is therefore considered a fix point, hence this language is often fixed or hard-wired in tools utilizing all levels of the architecture. Based on the concepts for modeling language definition defined at the M3 layer, the modeling language or metamodel is defined

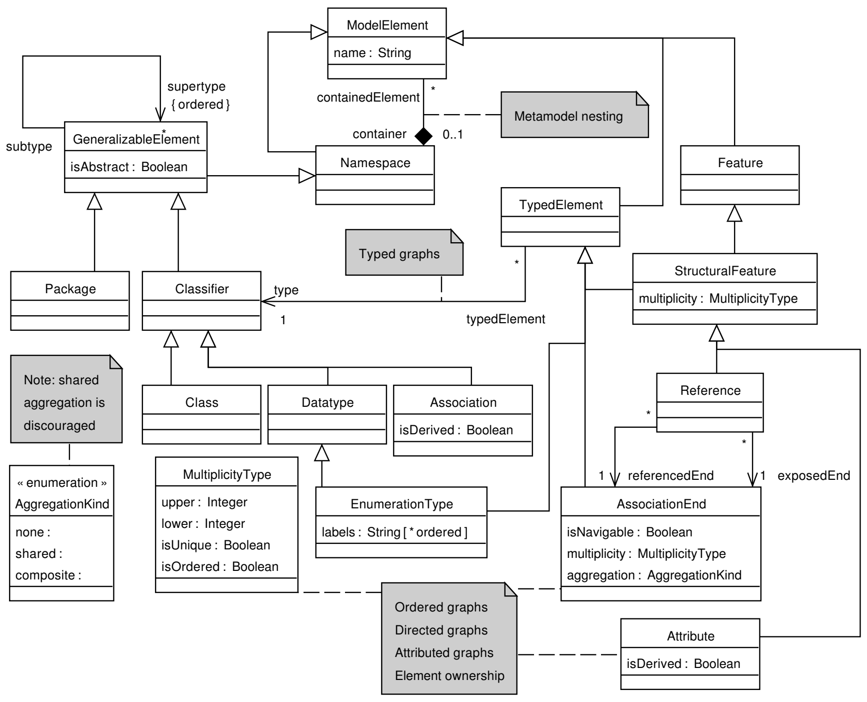

Figure 2.5: A fragment of the MOF metamodel

at the M2 layer. At this layer modeling languages such as UML, are defined. The modeling language defines the concepts that in turn can be used to construct all models conforming to it. The M1 layer, or the model layer, contains actual models used to e.g. describe a particular software system. While the M3 and M2 layers are often static when designing a system, the models on the M1 layer forms the actual model artifacts, such as UML models, developed by system designers. The fourth, M0 layer defines represents actual instances of model data in a run-time environment, such as objects in computer memory. It is therefore out of scope for the OMG modeling standards and is often omitted.

A concrete example of the organization of metamodeling languages, modeling languages and models can be found in Figures 2.5, 2.6, 2.7 and 2.8, each residing on it's own layer, respectively. As described previously, each layer defines the next via object instantiation. In these figures, Figure 2.5 describes the MOF meta-metamodel residing on layer M3. Based on the concepts of MOF, UML can be defined on layer M2. A fragment of the UML metamodel is shown in Figure 2.6. Again, the concepts defined in the UML metamodel are instantiated to models residing on layer M1. In this case, we show two different diagrams in Figures 2.7 and 2.8. These figures show two different representations of the same model fragment, Figure 2.7 shows a UML diagram in the UML concrete syntax and

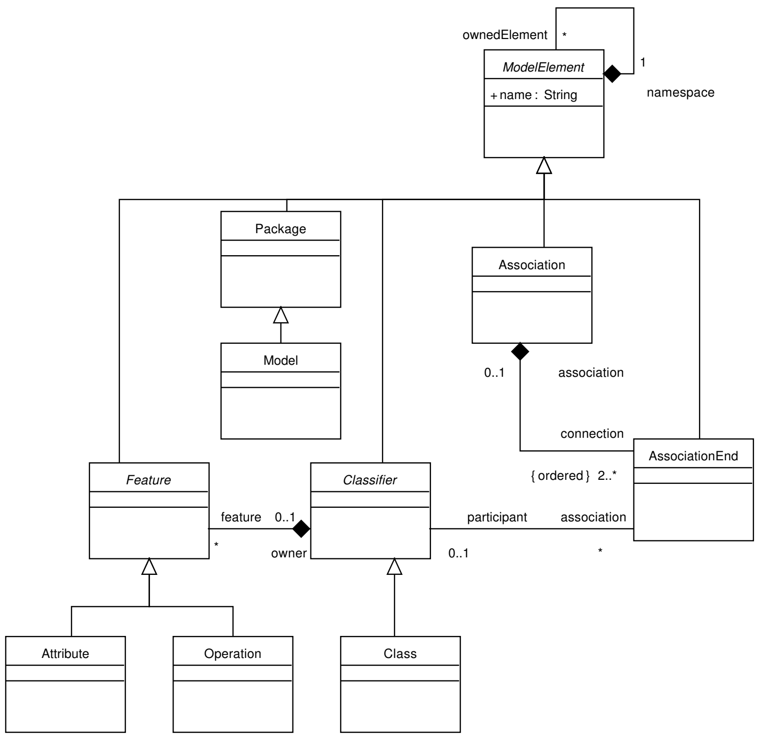

Figure 2.6: A fragment of the UML metamodel

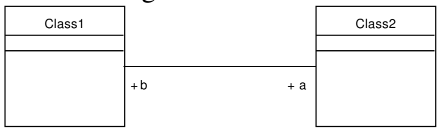

Figure 2.7: An example UML Class Diagram

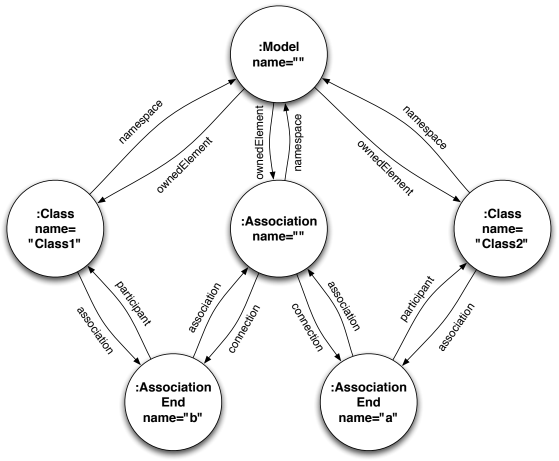

Figure 2.8: A graph representation of the Class Diagram from Figure 2.7.

Figure 2.8 shows the abstract syntax of the same model fragment. In Figure 2.8, each model element (instance of a metaclass) is represented by a labeled node and and each relationship (instance of a meta-association) as a directed, labeled edge.

The organization of models and modeling languages are an important part of a software modeling framework. The above examples show which are the artifacts and how these have been defined according to the modeling framework based on MOF.

2.6 Models and Modeling Languages as Graphs

In Publication I, we discuss and compare the modeling frameworks for software engineering in more detail. These frameworks are MOF, Graph eXchange Language (GXL) [100, 99] and the Eclipse Modeling Framework ECORE [16]. A modeling framework is essential in the definition of domain-specific modeling languages and consequently tools, as it defines how models and modeling languages are represented and organized in tools. Modeling frameworks can also include standards for model interchange and visualization, and as a consequence the choice of modeling framework can have an impact on the interchangeability of models and modeling languages, as modeling tools are often limited to a single modeling framework. Of the modeling frameworks in the study, all frameworks implemented explicit M3, M2, M1 layers, despite the fact that the frameworks had emerged from different backgrounds: GXL has emerged as a standard from graph theory community, MOF as a standardization effort from the industry and ECORE as a result from practical needs and open-source collaboration. While there is a difference in how languages are defined and interchanged, each of the approach had important benefits, for example GXL can interchange models without metamodel definitions and ECORE benefits from the increasing popularity of the framework and the rather small metametamodel, and MOF from being recognized as an OMG standard. From a practical implementation point-of-view, the difference between the approaches were minor.

Chapter 3

Graph Query and Transformation

3.1 Motivation

Graph theory can be used as a foundation to software modeling as software models can be seen as graphs and defining modeling languages can correspondingly be regarded as a method of defining graph grammars.

Graphs can contain large amounts of structured information and are therefore often used for representing and solving many important problems in computer science. Many fields of graph theory, such as graph transformation [85], relies on an efficient solution of the subgraph isomorphism [92] and related graph matching problems.

Graph query languages can be based on graph matching theory. In software development, graph query is used to find occurrences on subgraphs in a graph. Typical applications are pattern matching, specification and validation of constraints, graph transformation, metrics calculation et cetera. For example, in constraint validation, the absence of a given subgraph may declare the graph representing a model invalid [65]. In graph transformation, graph query is used to determine whether a *graph transformation rule* can be applied to construct a new graph. In software metrics, a query language in combination with aggregation can be used to calculate metrics [64].

Numerous query languages exist which can be used in software development. Many of these have been implemented for the use in conjunction with graph transformation languages. Notable examples are PROGRES [101, 13, 90], AGG [86], VIATRA [11], Fujaba [66] and GReAT [4]. The Object Management Group has also proposed QVT [81] for use in software development. All of these approaches uses advanced approaches for graph matching.

Subgraph isomorphism [92] can be used to detect if a graph G has an isomorphic subgraph H. For a positive match, a subgraph of G, H' isomorphic to H is obtained. In the simplest form of this approach both G and H are expressed as graphs and hence can be seen as a declarative approach to graph matching. While the basic approach is limited to detect isomorphisms in graphs, it has been extended with new variables to enable queries of higher complexity. Examples of such extensions are the inclusion of negative application conditions [38], path expressions and multi-objects (or also referred to as set-nodes) [90]. Path expressions are used to describe transitive relationships in graphs over many nodes and multi-objects bind multiple instances of a node to a single match. However, these approaches may not be sufficient for describing many hierarchical and recursive structures commonly occurring in many computer languages. Examples of such queries can be found in UML Class and Package diagrams and State Diagrams, where these can be needed for queries for inheritance hierarchies, nested Packages and nested States, respectively.

In Publication II we present a query language, CQuery, based on subgraph isomorphism to propose a new graph matching variable, the *star operator*. This language also supports the *isomorphic* and *negative operators*. In all cases the scope of matching operators are *regions*, which are annotated subgraphs in our approach. We have chosen to base our approach on subgraph isomorphism as we consider query languages should be declarative.

3.2 Regions and Matching Operators

In our approach, we operate on directed, typed and attributed graphs, conforming to a graph grammar. For the purposes of evaluation, we have based our approach on graphs defined using metamodeling, which we consider to be an alternative approach for defining graph grammars. A pattern consists of two parts, a graph defined similarly as the target graph and a second graph consisting of query language annotations.

A pattern graph consists of a graph partitioned in one or more *regions*, where each region is a connected subgraph of the pattern. Each region is associated to one matching operator. Consequently, the regions are non-overlapping and nodes belong to exactly one region. By using regions in combination with matching operators, it allows the defining of the scope of the operator to a subgraph rather than single nodes or edges in the pattern. This contributes to a certain generality in the operator semantics and is beneficial in case the query language is extended with new operators.

An edge can overlap region boundaries if it interconnects nodes in adjacent regions. We refer to such edges as *connection points*. These can be computed automatically if two given nodes are assigned to different regions. Connection points are used by the matching algorithm to traverse between regions as well as to validate whether operator-specific requirements regarding the connection points are satisfied before attempting to execute a query. We will discuss these requirements later in this section.

The isomorphic operator applied on a region in a pattern graph, describes an occurrence of an isomorphic subgraph in the target graph. When the pattern graph consists of only isomorphic regions or there are several adjacent isomorphic regions, the isomorphic regions can be merged without without affecting the resulting mapping between the pattern and the target graph.

The negative operator applied to a region describes the absence of a subgraph in the target graph. For a failed match, a subgraph isomorphic for the negative region is found in the target graph. Similar definitions of the negative operator in the context of graph transformations can be found in [38].

The star operator can be used to describe hierarchical or recursive structures and is conceptually similar to the star operator defined in Kleene algebra [52]. Star regions can be used to express a pattern where a subgraph isomorphic to the star region can occur zero or more times. Hence, the star region represents a group of patterns which can be expanded an arbitrary amount of times and be replaced by isomorphic regions of the same content. The resulting patterns can then be matched against a target graph.

To ensure the star region can be expanded an arbitrary amount of times, the star region needs to have two connection points to two other isomorphic regions. The edges representing these connection points must be derived from the same relationship in the grammar or metamodel, and define the point where star regions are expanded. Within the region, the star region can still have an arbitrary structure. Since the connection points have a compatible edge, the structure of the generated patterns will not violate the graph grammar or metamodel.

An example pattern is shown in Figure 3.1. The figure shows a graph pattern (above) with a star region that is used to generate new patterns (below). The pattern G_1 is generated by replacing one of the edges at the connection points (marked with circles) in G with an empty graph and rewriting the edge m (bold). Pattern G_2 is consequently obtained by replacing m in G_1 with an occurrence of star region R_2 and rewriting m. By repeating this process G_3 can be obtained.

Using for example subgraph isomorphism, the patterns $G_{1..n}$, generated from a pattern G, can be matched against a target graph. It must, however, be noted that generating intermediary patterns for the purposes of a matching application may be inefficient in large graphs, in comparison with an approach where the star regions are expanded dynamically during the matching process.

The star operator applied to regions can bring benefits when defining graph queries which involve hierarchical or recursive structures, especially if these structures involve subgraphs with more than one element. In Publication II, two examples in UML 1.4 [71] are presented for UML Generalization and UML CompositeStates. In UML 2.0 [74], the part of the metamodel involving state machines was changed, such that UML 1.4 CompositeStates were replaced by UML 2.0 States containing UML 2.0 Regions. Consequently in UML 2.0, star regions to express state hierarchies require a pattern with more elements to express a similar query.

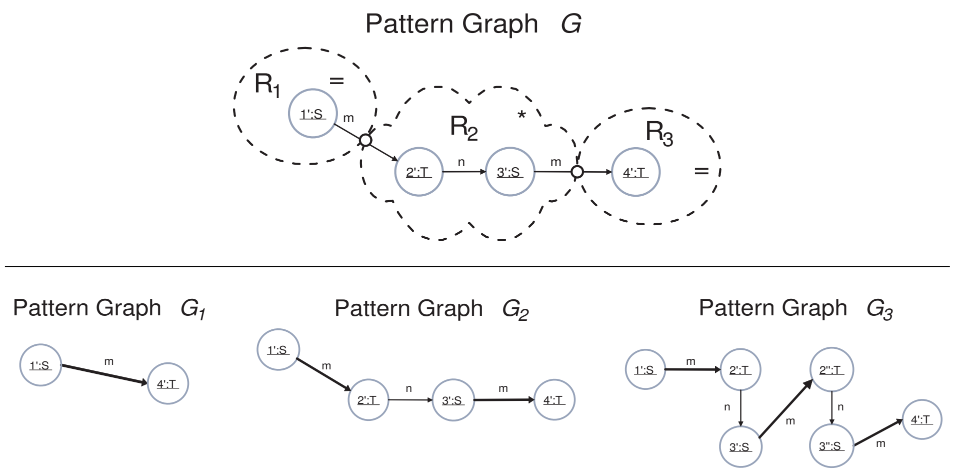

Figure 3.1: (Top) A pattern graph formed by two isomorphic regions, R_1 and R_3, and one star region R_2. (Bottom) Three possible patterns that could be generated from pattern G in the top of the figure.

3.3 Matching Algorithm

In Publication II an algorithm for matching patterns with regions is presented. The algorithm is an extension of an existing graph matching algorithm. An existing graph matching algorithm is still used to match the content of an individual region. We have in our work extended an algorithm presented by Lillqvist [56] based on CSP [91] and VF2 [23, 24].

The presented matching algorithm is based on an approach where the star regions are dynamically expanded during the matching process. This approach was chosen for efficiency reasons, as the number of patterns to be generated from a pattern containing star regions is arbitrary. The algorithm attempts to match one region at the time before proceeding to match adjacent regions.

In our experiments we found that the region matching algorithm is significantly faster if the starting point of execution is the largest isomorphic region in the pattern. We found that this is related to the fact that a larger region will be less likely to have multiple matches. Despite the matching algorithm operates on regions, matches are reported as a mapping between the nodes in the region and target graph is returned. For star region nodes, a mapping to a target graph node is reported as an ordered set.

3.4 Integration in Tool Environments

An experimental implementation supporting the concepts presented in the previous section was implemented in an experimental DSM tool, Coral. In this modeling tool, a graphical editor for CQuery and a matching engine was implemented. Coral treats languages as and their respective editors as modules, and the CQuery language was designed to contain unidirectional references to models written in any other modeling language loaded into the tool. The editor implementation followed a similar approach. The CQuery editor focused on creating the patterns with regions, operators and matching instructions to construct the model elements for the pattern. The CQuery editor dynamically loaded the editors of the target language (for example a UML Class Diagram editor) and added the CQuery actions to it. Consequently, there was no need to modify any of the existing metamodels or tools to support CQuery. As the CQuery editor also extends the diagram editor, it is possible to construct queries visually.

Query languages are fundamental in many applications based on graphs. The CQuery implementation is an integral part of the Coral Modeling Framework and many concrete modeling applications. Examples are a model search facility where a query pattern is loaded into a tool and occurrences of the pattern are matched towards models. CQuery is also used in a constraint evaluation component, which uses a set of pre-defined patterns describing well-formedness rules. The constraint evaluation component continuously checks for violations and reports offending elements. CQuery is also used as a matching facility for a model to text component.

Based on the CQuery language a graph transformation engine using the *double pushout approach* [85], called DPOTrans, was implemented. In this transformation approach, transformation rules are given as a pair of *left-hand side* (LHS) and *right-hand side* (RHS) graphs with an explicit mapping M between LHS and RHS. In DPOTrans, CQuery patterns are used to specify the LHS and RHS and to search for an occurrence of the LHS in the target graph.

The inclusion of a star operator in the double pushout approach differs significantly from the inclusion of the isomorphic and negative operators. The negative operator is widely used in many graph transformation systems, and is generally accepted as a standard application condition to be used on the LHS of the graph transformation rule [38]. Thus, for a LHS rule, the occurrence of a negative region in the target graph will prevent the creation of a mapping between the LHS and the target and consequently prevent the application of the RHS.

The star operator can be used to express that a specific region in a graph can be recursively matched in a graph a number of times which is not known *a priori*. However, after a valid mapping has been found, this number will be known. In our approach, star regions can occur in both LHS and RHS. The number of expansions of a star region required to match the pattern is identically applied to the corresponding star regions to dynamically create a new transformation rule, where the star regions are replaced by isomorphic regions. In this case, if there are mappings between nodes in corresponding star regions, these are also duplicated to maintain consistency. The duplication of the mappings have allowed the use of a general double pushout approach supporting star regions.

The transformation engine with support for star regions is implemented in the context of the Coral Modeling Framework, in which it is extensively used for defining transformation rules for model editing actions in the context of the diagram editor.

We have found that the inclusion of the star operator is beneficial in languages where complex hierarchies occur frequently, for example in UML Class and State Diagrams. As the star operator can describe such hierarchies at arbitrary depth, a single pattern with a star region can describe many patterns and consequently reduce the number of transformation rules needed to implement e.g. diagram editor transformation rules for a new modeling language.

3.5 Related Work and Discussion

The concept of introducing constructs to support hierarchies in graph matching and transformation approaches is not new and has received much attention. The PROGRES path expressions [85] describe transitive relationships which can be used to describe hierarchies in graph queries and transformation. In the PROGRES approach, however, path expressions are always defined over single nodes of compatible type on which a transitive relationship can occur. The Fujaba environment also presents a similar path expression construct [36]. Our approach extends this notion by allowing the definition of similar relationships over entire regions, while the contents of the star region is arbitrary. Our approach is also similar to PROGRES multi-objects, however, the star region operates on depth of multiple occurrences of a subgraph, while multi-objects describe multiple occurrences of a single in breadth. However, extensions for PROGRES has later been presented [35] to allow set-valued transformations, which are conceptually similar to regions in our approach, to make single node or edge operations and application conditions available to sets.

A similar construct as the star region, *SCORE subgraphs* is presented in [54]. In this approach, *interconnected set valued subgraphs* (ICONS) are presented, which can be used to match and transform repetitive subgraphs in patterns and graph transformation rules. This approach has been implemented as an extension to PROGRES. While the motivation for and the semantics of the ICONS are mostly similar to the star region, ICONS subgraphs are specified as sets of three almost identical subgraphs, *first*, *middle* and *last*, where the middle subgraph is similar to the star region. To the best of our knowledge, one significant difference is the ICONS subgraph cannot express the situation where the repetitive subgraph occurs zero or one times, whereas the star operator semantics are aligned with the Kleene star.

In [95], an approach for matching recursive graph structures based on magic sets used in the VIATRA2 language is presented. This approach is based on the idea that a graph pattern can be composed of a set of sub-patterns which can reference themselves to allow recursion in graph patterns. The presented approach supports the efficient matching of more general forms of recursion than the star operator, including the support for disjunction in the form of an OR operator. When recursion is combined with the OR operator, similar queries and transformations as provided by the star operator can be executed.

Chapter 4

Diagram Reconciliation

In the context of our research work, we represent the abstract and concrete syntax (diagrams) of a model as two different graphs which are maintained independently. We consider this separation important as it enables the abstract syntax to have multiple representations and visualizations in different contexts and diagrams and for these to be maintained independently. In the OMG [68] standards, modeling languages are defined using the Meta Object Facility (MOF) [77] or the UML 2 Infrastructure [73]. To represent the concrete syntax, OMG provides a standard for interchanging diagrams, called the UML 2.0 Diagram Interchange (DI) [80]. The DI is defined as a modeling language using the same metamodeling approach as the UML or MOF. The DI diagrams are stored alongside the abstract syntax for interchange using the XML Metadata Interchange (XMI) [76] standard. Despite this standard was originally proposed to be used to interchange UML diagrams, nothing prevents the standard from being used to interchange diagrams in domain-specific languages.

However, existing OMG standards do not specify how to define the relation (i.e. mapping) between the abstract syntax and diagrams. The DI standard is designed to contain only the structural information relevant for displaying diagrams, where DI elements are linked to corresponding abstract syntax elements, realizing separation of the abstract and concrete syntax. Additional elements are also used in DI to store information relevant to the correct structure and layout of the final diagram, such as placeholders for text information and visible and invisible compartments. Appendix C of the DI standard [80] presents a list of how the most common UML elements correspond to DI elements and which are the most common compartments used in UML 2.0 Diagrams. However, the level of detail of this information is not sufficient for processing diagrams in modeling tools. An exact specification of how to map models and diagrams is a requirement for correctly processing and interchanging DI diagrams in tools. To address this issue, OMG has published a *request for proposals* (RFP) for a new *model view to diagram* standard [78]. This RFP contains a set of requirements proposal submissions should address. Among these requirements are a mapping language from metamodels written using MOF and DI, as well as an extension to DI which allows the definition of notational symbols compliant with Standard Vector Graphics (SVG). A typical outcome of a RFP process is a standard, expressed as a language, with indications of it's use in software development processes or tools. To the best of our knowledge, we are not aware of any publicly available submission. In addition to the concerns presented in the RFP, we also consider the practical implications of using such diagram mapping languages are important.

Model transformation engines are examples of tool components, which take a model M and a transformation rule as input, executes the rule on M to produce a slightly different model M'. In our discussion, we generalize the concept of transformation engine to include all tool components which make changes to a model in any way. Many of these components may have been designed to operate on the abstract syntax alone. Hence, using these components can lead to a situation where the diagrams do no longer conform to the abstract syntax models or are lost. Since the diagrams are important for the human software developer, we consider that diagrams created or modified as a result of model transformation are updated accordingly to allow further manipulation using visual editors.

4.1 The Structure of DI Diagrams

The purpose of the OMG DI standard is to allow the diagrammatic representation of abstract syntax models. The focus of this language is to represent the structure and layout of a diagram. DI itself does not have any mechanisms for describing how diagrams are actually rendered. Rather, the use of DI is based on the assumption that a modeling tool will have built-in knowledge about the visual notation to be able to render an image of the diagram.

The DI is a small language consisting of only 22 metaclasses, of which the main concepts are GraphNode, GraphEdge, GraphConnector and SemanticModelBridge. GraphNodes are used to represent rectangular shapes, while GraphEdges represent represent edges. To provide anchor points for GraphEdges, GraphConnectors are used. All of these elements also contain properties to specify placement and dimensions in two dimensions. SemanticModelBridges are used to link DI elements to the abstract model. GraphElements (GraphNodes and GraphEdges) representing a UML element are connected using a Uml1SemanticModelBridge with a directed link. To add semantic information to GraphElements important for the correct layout and rendering, SimpleSemanticModelElements containing a string named *typeInfo*, are used.

DI GraphElements can contain other GraphElement to form a hierarchy to describe complex layout information and provide precise placeholders for rendered symbols or text. The GraphElements necessary to describe a single abstract model element can therefore be said to form a *subtree*. Exceptions to a strict tree structure

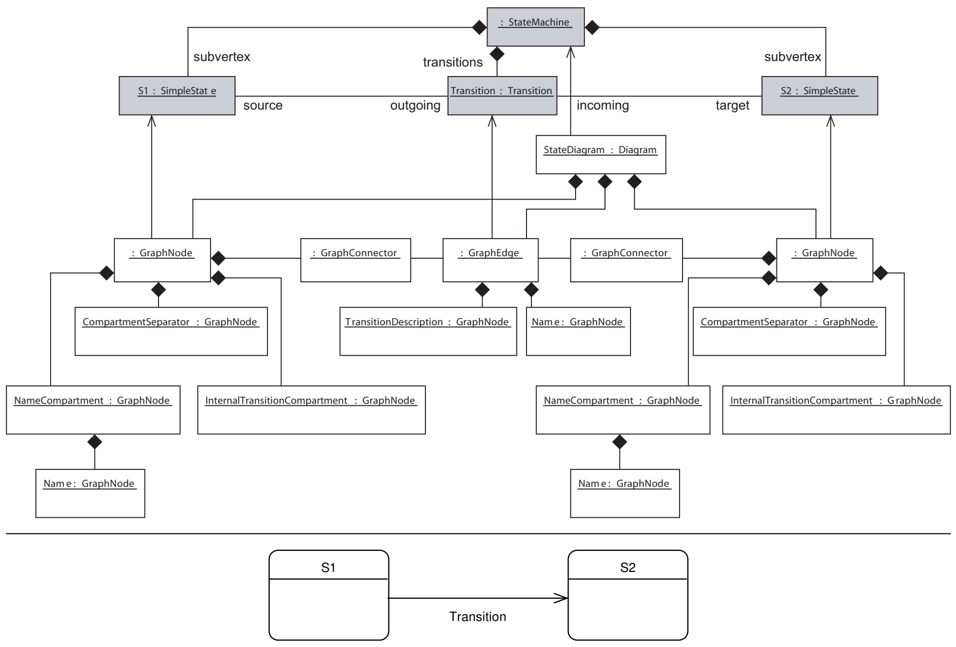

Figure 4.1: (Top) UML model in gray with two SimpleStates and a Transition and its representation in DI. (Bottom) DI diagram rendered using the UML concrete syntax.

are the GraphConnectors, interconnecting subtrees. A full DI Diagram is composed of a similar structure of subtrees. In this case, we consider a partitioning of the entire DI Diagram graph in subtrees, where a subtree is defined as a GraphElement mapped to an abstract model element via a Uml1SemanticModelBridge, including all transitively contained GraphElements.

In Figure 4.1, an example of a UML 1.4 State Diagram using DI is shown using a notation similar to a UML object diagram, where we show containment and directed links explicitly. In the top of this figure, SimpleSemanticModelElements are shown as instance names and Uml1SemanticModelBridges are shown as directed links to UML elements (gray). The DI elements with a Uml1SemanticModelBridge represent the root of each DI subtree. The bottom part shows the same DI diagram rendered using the UML concrete syntax.

From the figure, it can be seen that the mapping between a state machine and the corresponding diagram involves many DI elements organized such that the

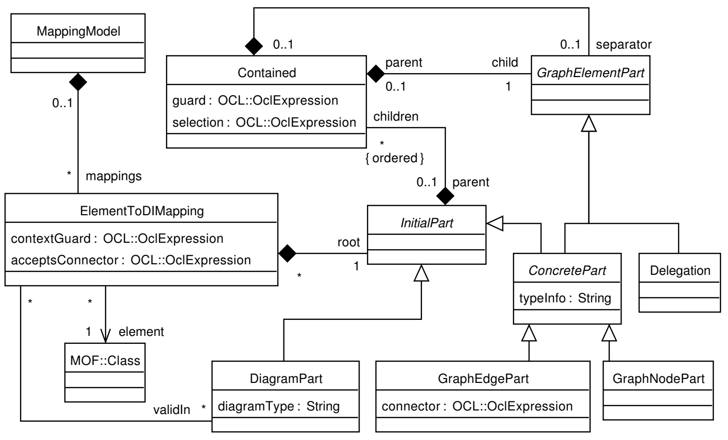

Figure 4.2: The DIML metamodel.

mapping is not trivial. Many of these DI elements are not visible in the rendered diagram, but are important for the layout of the diagram. However, to correctly create, modify and display a diagram, the exact mapping between the abstract and concrete syntax models need to be known.

4.2 The Diagram Interchange Mapping Language

To address this issue, we propose a language called the Diagram Interchange Mapping Language (DIML). The purpose of this language is to provide a mechanism to describe the relations between MOF-based modeling language and DI. The proposed mapping language also deals with the hierarchical composition of DI diagrams. The DIML metamodel is presented in Figure 4.2.

In Publication III we present a thorough discussion of the DIML. The DIML is a mapping language, defined as a modeling language. A mapping expressed in DIML is based on the idea that each metaclass in a modeling language is mapped using an ElementToDIMapping to one or more trees of DIML Parts. The ElementToDIMapping can contain a positive application condition, *context guards*, to enable multiple diagram representations for the same abstract model element. This context guard defaults to `true`, but can be specified arbitrarily as long as the guards are mutually exclusive.

Each DIML tree consists of an InitialPart as a root element, followed by a hierarchy of Contained-elements and GraphElementParts. The GraphElementParts specify a corresponding DI GraphElement in diagrams, while the Contained-elements specify that a GraphElement can contain other elements. The Contained-element also provides the main mechanisms of parametrization in a DIML tree, *guard* and *selection*, defined as OCL expressions. Guards are a mechanism to

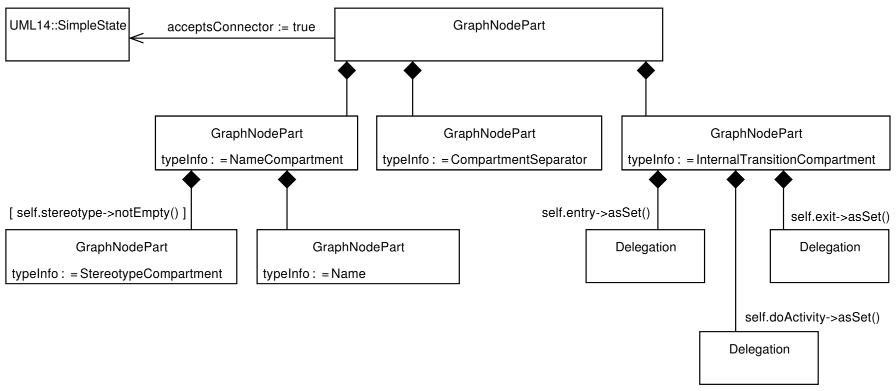

Figure 4.3: The DI mapping rule of UML 1.4 SimpleState.

control the creation of a specific subtree within a DIML tree. A guard will allow the creation of a subtree if the specified expression is evaluated to the value `true`. Selection expressions, used in conjunction with DIML Delegation-elements, are used to invoke another ElementToDIMapping. In this case, the selection expression returns a collection of abstract model elements, for which the corresponding mapping is invoked. Thus, Delegation only symbolizes the invocation of another mapping rule satisfying the context given by the abstract model. This mechanism allows the creation of DI Diagrams of arbitrary depth, adds scalability to the mapping language, as well as the reuse of existing DIML mappings in many different diagrams.

An example of a DIML mapping for a UML 1.4 SimpleState is shown in Figure 4.3.

4.3 Generation and Reconciliation of Diagrams

The DIML provides a mechanism to specify the mapping between abstract model data and DI Diagrams. Analogous to a graph transformation rule, this information can be used to transform an abstract syntax model into a diagram, while simultaneously creating links between the new diagram and the underlying abstract syntax model. In this section, we will discuss what we consider are the most important applications of DIML, diagram generation and reconciliation.

Diagram generation assumes only the abstract syntax model is present, and consequently can be useful when a model transformation program has created a new model without diagrams, while diagrams are later required. A similar scenario can also occur in a DSM tool setting, when exporting diagrams in DI if another diagram language is primarily used in the tool.

In Publication III an algorithm for diagram generation is presented. The diagram generation algorithm is based on the idea that for an abstract model element *e* and the desired diagram type, an ElementToDIMapping is found. The algorithm will then traverse the mapping rule using depth-first search and create diagram elements accordingly with respect to the structure and defined guards. Where Delegation elements are found in the rules, the algorithm evaluates the selection expression to obtain a set of abstract model elements and invokes other ElementToDIMappings for each element. Finally, when all diagram elements have been created, DI GraphConnectors are created for all GraphEdges.

It must be noted that diagrams created using the diagram generation algorithm will need to be layouted using an appropriate layout algorithm for the diagram type in question.

The creation of a new diagram is an important operation when no previous DI diagram exists for a particular abstract model. However, we assume a setting in which diagrams and the underlying abstract syntax model is being edited in parallel. If a diagram exists and changes are introduced to the underlying abstract syntax model, the diagram will no longer be consistent. To regenerate a diagram can be a slow and costly operation if only small changes are made to the abstract syntax model, for example by an editor or a model transformation program. In these cases, a more efficient approach is to perform only the changes required to keep the diagram up to date.

Diagram reconciliation is the process of maintaining consistency between an abstract model and a diagram after executing a model transformation program modifying the abstract model. To determine the changes made to the abstract model, a change description from a model repository can be used. A *diagram reconciliation component* is a component which, based on the changes made in the abstract model, performs only the necessary changes required to to bring the corresponding diagrams up to date, while preserving as much as possible from the existing diagrams. The mappings created using DIML are declarative constructs, meaning the diagram reconciliation component can use any algorithm for performing diagram reconciliation as long as the resulting diagrams correspond to the mappings.

In Publication III a high-level outline of a diagram reconciliation algorithm is presented. The presented algorithm takes an abstract model element *element* and the *slot* which has changed. For example, the element can be an instance of a UML CompositeState and the slot “subvertex”. Using this information, the algorithm finds all diagrammatic representations of the element and all the corresponding DIML mappings. Based on the structure of the DIML mapping, different reconciliation approaches are used. In the algorithm we present three functions for the reconciliation of DIML GraphEdgePart connectors, GraphElementParts and Delegations. All of these functions compare the DIML mapping against the existing DI subtree, and creates or removes diagram elements from the subtree depending on the changes in the abstract model and the DIML mapping.

The proposed high-level diagram reconciliation algorithm reconciles diagrams based on individual changes in a single slot of an abstract model element, as provided by a change description list. To improve the efficiency of the diagram reconciliation algorithm, especially for executing larger model transformation programs, it may be possible to group together some of the changes in the change description list, for example if there are several insertions of new elements into the same slot performed in sequence.

4.4 Integration in Tool Environments

While the DI standard enables a full separation of the abstract and concrete syntax by separating the diagrammatic elements into a diagram model, the DIML mappings and the diagram reconciliation component can be used in combination to decouple the changes made in the abstract syntax model from the updates required in the diagrams.

We have implemented DIML and a diagram reconciliation component in the Coral Modeling Framework tool. In this tool, DIML mappings are used to define and describe diagrams in the context of a diagram editor component. As a proof of concept, we have implemented DIML mappings for many diagrams in conjunction with UML 1.4 and UML 2.0 editors. A large collection of UML 1.4 mappings is presented in [58]. By using the mappings for UML 1.4, we have assessed compatibility with Gentleware Poseidon [37] version 3.0. As Gentleware was one of the main contributors to the DI standard [14], we consider Poseidon to be a reference implementation of DI. DIML mappings have also been implemented for numerous domain-specific languages, such as DIML itself, SOCOS [33], a language for defining invariant-based programming diagrams and MICAS and EFCO, presented as examples in Chapter 6. We consider that these examples show that DIML is a viable language for defining diagrams both for UML and domain-specific languages.

The diagram reconciliation component is implemented as a change propagation component built on top of the Coral model repository [7]. The model repository automatically provides a change description as a list of changes grouped together as a *transaction* when changes occur in models loaded in the repository. At the end of a transaction, the diagram reconciliation component is invoked to update all the diagrams for models which have changed. In Coral, models can be manipulated in several ways, using diagram editors, form-based editors, model transformations and imperative scripts loaded into a Python shell. Neither the model repository nor the diagram reconciliation can distinguish between which method is used for model manipulation. In fact, when a diagram is manipulated in the Coral diagram editor, the diagram editor performs only changes on the underlying abstract model using the model transformation engine described in Section 3.4, while new diagram elements are created in the diagrams as a result of executing the diagram reconciliation component.

4.5 Discussion

DIML and the diagram creation and reconciliation components can be seen as a *domain-specific* transformation language and engine respectively. In our approach, we treat the abstract model and diagram as two linked graphs, of which the diagram can be derived from the abstract syntax alone. The proposed DIML and its applications are as such suitable for maintaining diagrams only. While DIML can be seen as an *exogenous model transformation language* [63], we consider a specific-purpose approach is motivated since this allows for the development of a mapping language optimized to deal with the hierarchical structure of diagrams and maintaining diagrams is a crosscutting DSM tool concern. The DIML mappings are further independent of the definition of model transformation rules and their execution. That is, once DIML mappings for a modeling language have been created, new model transformation rules will not impose changes in the DIML mappings.

We have chosen an approach based on the OMG MOF and DI standards, as standard compliance is an important requirement for tool interoperability. However, the general principles of DIML and its applications are not as such limited to the OMG standards and can be adapted to other diagramming standards with similar characteristics.

To the best of our knowledge, there exists no similar approach for maintaining consistency between abstract syntax models and diagrams based on the OMG standards.

The work by Ráth et. al [88] presents an approach for synchronizing abstract and concrete syntax based on a mapping language between Eclipse EMF and the diagram metamodel of GMF using VIATRA2 live transformations [84]. This approach has many conceptual similarities with DIML and diagram reconciliation. In this approach, while an own bidirectional mapping language is used to describe the relation between models and diagrams, their diagram synchronization facility is implemented using a general-purpose model transformation language. In addition this approach supports synchronization from diagrams to abstract models. This mapping language has been implemented in ViatraDSM [84] tool.

Existing general purpose model transformation languages such as [27, 83, 41, 94, 12, 45, 75] are designed to deal with the abstract syntax alone. Such transformation languages could be extended to include abstract syntax model to diagram transformations. However, we still consider diagram definitions should be separated from model transformation rules, especially as all components in a DSM tool modifying abstract syntax may have an interest in maintaining diagrams.

Existing DSM tools such as Eclipse GMF [3], MetaEdit+ [49], GME [55] and Microsoft DSL Tools [25] present approaches for defining diagrams using their own diagramming languages, respectively. In these approaches, however, only one-to-one mappings between the abstract and concrete syntax are possible, whereas in DIML a single metaclass can have an arbitrary number of distinct visual representations in many different diagrams depending on the diagram it appears in and the context in which the abstract syntax model it is constructed. In the existing approaches the consistency of diagrams is only maintained when editing occurs using the diagram editor component, where as the presented diagram reconciliation component can maintain this consistency regardless of which component initiated the changes in the model.

Chapter 5

Integration of Model Transformation Components in DSM tools

In order to reduce the effort of developing tool support for DSM, several DSM frameworks have been proposed, such as the Eclipse EMF, MetaEdit+, GME, MS DSL Tools. These frameworks provide the possibility to construct basic DSM applications using built-in tools. Many DSM frameworks provide programmatic access to tool infrastructure and models using an application programming interface (API) to allow the integration of custom or third party components. Such components can, for example, be specialized editors, model transformation components and sometimes other modeling tools.

5.1 Functional Requirements

In Publication IV we have listed and motivated some of the functional requirements we consider most important in DSM tools: (R1) Multiple, standard and user defined visual languages, (R2) Define new languages based on metamodeling standards, (R3) Multiple interactive editors and (R4) Model transformation support.

While (R1) is rather obvious in a DSM framework, (R2) is not always supported in full. In our view, we consider that the full support of (R2) implies tool-independent and high-level metamodeling standards are used to define new modeling languages. We consider this important as it enables interchange of modeling languages which would ensure interoperability between tools and tool components. While requiring tools to adhere to a specific standard such as Meta Object Facility (MOF) [77] is practically difficult, approaches such as Eclipse ECORE [16], Kernel Meta Meta Model (KM3) [44] and Simple Metadata Description (SMD) [6] provide a high level of MOF compliance and are conceptually very similar. However, compatibility at the metamodeling framework level (OMG M3 level) does not still ensure full interoperability, as this requires that the metamodel (M2 level) externally and internally complies with the specification. In the OMG standards, XML Metadata Interchange (XMI) is proposed for interchanging models between tools. XMI itself operates on models alone (M1 level) and is as such independent of the modeling framework [8]. Studies such as [57] show that there are still interoperability issues due to implementation and interpretation differences of various standards. While there is a need to integrate various modeling tools and components, such differences as described above has lead to the development of integration approaches such as Modelbus [9] and tool adapters [46].

Support of (R3) and (R4) suggests a certain level of flexibility is required in DSM frameworks, by not restricting methods of representing, manipulating and transforming models. For example, while a diagram editor component clearly needs to maintain consistency between the abstract and concrete syntax (diagram) models, the focus of e.g. rule-based model transformation is on transforming abstract syntax models alone. Integrating model transformation components into a DSM tool can lead to a situation where diagrams are not maintained as the relation between the abstract syntax and diagrams is not trivial and methods to define and represent diagrams can be tool-specific. This can have a negative effect on the integration of different editors and transformation engines in a DSM tool. This has the implication that there is a need for mechanisms which collaborate to maintain consistency between the various artifacts such as models, diagrams and other derived artifacts, while still not restricting either the method of editing or transforming models or the order in which these are used.

5.2 Quality Attributes

In the implementation of these functional requirements, there are also some quality attributes we consider especially important in the implementation of DSM tools and tool components. Examples of these quality attributes are customizability and language independence (reusability), low tool coupling and low customization effort. A high reusability will allow a tool component to be used in the context of many different domain-specific tool environments. To achieve this we propose to base tool components on standards. This will in addition promote tool interoperability. A low coupling to the tool environment is important to ensure that new components can be integrated with the tool with reasonable effort. To achieve this, we propose to use simple patterns of communication between tool components, such that tool components need to focus only on the changes in the models, not on update or change propagation issues in the tool environment.

Perhaps the most difficult quality attribute is to ensure a low customization effort.

It can be argued that a generic tool framework, with a powerful and feature-rich application programming interface (API) library, where customization of the tool for a certain domain-specific solution is carried out by programming using a standard programming language such as C++ or Java, provides the highest potential of reusability, as there are few limitations on which applications can be developed. However, this will require special knowledge about both the domain-specific language and the tool specific API. It is also likely such a tool API will have a high learning curve and as components are customized using programming languages, customization will be time-consuming.

On the other hand, the customization could be based simply on facts about the DSML that can be automatically derived from the metamodel. In this case the metamodel would be used directly as the only input to editor generation components and editors can be obtained relatively quickly and with little customization effort. While this can be adequate in some cases, this approach will not necessarily allow customization and the set of features are limited to such that generally can be used with all domain-specific languages. If the editor generation is based on the semantics of the metamodeling language (e.g. MOF, ECore) the reuse potential is limited to what metamodels can be expressed in these languages.

The two mentioned customization methods represent two extremes, but indeed both can be beneficial to include in an open tool environment. For example, when an new model editing method is developed for a DSM solution, an API is an important prerequsite. When an automatically generated solution matches or exceeds the requirements of the DSM solution, there is a benefit with this approach.

To tackle the requirement of low customization effort, we propose an approach using generic tool components, which are customized using a graph-based, declarative approach. This approach is similar to DSM, but the target domains are different types of editors for domain-specific languages. That is, the basic approach to the customization of tool components is using a set of high-level declarative descriptions, rather than to use an API alone. Existing API is in this case not redundant, as it can be used as domain framework. Clearly, and in line with the principles of DSM and generative programming, this requires a mature domain (the editor component or more generally the method of editing in a tool) and sources of variability and means to create configurations. There are several such sources, for example the DSML metamodel, the metamodeling language, existing standards in the domain and how they have been applied and finally, commonly accepted and anticipated domain requirements. The variable features are then selected to create a customization (configuration) that matches the user requirements for the editor component in question. In the actual implementation of editor components, by finding general approaches to relate viewers and editors via metamodels and graph transformation the effort of customizing editors can be reduced. A concrete example of this is the diagram reconciliation component discussed in Chapter 4.

There are, however, some important trade-offs. The potential variability will certainly be more limited in comparison to using solely programming languages with an API. But on the other hand, this approach promotes reusability and less customization effort especially in the general case, while still providing sufficient capabilities of developing domain-specific language tool components and environments.

5.3 Tool Architecture

The architecture of a proposed DSM tool is outlined in Publication IV. The presented architecture is based on a shared model repository, where all application data is stored as graphs. The model repository acts as a modeling language independent storage of models and modeling languages and can be used as the core of a modeling tool. In this architecture, there is no explicit communication between tool components to exchange data: components may do this only by reading or writing data in the shared repository. We have chosen this approach as it minimizes the coupling between editor components working on models and change propagation components. This has the important side-effect that e.g. model viewers will need to react to data being written or modified in the model repository.

Central in this architecture is a communication pattern based on a transaction mechanism emitting notification events to registered tool components. The transaction mechanism allows tool components introducing changes in models, *active components*, to group a list related changes in models into transactions. When the transaction ends, a notification event is emitted containing a reference to the last transaction. This mechanism allows components responsible for change propagation, maintaining derived artifacts in the model repository and views, to react to the introduced changes. In our approach, we refer to such components as *reactive components* and concrete examples of such are diagram viewers and the diagram reconciliation component discussed in the previous chapter.

It is up to the active components to decide the size of the transaction. However, regarding interactive editor components, a typical transaction would include the changes in models due to executing an editor action. It is the responsibility of the reactive components to in turn perform change propagation efficiently based on the transactions alone. The changes in the model repository introduced by the reactive components are also monitored, and these changes are appended to the original transaction. This has the benefit that e.g. undo is a task of reversing the changes in the transaction list. The use of transactions has the benefit that both active and reactive components are able to work incrementally and driven by a set of related changes to be processed as units of work, which in turn ensures responsiveness in the modeling tool.

Based on this communication pattern, we propose an editor architecture for visual language editors. The architecture, explained in more detail in Publication IV, is based on a strict distinction of active and reactive modeling language independent tool components, each performing a designated task or maintaining a specific artifact stored in the model repository. Such artifacts can be models, diagrams, modeling languages, model transformation definitions etc. As there is no explicit communication between the various components, components detect the need to update views or regenerate derived artifacts solely based on changes in the artifact of interest. We consider this approach to promote high cohesion, as components need to focus only on one specific task, and low coupling, as the various components are coordinated via the model repository alone.

In our approach, the responsibility of updating views and derived artifacts is strictly delegated to designated components. In line with the results presented in Publication III, we consider the abstract and concrete syntax of models to be two separate artifacts maintained individually. In the context of this research work, we consider a mapping between the abstract and concrete syntax can be used to reconcile diagrams after changes occur in models. We have in our architecture used this result to show diagrams can be treated as a derived artifact in practise by integrating the diagram reconciliation engine to manage diagram updates.

In Figure 5.1, we exemplify this architecture in context of a DSM tool, where we for the sake of clarity show only the components related to displaying a diagram in a diagram viewer. In this figure, a tool setting consisting of a model repository is shown, where a diagram viewer, the diagram reconciliation engine and a model transformation engine are integrated as tool components. The components presented here are generic tool components configured using graphs, stored in the model repository. The model transformation component acts as an active component reading a model transformation definition (1.1) and executes it (1.2) in the context of a UML model, which is modified. The diagram reconciler then acts as a change propagation component and is notified that the UML model has changed (2.1). The diagram reconciler then reads the mappings for UML (2.2), to finally bring the UML diagram up to date. A second change propagation component, the diagram viewer, is notified of the changes in the diagram model (3.1) and redraws the diagram view based on the UML notation (3.2).

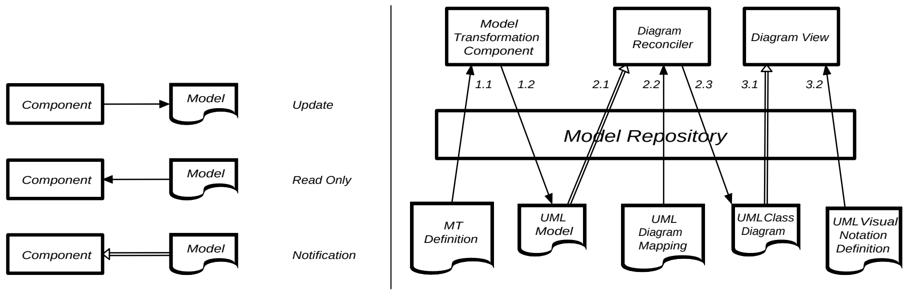

Figure 5.1: Using model transformations.

As we discuss in Publication III and IV, we can generalize the concept of model transformation engines to include interactive editor components. The architecture supports this generalization, as we can use the same architecture as described in Figure 5.1 to connect interactive editors.

As a proof of concept, we have implemented the presented architecture using generic, modeling language independent components in an experimental DSM framework, called the Coral Modeling Framework. In Coral, the controller part of our diagram editor is based solely on transformations executed on the abstract syntax of a model alone, while the diagram view is implemented as a change propagation component propagating the changes made in the diagram model by the diagram reconciliation engine.

In Coral, the change propagation components are driven by the transaction and notification mechanisms, and consequently, from the point of view of the model repository, there is no distinction of which type of component introduces the changes in models. This allows the integration of model transformation components into the tool environment based on the abstract syntax alone, while still ensuring existing diagrams are maintained consistently.

Chapter 6

Applications Using The Coral Modeling Framework

The Coral Modeling Framework can be used to build DSM tool environments supporting general-purpose languages and custom domain-specific languages. Coral is also a modeling tool, which implements various different customizable editors and components for manipulating models and transformation engines for executing model-to-model or model-to-text transformations. In these components, customizability has been ensured by basing the tools on metamodeling standards. This is aligned with the OMG meta-data architecture where the M3 can be considered a fixpoint. Thus, defining a DSM tool environment is ideally a matter of customizing the framework component to the application domain in particular.

We have chosen to use an approach based on DSM to implement individual framework components. The most important reasons for this is that we consider the problem domains of one single editor component narrow-focused and targeted enough to allow the description of a language-specific editor in terms of the tool component's problem domain and the metamodel of the DSML, while still being able to generate an efficient implementation. We have used this approach to create generic editor and tool components which are configurable for both general-purpose and domain-specific modeling languages. In contrast to developing language-specific components using general purpose programming languages and tool specific API's, the components are configured using graph-based, declarative artifacts as exemplified in Section 5 of Publication IV. This allows a both DSM *definition* and *usage* components to reuse substantial parts of the framework, effectively treating DSM definition components as editor components.

Next, we will present two DSM applications, which have been implemented in the Coral Modeling Framework.

6.1 MICAS

In Publication V and VI we present a tool environment for defining peripherals for mobile phones called MICAS (Microcode Architecture for System-on-Chip), built as customizations of the Coral Modeling Framework. The objective of these studies was to evaluate the use of a model based approach to design System-on-Chip (SoC) [69] architectures for embedded systems. The MICAS architecture describes a platform for static integration of components implementing a *process*, for controlling the resource assignment of components, as well as dynamic data processing, such as configuring services and streams between these components.

Components implementing a dedicated data-processing task (load files, decode video, display video), can then be combined as a domain consisting of a micro-controller and interconnected components (e.g. a video player). Several domains can in turn be interconnected as a network. The dynamic aspects of MICAS allows the definition and combination of services, which in turn consist of composed or basic streams. The optimal routing of data streams is the result of a design-space exploration process, where the static and dynamic configurations of a MICAS application is evaluated using simulation.

The MICAS tool is a tool environment consisting of a domain-specific language implementing several diagram types, model transformations chains between high-level and more detailed models, a set of domain-specific model editors, a constraint evaluation engine and code generators. In addition to these editors, the MICAS tool implemented libraries consisting of reusable components and transformation definitions.

The publications V and VI present two stages of the development of the MICAS tool, of which the former is based on a domain-specific modeling language (DSML) implemented as a UML profile extending UML Class and Object diagrams and the latter as a DSML defined using a metamodel. Both versions were implemented using Coral. We consider both approaches to represent alternative methods for defining DSMLs. A comparison of these approaches can be found in [97].

The UML profile for MICAS is defined using a set of dedicated UML Stereo-Types and TaggedValues. In the implementation of this profile we extended the UML diagram editor and created a new diagram editor which was in turn based on the MICAS problem domain. The diagram editor actions could in turn be customized to add the appropriate stereotypes and tagged values needed for other tool components such as code generators to properly identify MICAS models. However, MICAS models still conformed to the UML metamodel and could still be loaded and manipulated in other UML tools. While this was considered an advantage, a clearly identified drawback of the MICAS UML profile was the additional UML profile layer which needed to be taken into account when manipulating or traversing MICAS models in different tool components or when defining well-formedness rules for validating designs.

The latter implementation presented in Publication VI presents a more elaborate version of MICAS implemented using a DSML based on a metamodel. The conceptual MICAS metamodel presented in Publication V was extended with several new diagram types where many of the elements had alternative representations depending on the how the element was used in the MICAS models. In addition, many of the elements used in MICAS designs were defined in libraries, meaning a library element could be linked to many different models. As presented in Section 6 of Publication VI, the tool environment consisted of many different editors, each providing different editing methods of a MICAS model. While most model editing took place at the highest level of abstraction proposed by the MICAS language, we used two additional lower levels of abstraction, obtained using model-to-model transformations to bring the MICAS models closer to the abstraction level of the implementation from which executable simulation code was generated. All of these editors and transformation components operated on the abstract syntax alone while still supporting diagrams at all stages of the design process of MICAS applications.

6.2 EFCO

In Publication VII a tool environment for efficient combination and evaluation of reusable assets is presented. The EFCO (Efficient Feature COmbination evaluation) tool was designed to extend the workflow of an existing embedded systems design tool, CoFluent Studio [1, 19], with support for treating existing designs as reusable design assets. These design assets consists mainly of *Use Cases*, describing a component, which in turn consists of a hierarchy of *Features*, describing functional units, which in turn defined a set of *Services*. The EFCO tool extended CoFluent Studio by implementing a library of design assets, from which these assets could be combined [34] and explored to create new application designs. These designs could in turn be exported back to CoFluent Studio for the purposes of evaluation by simulation, and eventually added as new reusable assets in the EFCO tool. Thus, a process of design space exploration based on reusable assets was possible by a joint workflow of the EFCO tool and CoFluent.

The EFCO tool was implemented as a customization of the Coral Modeling Framework. The tool environment consists of a DSML for managing projects and design assets, diagram editors for various diagram types, form-based editors, a constraint evaluation engine for evaluating well-formedness rules and a set of integration tools for CoFluent Studio.

Importing CoFluent designs into the EFCO tool involves a conversion between the tool-specific CoFluent XML-format and the EFCO DSML and storage in a library as reusable assets, mainly complete EFCO UseCases composed of Features and Services. The EFCO tool included an editor which could be used to inspect and modify the lower level constructs which were used to compose the UseCase. In this case, new diagrams were created if needed for these components, to allow graphical editing of the detailed structures at the UseCase, Feature or Service level.

The EFCO library supports the reuse of assets at all levels, for example existing UseCases could be extended by adding new Features, or completely new Features can be composed from existing Services. The UseCases created or modified in the EFCO tool can then be stored as new reusable assets and exported to CoFluent for simulation.

The EFCO tool serves as an example of how third party tools can be integrated with a domain-specific tool environment implemented as a customization of the Coral Modeling Framework.

Bibliography

[1] CoFluent Design Homepage, available at http://www.cofluentdesign.com, visited May 9, 2011.

[2] Eclipse Model Development Tools. `http://www.eclipse.org/uml2`, visited May 9, 2011.

[3] The Eclipse Graphical Modeling Framework. Available at http://www.eclipse.org/gmf/, visited May 9, 2011.

[4] Aditya Agrawal. Metamodel based model transformation language to facilitate domain specific model driven architecture. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 118–119, New York, NY, USA, 2003. ACM.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, January 1986.

[6] Marcus Alanen. A Meta Object Facility-Based Model Repository With Version Capabilities, Optimistic Locking and Conflict Resolution. Master's Thesis in Computer Engineering, Department of Computer Science, Åbo Akademi University, Turku, Finland, November 2002.

[7] Marcus Alanen. *A Metamodeling Framework for Software Engineering*. PhD thesis, Åbo Akademi University, May 2007.

[8] Marcus Alanen and Ivan Porres. Model Interchange Using OMG Standards. In Bob Werner, editor, *Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications*, pages 450–458. IEEE Computer Society, Aug 2005. ISBN 0-7695-2431-1.

[9] Aitor Aldazabal, Terry Baily, Felix Nanclares, Andrey Sadovykh, Christian Hein, and Tom Ritter. Automated model driven development processes. In *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*, 2008.

[10] G. Arango. Domain analysis: from art form to engineering discipline. *SIGSOFT Software Engineering Notes*, 14(3):152–159, 1989.

[11] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*, pages 25–32, New York, NY, USA, 2008. ACM.

[12] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. The ATL Transformation-based Model Management Framework. Technical Report 03.08, University of Nantes, France, 2003.

[13] Dorothea Blostein and Andy Schürr. Computing with graphs and graph transformations. *Softw. Pract. Exper.*, 29(3):197–217, 1999.

[14] Marko Boger, Mario Jeckle, Stefan Mueller, and Jens Fransson. Diagram interchange for uml. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *«UML» 2002 — The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 371–378. Springer Berlin / Heidelberg, 2002.

[15] Eckard Bringmann and Andreas Krämer. Model-based testing of automotive systems. In *ICST*, pages 485–493. IEEE Computer Society, May 2008.

[16] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, August 2003.

[17] Jordi Cabot and Ernest Teniente. Constraint support in mda tools: A survey. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin / Heidelberg, 2006.

[18] Jordi Cabot and Ernest Teniente. Incremental evaluation of ocl constraints. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, 2006.

[19] Jean-Paul Calvez. *Embedded Real-Time Systems. A Specification and Design Methodology*. John Wiley and Sons, 1993.

[20] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[21] J. Craig Cleaveland. Building application generators. *IEEE Software*, 5:25–33, 1988.

[22] Thomas Connolly. *Database Systems*. Oxford University Press, Oxford Oxfordshire, 1999.

[23] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proceedings of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations. Italy*, pages 149–159, 2001.

[24] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[25] Microsoft Corporation. Microsoft Domain-Specific Language Tools. Available at http://msdn.microsoft.com/vstudio/DSLTools/, visited May 9, 2011.

[26] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[27] David H. Akehurst and Stuart Kent and Octavian Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.

[28] J. de Lara and H. Vangheluwe. Using Meta-Modelling and Graph Grammars to Process GPSS Models. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.

[29] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[30] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64(2):249 – 283, 2002.

[31] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 1 edition, March 2006.

[32] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inf.*, 74(1):31–61, 2006.

[33] Johannes Eriksson. *Tool-Supported Invariant-Based Programming*. Ph.d. thesis, Turku Centre for Computer Science, Finland, 2010.

[34] Niklas Fors. Efficient combination of reusable components in embedded system design. Master's thesis, Åbo Akademi University, Faculty of Technology, 2008. http://research.it.abo.fi/research/ese/projects/efco/fors.pdf.

[35] Christian Fuss and Verena Tuttlies. Simulating set-valued transformations with algorithmic graph transformation languages. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 442–455. Springer Berlin / Heidelberg, 2008.

[36] Leif Geiger and Albert Zündorf. Statechart modeling with fujaba. *Electronic Notes in Theoretical Computer Science*, 127(1):37 – 49, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[37] Gentleware. The Poseidon for UML product. `http://www.gentleware.com/`, visited May 9, 2011.

[38] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[39] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[40] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[41] Jan Hendrik Hausmann and Stuart Kent. Visualizing model mappings in UML. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 169–178, New York, NY, USA, 2003. ACM Press.

[42] Rational IBM. Rational Rose Modeler. `http://www.ibm.com/software/awdtools/developer/rose/modeler/`, visited May 9, 2011.

[43] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, 1975.

[44] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.

[45] Audris Kalnins, Janis Barzdins, and Edgars Celms. Basics of Model Transformation Language MOLA. In *Workshop on Model Transformation and Execution in the Context of MDA (ECOOP 2004)*, June 2004.

[46] Gabor Karsai, Andras Lang, and Sandeep Neema. Design patterns for open tool integration. *Journal of Software and System Modeling*, 4(2):157–170, May 2005.

[47] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[48] Stuart Kent. Model Driven Engineering. In *Proc. of IFM International Formal Methods 2002*, volume 2335 of *LNCS*. Springer-Verlag, 2002.

[49] Heiko Kern. (Meta)Model Interchange between MetaEdit+ and Eclipse EMF using M3-Level-based Bridges. In *The 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 14–19, October 2008.

[50] Donald E. Knuth. The new versions of TEXand MF. *TUGboat*, 10(3):325–328, November 1989.

[51] J.E. Kottemann and B.R. Konsynski. Dynamic metasystems for information systems development. In *Proceedings of the 5th International Conference on Information Systems*, pages 187–204. ACM Press, 1984.

[52] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. volume 110, pages 366–390, Duluth, MN, USA, May 1994. Academic Press, Inc.

[53] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[54] Anne-Thérèse Körtgen. Modeling successively connected repetitive subgraphs. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 426–441. Springer Berlin / Heidelberg, 2008.

[55] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, May 2001.

[56] Tomas Lillqvist. Subgraph Matching in Model Driven Engineering. Master's Thesis in Computer Science, Department of Information Technologies, Åbo Akademi University, Turku, Finland, March 2006.

[57] Björn Lundell, Brian Lings, Anna Persson, and Anders Mattsson. Uml model interchange in heterogeneous tool environments: An analysis of adoptions of xmi 2. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 619–630. Springer Berlin / Heidelberg, 2006.

[58] Torbjörn Lundkvist. Diagram Reconciliation and Interchange in a Modeling Tool. Master's Thesis in Computer Science, Department of Computer Science, Åbo Akademi University, Turku, Finland, November 2005.

[59] MagicDraw. MagicDraw UML. `http://www.magicdraw.com/`, visited May 9, 2011.

[60] Kim Marriott and Bernd Meyer, editors. *Visual language theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[61] James Martin. *Application Development without Programmers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1982.

[62] Jacqueline McQuillan and James Power. On the application of software metrics to uml models. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 217–226. Springer Berlin / Heidelberg, 2007.

[63] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[64] Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):69–80, November 2002.

[65] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin / Heidelberg, 2006.

[66] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM.

[67] Matias Nyman. A Model-Based Approach to Text Generation from Software Models. Master's Thesis in Computer Science, Department of Information Technologies, Åbo Akademi University, Turku, Finland, May 2006.

[68] Object Management Group website. `http://www.omg.org/`.

[69] OMG. UML Profile for System-on-Chip (SoC) Specification. Version 1.0.1 formal/06-08-01. Available at www.uml.org, 2006.

[70] OMG. OMG Model Driven Architecture, July 2001. Document ormsc/2001-07-01. Available at `http://www.omg.org/`.

[71] OMG. OMG Unified Modeling Language Specification, version 1.4, September 2001. Available at `http://www.omg.org/`.

[72] OMG. Common Warehouse Metamodel, version 1.1, March 2003. Document formal/2003-03-02, available at `http://www.omg.org/`.

[73] OMG. UML 2.0 Infrastructure Specification, September 2003. Document ptc/03-09-15, available at `http://www.omg.org/`.

[74] OMG. UML 2.0 Superstructure Specification, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.

[75] OMG. MOF 2.0 Query / View / Transformation Final Adopted Specification, November 2005. OMG Document ptc/05-11-01, available at `http://www.omg.org/`.

[76] OMG. XML Metadata Interchange (XMI) Specification, version 2.1, September 2005. Document formal/05-09-01, available at `http://www.omg.org/`.

[77] OMG. Meta Object Facility (MOF) Core Specification, version 2.0, January 2006. Document formal/06-01-01, available at `http://www.omg.org/`.

[78] OMG. Model View to Diagram Request for Proposal, November 2006. Document ad/06-11-07, available at `http://www.omg.org/`.

[79] OMG. Object Constraint Language, version 2.0, May 2006. Document formal/2006-05-10, available at `http://www.omg.org/`.

[80] OMG. UML 2.0 Diagram Interchange, version 1.0, April 2006. OMG document formal/06-04-04. Available at `http://www.omg.org`.

[81] OMG. MOF 2.0 Query / View / Transformation V1.0 Specification, April 2008. OMG Document formal/08-04-03, available at `http://www.omg.org/`.

[82] OMG Architecture Board. Model Driven Architecture—A Technical Perspective, 2001. Document ormsc/01-07-01, available at `http://www.omg.org/`.

[83] Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.

[84] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 107–121, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[86] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 238–251, London, UK, 2000. Springer-Verlag.

[87] James Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[88] István Ráth, András Ökrös, and Dániel Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software and Systems Modeling*, 9:453–471, 2010. 10.1007/s10270-009-0122-7.

[89] Peter H. Salus. *Little Languages and Tools*. Macmillan Technical Publishing, 1998.

[90] Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, pages 487–550, 1999.

[91] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

[92] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[93] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[94] Dániel Varró. Automatic Program Generation for and by Model Transformation Systems. In Hans-Jörg Kreowski and Peter Knirsch, editors, *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pages 161–173, Grenoble, France, April 12–13 2002.

[95] Gergely Varró, Ákos Horváth, and Dániel Varró. Recursive graph pattern matching. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12,*

2007, Revised Selected and Invited Papers, volume 5088 of *Lecture Notes in Computer Science*, pages 456–470. Springer, 2007.

[96] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

[97] Ingo Weisemöller and Andy Schürr. A comparison of standard compliant ways to define domain specific languages. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 47–58. Springer Berlin / Heidelberg, 2008.

[98] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[99] Andreas Winter et al. The Graph Exchange Language website. `http://www.gupro.de/GXL/`, visited May 9, 2011.

[100] Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

[101] Albert Zündorf. Graph pattern matching in PROGRES. In *Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science*, pages 454–468, London, UK, 1996. Springer-Verlag.