MŰEGYETEM 1782

Budapest University of Technology and Economics
Department of Measurement and Information Systems

Declarative Specification of Domain Specific Visual Languages

Master's Thesis

István Ráth

Supervisor:

Dr. Dániel Varró
assistant professor

Budapest, 19 May 2006

I would like to thank Dr. Dániel Varró for his continued support, friendly advice, and enthusiasm. I am also very thankful to my friends and colleagues, Dávid Vágó and András Schmidt, without whose work my research would have been impossible. I would also like to thank all other colleagues and friends who participated in the VIATRA2 *project by providing valuable testing and feedback.*

Nyilatkozat

Alulírott Ráth István, a Budapesti Műszaki és Gazdaságtudományi Egyetem műszaki informatika szakos hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem.

Ráth István

Összefoglaló

Napjainkban a modell bázisú szoftverfejlesztési paradigma az iparban széles körben elfogadott módszertan. Ez az elképzelés a fejlesztési folyamatot egy precíz modellezési lépéssel kezdi, általában UML alapokon. Az alkalmazás forráskódját ezekből a modellekből származtatjuk, automatikus kódgeneráló eljárások segítségével.

Az ipari tapasztalatok azt mutatják, hogy az UML általános elemkészlete nem minden esetben alkalmas az alkalmazási célterület (domain) speciális igényeinek maradéktalan kielégítésére. Továbbá, sok esetben fontos, hogy a rendszert egyszere több nézőpontból tervezhessük, minden aspektusban a legmegfelelőbb modellezési nyelv használatával. Az UML új, 2.0-ás változata is ezt a filozófiát követi: minden diagramtípus felfogható egy önálló grafikus domain specifikus modellezési nyelvként, emellett további nyelvekkel is kiegészíthetjük az alapkészletet. A jelenlegi modellezési eszközök (pl. az Eclipse projekt részeiként elérhető EMF és GEF) csupán alapvető támogatást nyújtanak az új nyelveket támogató eszközök készítéséhez, ezért egy-egy új grafikus nyelv kifejlesztése drága és lassú, valamint a már elkészített elemek újrahasznosítása gyakran problémákba ütközik.

A domain specifikus modellezési nyelvek alkalmazásának legfontosabb aspektusai a következők: (i) az egyes nyelvek közötti átjárás biztosítása (fordítás), valamint (ii) az egyes részmodellekből a globális és koherens rendszermodell származtatása. Mindkét kihívásra kézenfekvő válasz a modelltranszformációs technológia alkalmazása, azonban az elérhető eszközök még a fejlesztés korai fázisában vannak. A jelenlegi modellezési keretrendszerek további, közös hiányossága a grafikus megjelenítés absztrakt szintakszishoz kötése. Bár az absztrakt és konkrét szintakszis a nyelv két különböző metaszintjét képviselik, az elérhető eszközök esetében a diagramok csupán az absztrakt modellek (felhasználó által meghatározott) részeit jelenítik meg. Ez egyszerű nyelvek esetén elfogadható megoldás, azonban összetett rendszermodellezésnél a bonyolultság kezelhetetlenné válhat. Ezért szükséges az absztrakt és konkrét szintakszis minél széleskörűbb szétválasztása.

A diplomatervben bemutatom a ViatraDSM keretrendszert, amelyet Vágó Dáviddal közösen terveztünk és fejlesztettünk. A rendszer formális és egységes támogatást nyújt a domain specifikus modellezési nyelvek szerkesztőinek modell bázisú tervezéséhez, beleértve a szimulációs és kódgenerátor funkciókat is. Legfontosabb tervezési célkitűzésünk a minél széleskörűbb újrafelhasználhatóság, a több nézőpontú tervezés és nyelvközi transzformációk támogatása volt.

Önálló munkám legfontosabb eredményei a következők:

- kiterjesztettem a ViatraDSM rendszer többnézőpontú modellezést támogató komponenseit, így a rendszer támogatja több modellezési aspektus integrálását metamodell-címkézési és transzformációs technikákkal;
- kifejlesztettem a keretrendszer megjelenítési funkcióit, amelyek támogatják az absztrakt és konkrét szintakszis metamodell-szintű szétválasztását, a két réteg közötti kétirányú megfeleltetés segítségével.

Az önálló labor és a diplomatervezés során végzett munkánk eredményeképpen a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztett VIATRA2 modelltranszformációs keretrendszer 2005 szeptemberétől hivatalosan az Eclipse Generative Model Transformers projekt része.

Abstract

Nowadays, the model based development paradigm has gained considerable acceptance within the software development industry. This paradigm begins the development process with a precise modeling step, usually based on UML. Application source code is generated from these models using automated code generation technology.

Industrial experience with model driven development shows that UML's general concepts do not always fulfill the special requirements of the application's target domain. Moreover, in many cases it is practical to design a system using multiple perspectives, with the most appropriate domain specific language being used for each modeling perspective. Even UML 2.0 follows this philosophy: each of its diagrams can be considered a (graphical) domain specific language (DSL), but it can also be extended to support new languages for modeling in a special domain. Present tools (e.g. Eclipse's EMF and GEF) only provide basic support for creating DSL tools, thus the development of complex and visual domain specific languages is expensive, and existing solutions are difficult to reuse.

The most critical aspects of utilizing domain specific language technology are: (i) providing support for translation between DSLs, and (ii) generating a global and coherent system model from small domain specific submodels. Both of these problems should typically be solved using automated model transformations; existing tools, however, have not yet progressed beyond development prototype status.

A common problem of present day tools is a simplistic approach to graphical representation. While abstract and concrete syntaxes represent different levels of abstraction, current implementations do not allow the language engineer to customise the visualisation of modeling languages independently of the internal representation. This is acceptable for simple domains, however for complex systems engineering a more flexible approach is necessary.

In this thesis, I present the ViatraDSM framework, designed in co-operation with Dávid Vágó, which provides uniform and formal support for creating editors, model transformations, simulators and code generators for domain specific visual languages. This framework was designed to ensure reusability, and support multi-domain modeling and inter-domain translations.

My most important results are the following:

- I extended the capabilities of the ViatraDSM framework to support the multi-domain modeling of complex systems using light-weight modeling and model transformation approaches;
- I developed the presentation layer of the ViatraDSM framework to support the metamodel-level separation of the abstract and concrete syntax modeling layers, facilitating a bi-directional mapping based on a metamodel specification.

Based on our efforts, the VIATRA2 system, developed at BUTE's Department of Measurement and Information Systems, officially became part of the Eclipse Generative Model Transformers subproject as of September, 2005.

Contents

Chapter 1

Introduction

Models are projections of reality. In engineering, they are used to capture concepts related to the engineer's task: the product that is being designed, its environment, and processes that may be involved in its life cycle. Models are attractive because they allow a problem to be precisely grasped at the right level of abstraction without unnecessarily delving into detail.

1.1 Models in software engineering

In software development, models are used to represent data structures, communication between agents, algorithms, or at a higher abstraction level, entities in a complex system, e.g. organisational units or products in an e-commerce application.

The software development industry has been using high level models at the core of development for more than a decade now, because it is widely believed that software systems of large complexity can only be designed and maintained if the level of abstraction is set considerably higher than that of conventional programming languages. On the other hand, computers can only operate at the lowest possible level of abstraction (machine code consisting of elementary operations and data primitives), thus models need to be translated into the language that the target *platform* can understand and run. By platform I mean the low level software and hardware architecture that executes application code (which includes software libraries, operating systems, a computer architecture, or even a virtual machine with a runtime framework, such as Sun's Java or Microsoft's .NET).

For low level models, this translation is usually called *compilation* (in this context, a programming language construct, i.e. source code is also considered a model); while for high level models, the term *model transformation* is frequently used. This is traditionally called *model-to-model transformation*. In contrast, a special case of model transformation is referred to as *code generation*, where source code is generated from a (graphical) model, using a code generator (*model-to-code transformation*).

The model based software development paradigm is based on the idea that the developer should work with high abstraction level models during most of the development cycle. For this to work in practice, model transformations are required. Source code, the 'traditional' product of software development, should be generated to the largest possible extent, to minimize the amount of business logic that is outside of the scope of modeling, and is only represented by handwritten code.

1.2 The evolution of approaches

1.2.1 CASE

Whilst all aspects of the software development process could be supported by software tools, computer aided software engineering (CASE) tools are usually used to assist software design and analysis. Historically, CASE emerged during the 1980's, when many software development companies realised that meeting increased demand for high quality and complex software required more sophisticated development methods than those used previously. These tools arose out of developments such as Jackson Structured Programming and the software modelling techniques promoted by researchers like Ed Yourdon, Chris Gane and Trish Sarson (SSADM: Structured Systems Analysis and Design Methodology). CASE is a very broad concept, and in that sense even modern integrated develoment environments, such as Eclipse, or Visual Studio can be considered CASE tools. In this historical context, however, the term 'CASE tool' refers to the earliest programs designed to assist software development and analysis.

The problem with early CASE tools stems from the fact that they lacked a common approach. Although the methodologies they were built to support shared similar concepts, on the implementation level they differed substantially. Due to the lack of a common graphical notation system, even development documentation was hardly reusable.

1.2.2 UML

The Unified Modeling Language was conceived to provide a common framework for specification, modeling, and documentation in the software development process. In many senses, it was a spectacular success, because it established a *visual*, easy to use notation system which was comprehensive enough to capture all major aspects of software engineering. Today, UML is *the* industry standard for modeling and specification. It's use is not restricted to modeling software, it is also widely used for business process modeling, organizational structure modeling, and even hardware design. UML represents a compilation of best engineering practices which have proven to be successful in modeling large, complex systems, especially at the architectural level.

History UML was developed by Grady Booch, James Rumbaugh, and Ivar Jacobson. It was first standardized in 1997 under the supervision of the Object Management Group, a consortium, including industry heavyweights such as IBM, Hewlett-Packard, Apple Computer, and Sun Microsystems, formed in 1989 to set standards in object-oriented programming and system modeling. Along with the standardization of UML, the OMG's most important work is CORBA (Common Object Request Broker Architecture), an architecture designed to enable applications on heterogenous platforms to interoperate using a common set of application programming interfaces (APIs), communication protocols and information models. CORBA was a very ambitious project, but it is a debated issue whether it can be considered successful. However, it fits well into the pattern of technologies promoted by the OMG: all are fairly large and complicated, of *in-width* nature, trying to achieve very ambitious goals by being "everything to everyone".

Aspects UML handles three domains of system modeling: requirement, static, and dynamic models. Requirement modeling employs concepts which are related to how the system interacts with its surroundings (Use Case Diagrams). Static modeling deals with the structure of the system, and uses concepts such as classes, objects, attributes, operations, and associations (Class and Deployment Diagrams). Dynamic modeling captures the behaviour of the system with concepts like activities, messages, function calls, states, concurrency, transitions (Activity Diagrams, Sequence Diagrams, State Chart Diagrams).

Apparently, UML is intended to be a general-purpose modeling language, which is independent of the application domain - even if some of the diagrams evolved from languages already being used before in certain domains (e.g. statecharts in embedded systems).

1.2.3 Model Driven Architecture

The Model Driven Architecture (MDA) is OMG's newest approach to model-based software development. MDA is essentially an approach to model-based software development utilizing OMG's flagship techonogies, UML, the Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM).

MDA is a visionary concept: it is a model-based software development paradigm to support evolving standards in application domains as diverse as enterprise resource planning, air traffic control and human genome research; standars that are tailored to the need of these diverse organizations, yet need to survive changes in technology and the proliferation of different kind of middleware. The OMG Model Driven Architecture addresses the complete life cycle of designing, deploying, integrating, and managing applications as well as data using open standards. MDA-based standards enable organizations to integrate whatever they already have in place with whatever they build today and whatever they build tomorrow[45].

Design goals MDA was designed with the following goals in mind:

- *Portability and reusability*, increasing application reuse and reducing the cost and complexity of application development and management.
- *Cross-plaform interoperability*, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions.
- *Platform independence*, greatly reducing the time, cost and complexity associated with retargeting applications for different platforms.
- *Domain specificity*, through domain-specific models that enable rapid implementation of new, industry-specific applications over diverse platforms (the term *domain* will be explained in detail in 1.4).
- *Productivity*, by allowing developers, desingers and system administrators to use languages and concepts they are comfortable with, while allowing seamless communication and integration across the teams. Moreover, a significant reduce in costs is attained by models of the target application that can be directly tested and simulated.

Development steps As it can be seen on Figure 1.1, MDA emphasizes the clear distinction between Platform Independent Models (PIM) and Platform Specific Models (PSM), thus, software development in MDA is envisioned as a three-step process. First, the Plaform Independent Model is designed, which is supposed to use modeling concepts which are not platform specific. The PIM is a pure UML model, with constraints specified in the Object Constraint Language (OCL), and behavioral semantics described in Action Semantics (AS) language.

The second step is to generate a Plaform Specific Model, which contains additional UML models, and represents an implementation of the system under design which can run on the target platform. The transition between PIM and PSM should typically be facilitated using automated

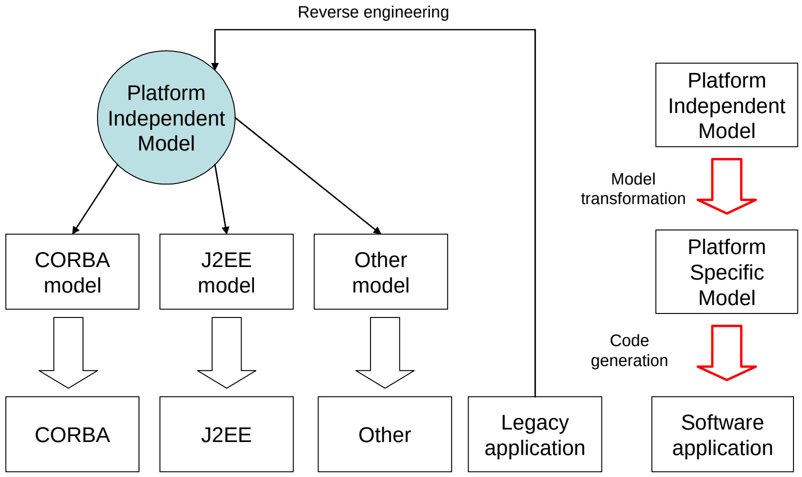


Figure 1.1: Model Driven Architecture

model transformation technology. The most important keyword of this phase is ”standard mappings”, i.e. it is very important that this transformation step be *agile*, meaning that it should require the smallest possible amount of human interaction (otherwise, there is no point in wasting lots of time on platform independent designs).

Finally, application code is generated from the Platform Specific Model. Again, code generation should be as extensive as possible, in order to minimise the amount of necessarily slow and error-prone manual coding. This, in turn, requires PSMs that are expressive enough, not only from a static, but also from a dynamic point of view of the system, to produce all of the application code.

1.3 Problems with MDA

1.3.1 Domains in MDA

In MDA, initial system design was carried out on the first, plaform independent level. ”Domain knowledge” appears here as UML profiles, or applied in-house design patterns (general, best-practice solutions to common domain-specific problems). The platform specific model was automatically generated using a standard mapping. As PIM is one level of abstraction higher than the PSM, for the PSM to be as comprehensive as possible, all necessary information should be provided in these standard mappings. This information, in MDA terms, is *platform-specific*, rather than *domain-specific*. Therefore, as MDA is based on UML, the success of an MDA design highly depends on how expressively a domain can be modeled using a general-purpose modeling language, on a platform-independent level.

From the late 1990’s, UML rapidly gained industry-wide acceptance, many software houses began to experiment with using UML not only for documentation, but for real model-driven software development. New technologies, such as extensive code generation and model verification were found to be difficult to implement with UML models. At the end of the decade, several shortcomings in UML have been pinpointed, in surveys such as [24]. The most important weaknesses of UML are identified in [46] as:

> *...its imprecise semantics, and the lack of flexibility in domain specific applications. In principle, due to its in-width nature, UML would supply the user with every construct*

> *needed for modeling software applications. However, this leads to a complex and hard-to-implement UML language, and since everything cannot be included in UML in practice, it also leads to local standards (profiles) for certain domains.*

Thus, the two key points, where UML needed improvement, were: (i) precise semantics, and (ii) flexibility to integrate domain-specific concepts.

The Object Management Group has partially succeded in identifying these problem areas, as considerable efforts have been made towards an *in-depth* evolution of the UML standard for version 2.0. According to these ideas, UML 2.0 would consist of a core kernel language (UML Infrastructure 2.0), and an extensible family of distinct languages (UML Superstructure). Each UML sublanguage would have its own (individually defined) semantics, which fundamentally requires an appropriate and precise metamodeling technique.

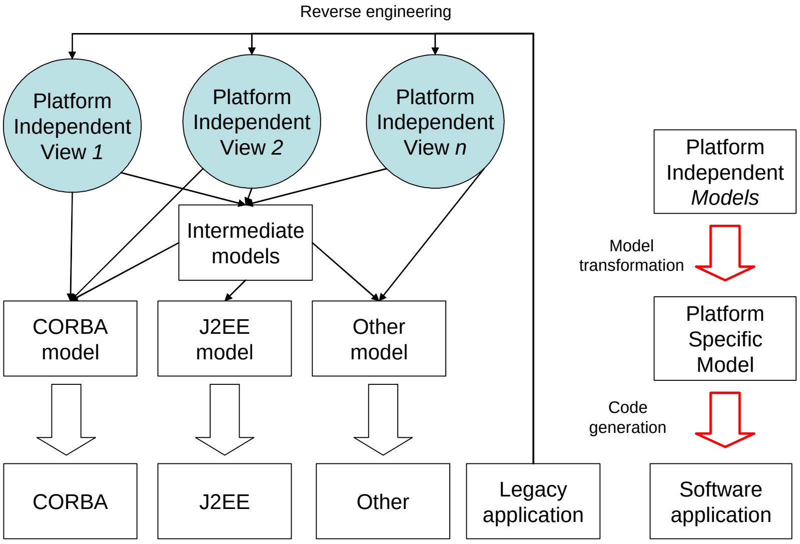


Figure 1.2: Model Driven Architecture - in reality

1.3.2 Transformations in MDA

Such a metamodeling-based architecture of UML highly relies on transformations within and between different models and languages. In practice, transformations are necessitated for at least the following purposes [46]:

- model transformations within a language should control the correctness of consecutive refinement steps during the evolution of the static structure of a model, or define a (rule-based) operational semantics directly on models;
- model transformations between different languages should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design;
- a visual UML diagram (i.e., a sentence of a language in the UML family) should be transformed into its (individually defined) semantic domain, which process is called model interpretation (or denotational semantics).

The crucial role of model transformation (MT) languages and tools for the overall success of model-driven system development have been revealed in many surveys and papers during the recent years. To provide a standardized support for capturing queries, views and transformations between modeling languages defined by their standard MOF metamodels, the Object Management Group is soon to issue the QVT standard.

QVT provides an intuitive, pattern-based, bidirectional model transformation language, which is especially useful for synchronization kind of transformations between semantically equivalent modeling languages. The most typical example is to keep UML models and target database models (or UML models and application code) synchronized during model evolution in a bidirectional way.

However, there is a large set of model transformations, which are unidirectional by nature, especially, when the source and target models represent information on very different abstraction levels (i.e. the model transformation is either *refinement* or *abstraction*). Unfortunately, the current QVT Mapping Language is far less intuitive and easy-to-use in case of unidirectional transformations.

The VIATRA2 model transformation framework primarily aims at designing model transformations to support the precise model-based system development with the help of invisible formal methods[4].

It is important to recognize that UML is intended to be the *one and only* language, a universal standard. There are, however, domains where engineers either do not understand UML, or the general concepts of UML are simply inappropriate for modeling effectively - in fact, they might already have their own standard languages or tools which are.

1.4 Domain-specific modeling

What makes a model domain specific? A *domain* can be defined as the set of concepts and their relations within a specialized problem field. This definition implies that all software is built to solve domain-specific problems, since software engineering is all about providing software solutions to problems in specialized problem fields (e.g. pharmaceutical, business processes, civil engineering). Even software design is a domain in this sense. On the other hand, in software design, the term *application domain* refers to a knowledge base that is outside of the scope of software development. For example, if an application for managing a book shop is being designed, all information that describes actual business processes and products, how they interact, what their attributes are, etc. constitutes *domain knowledge*. Domain knowledge can only be obtained from domain experts, and it is therefore one of the most valuable assets of software development.

Domain-specific (programming) languages (DSLs) are specialised languages, suitable for efficiently writing programs to solve problems that are specific to an application domain. In contrast to general-purpose programming languages, DSLs are much more expressive and comprehensive for the experts in their own domain. For example, when using SQL, it is very easy to write database queries, but it is impossible to implement an operating system kernel. In the C language, it is possible to do both, but a single line of SQL code is likely to correspond to tens, even hundreds of lines of C code.

Intentional Programming (IP) by Charles Simonyi [41], shares similar concepts with domain-specific programming languages. However, IP is more like a bottom-up approach, in contrast to the top-down nature of DSLs, where the languages are specifically tailored to the needs of the target domain. IP was envisioned to be an evolution of general-purpose programming languages.

In IP, the level of abstraction is raised by introducing to notion of *intention*, the abstract concept behind common programming language constructs such as iteration, recursion, variable, function calls etc. Intentional programmers create source code by glueing together their intentions into a coherent logical structure which represents the functionality of the system under design. This approach combines the flexibility of textual languages with the visual and abstract nature of model-based development.

Aspect-oriented Programming (AOP) by Gregor Kiczales et al[21], was also meant to be an evolution of conventional procedural and object-oriented programming paradigms. However, instead of raising the level of abstraction above the source code level, AOP is a horizontal approach. In AOP, the most important goal was the *separation of concerns* by splitting the logical structure of application source code into *aspects*, which are later joined at the designated *join points* by an automated process called *weaving*.

Note that both IP and AOP are targeted towards programmers, not domain experts, which is a crucial difference.

Domain-specific modeling languages Analogously to domain-specific textual languages, domain-specific modeling languages are modeling languages which operate with elements and rules that are special to the target domain. For example, if one wants to design user interfaces for mobile phones, domain-specific model elements could include *Menu*, *MenuItem*, *DialPad*, *SMS* etc. Domain-specific modeling languages are, just as DSLs, much more expressive in their domain, than general-purpose modeling languages, such as the Unified Modeling Language (UML).

Domain-specific modeling (DSM) is a new approach to model-based software development. In contrast to IP and AOP, DSM is a top-down and vertical approach: instead of trying to create high abstraction level ”interfaces” to source code, DSM gives the designer the freedom to use structures and logic that is specific to the target application domain, and thus, completely independent of programming language concepts and syntax. Similarly to MDA, DSM is a model-based approach, however, while MDA emphasizes the importance of a single and universal modeling language at the center of the development process, proponents of domain-specific modeling argue that flexibility and ease of use by domain experts is more important than sticking to (pure) UML.

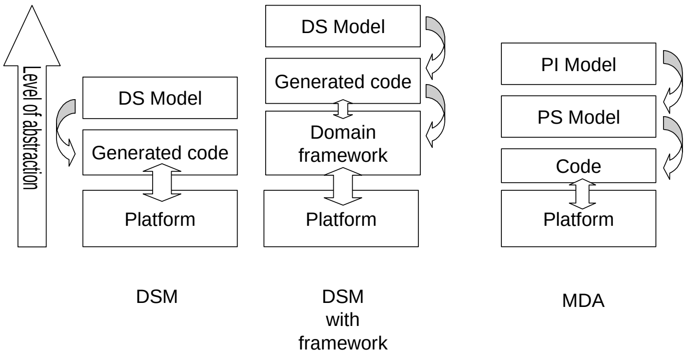


Figure 1.3: The difference between various DSM and MDA appoaches

As it can be seen on Figure 1.3, in current domain-specific modeling technology, source code is generated directly from domain-specific models, instead of going through the platform-independent and platform-specific abstraction levels found in MDA. While it is often claimed that it is exactly this difference which makes 100% code generation possible [19], this approach can make the implementation of platform-specific code generators difficult. Thus, recent DSM approaches, such as the one presented in [38], emphasize the importance of a *domain framework*, which makes code generation easier by ensuring portability across various platforms (thus, this domain framework has to be implemented on various architectures, similarly to the Java or .NET idea).

1.4.1 Motivation for DSM

If the model-based software development paradigm is to succeed on a wider scale, effective methods for rapid, productive and agile modeling and code generation need to be established.

As it has been previously concluded, UML is a general-purpose modeling language, and as such, it is rather limited in integrating domain-specific modeling concepts. Recent articles, such as [14], see domain-specific modeling languages as *"...the next step towards developing a technology for software manufacturing"*. Others, like [19], claim that a pure DSM-based approach consistently results in productivity increases of 500-1000%, compared to the mere 35% found with MDA (interestingly, all the articles refer to one and the same study[6]). Steven Kelly, the Chief Technical Officer of MetaCase, a company that produces a popular DSM framework, recently wrote[20]:

> *While good, that* [35% increase in productivity] *is far from the 500%-1000% consistently found with DSM. MDA proponents envisage higher forms of MDA incorporating elements of DSM, and these may offer some more hope. In these, the base UML can be extended with domain-specific enhancements, or even replaced with new MOF-based metamodels. However, experiences with the former have found current tools lacking the necessary extensibility, and no tools support the latter.*

1.5 MDSE = MDA + DSM

From the author's perspective, MDA and DSM are complementary concepts rather than rivals. Both emphasize the importance of reusable and verifiable design based on models of a high abstraction level. What OMG seems to fail to realise with MDA is that in order to integrate development from the widest possible range of application domains into a single framework, the standard needs to be as flexible as possible on the highest abstraction level (PIM), while being as precise as possible concerning the automated transformations between the abstraction levels. If these two requirements are met, it does not really matter what modeling language is used for documentation (UML was a good choice in this respect).

The idea behind the results presented in this thesis is a synthesis of the Model Driven Architecture and Domain-specific Modeling approaches based on a robust model transformation framework: **Multi-Domain Systems Engineering (MDSE)** can be considered an MDA variant, but also a DSM variant. As it can be seen on Figure 1.4, our approach recognizes (i) providing support for translation between domain-specific modeling languages, and (ii) generating a global and coherent system model from small domain specific submodels as the most critical aspects of utilizing DSM technology in MDA.

A domain-specific modeling environment is only effective if its visual languages are sufficiently expressive, intuitive and easy-to-use. This is only possible if the system provides support to all aspects of language engineering:

- *Concrete syntax*, so that designers can use familiar visual symbols;

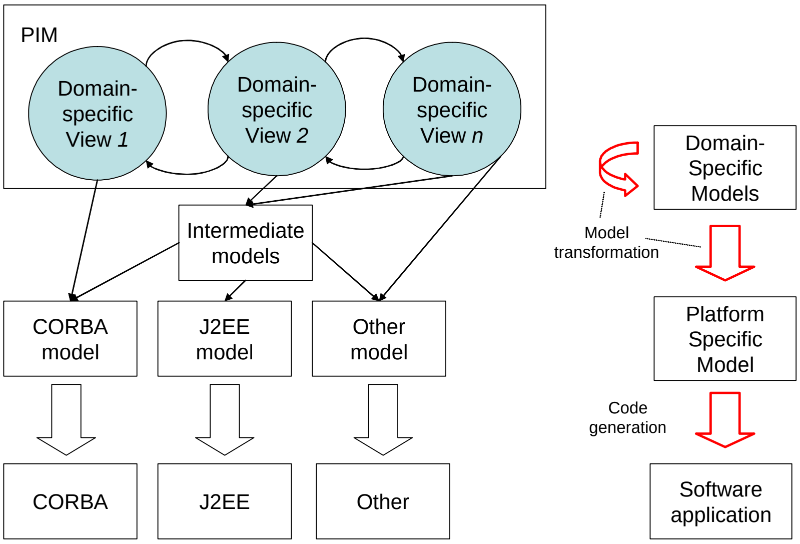


Figure 1.4: Model Driven Systems Engineering

- *Abstract syntax*, to make efficient and precise model transformations possible;
- *Well-formedness rules*, to integrate domain-specific constraints into high-level models;
- *Dynamic semantics (simulation)*, to enable the designer to visualize the system as it interacts with its environments;
- *Transformations*, to facilitate the automated translation of high-level models into a low-level representation (PSMs, or application source code).

Additionally, it is important to emphasize that abstract and concrete syntax represent different levels of abstraction. Therefore, a *conceptual separation* of these modeling layers is required to ensure that abstract syntax can be optimised to the requirements of the modeling infrastructure (e.g. low memory footprint, suitable structure for transformations, etc), without compromising the end user's ability to work with perspicuous and familiar-looking visual models of manageable complexity.

1.6 ViatraDSM: a tool supporting MDSE

The ViatraDSM framework, a tool developed by Dávid Vágó and myself, is a general and flexible domain-specific modeling and transformation environment, leveraging the model transformation facilities of the VIATRA2 framework. Our approach enables the handling of all critical aspects of language engineering (abstract syntax, concrete syntax, well-formedness constraints, model simulation and transformations) within a single framework.

Our approach integrates a mathematically precise model transformation engine with an easy-to-use domain-specific modeling interface, in order to:

(i) enable language engineers to design domain-specific languages utilising all aspects of language engineering;

(ii) provide support for precise model-to-model and model-to-code transformations, using a powerful domain-specific programming language;

(iii) integrate the system into an existing model-driven development infrastructure through import and export capabilities;

(iv) so that domain experts can use an intuitive graphical user interface to create multi-domain models with support for interdomain translations and simulation, seamlessly integrated into a model transformation framework; thereby facilitating the automatic generation of documentation, low-level models, and application source code from high abstraction level models.

ViatraDSM was first introduced in a report by Dávid Vágó, András Schmidt and myself, titled *Automated Model Transformations in Domain Specific Visual Languages*, which won first prize at the Scientific Students' Association conference (”Tudományos Diákköri Konferencia” in Hungarian) in October 2005. Since then, the tool has matured: some of the theoretical background has been revisited and new ideas and features were introduced.

1.7 Objectives

In the last semester, development work on both ViatraDSM continued; my research goals – and the primary contributions of this thesis – were to develop the multi-domain modeling capabilities and design and implement the presentation layer supporting the complete separation of abstract and concrete syntaxes.

In this thesis, I present the following results:

- I evaluate the state-of-the-art of language engineering tools by comparing various academic and commercial domain-specific visual environments (Chapter 2);
- I describe the theoretical background and goals of the ViatraDSM framework (Section 2.7) as developed by myself, including:
  - a method for the precise specification of domain-specific visual languages using a high-level formal language;
  - a proposal for the definition of the abstract syntax of these languages using visual and precise metamodeling techniques;
  - techniques to uniformly specify modeling constraints, model transformations, model simulation and code generation by a combination of graph transformation and abstract state machines.
- I overview the VIATRA2 framework, which provides the model transformation infrastructure for the ViatraDSM framework, including the modeling basics and the native textual languages (Chapter 3);
- I discuss the design and implementation details of the ViatraDSM framework (Chapter 4), developed in co-operation with Dávid Vágó;
- I describe the fundamentals for multi-domain modeling based on multiple aspects (designed by myself), and discuss several techniques implemented in the ViatraDSM framework, by Dávid and myself (Chapter 5);

- I propose multiple imperative and declarative techniques to facilitate the mapping between the abstract syntax and concrete syntax model layers of visual languages (Chapter 6):
    - I analyse the importance of the conceptual separation of abstract and concrete syntax on various levels of abstraction (Sec. 6.2);
    - I discuss my design targeting the complete, yet manageable, metamodel-level separation of abstract and concrete syntax representation as implemented in the ViatraDSM framework (Sec. 6.3);
    - I describe three different, but interoperable approaches and their implementation details (Sections 6.3.2, 6.3.3 and 6.3.6).
- I demonstrate the feasibility of our approach with the case study of a simple, but descriptive example (domain-specific Petri net editor) (Chapter 7).

Chapter 2

The State of the Art of Language Engineering

2.1 Goals

Our approach, Multi-Domain Systems Engineering, intends to integrate the Model Driven Architecture and Domain-Specific Modeling development paradigms. The success of such an approach is highly dependent on two key aspects: (i) domain language engineering and (ii) domain integration.

2.1.1 Language engineering

With models, we can capture static attributes, as well as dynamic behaviour, i.e. it is not only possible to describe the structure, but how this structure changes as the represented system operates. To construct models, *modeling languages* are used. For textual languages, one can think of an alphabet and a grammar. In this thesis, the term *modeling language* refers to a visual (graphical) language. The rules and elements for modeling languages are defined by *metamodels*. These are also models, constructed using a *metamodeling language*. A metamodeling language is defined by the following features:

- **Concrete syntax**, which is a specification of all the visible features of a modeling language. In textual languages, complex expressions in concrete syntax may be faster to write in a compact form, but this also means that they can be difficult to read above a certain level of complexity. In contrast, visual languages are generally easier to read, and, more importantly, *safe* to write, because a good visual editor does not allow to create models with syntax errors. (Note however, that semantic mistakes are much harder to eliminate - DSM tools can be good at making such faults more apparent because these can be easier to detect on a higher abstraction level).

- **Abstract syntax** defines the vocabulary of language concepts and how these can be combined in models. The abstract syntax is also called the *language metamodel*. Apart from the definition of language concepts and their relationships, metamodels also contain information concerning taxonomy and ontology (abstraction and specialization). Metamodels are constructed using *core metamodeling languages*, which are one metalevel higher, and specify what concepts can be used for language specification. An example of a core metamodeling language is MOF.

- **Well-formedness rules** are constraints which must be satisfied by models. Typical examples are multiplicity constraints, aggregation (e.g. ”at most one parent for each model element”), or language/domain specific constraints. In UML, such well-formedness rules may be expressed as part of the model (multiplicity), or using a separate constraint description language (OCL).

- **Dynamic (operational) semantics**, in contrast to the previous three features, models the operational behaviour of language concepts. In design, *simulators* are just as important as static descriptions because they allow the modeler to view the system as it will effectively behave and interact with its surroundings, at a high level of abstraction.

- **Transformations (Denotational/Translational semantics)** specify how the abstract syntax can be translated into a semantic domain (e.g. programming language). This is important from a practical point of view, since models on their own are not very useful, they need to be transformed to a lower level of abstraction so that the platform can execute the represented system.

The special importance of transformations As it can be seen on Figure 1.4, our approach integrates domain-specific models as various views of the plaform independent model structure. It is also important to emphasize the possiblity of translations between these views, which makes it possible to design one and the same system from *different* (requirement) domains (*multi-domain modeling*). This is only possible if automated tool support is available to all five aspects of domain specific metamodeling and model transformation. Current technology, however, only covers the first three, with a separate code generator used for specifying denotational semantics. Simulation (operational semantics) support is limited, as UML's standardized solution, Action Semantics is not precise enough, many tool providers experiment with complicated Application Programming Interfaces, or other custom, non-intuitive techniques.

2.1.2 Domain integration

Some of present domain-specific modeling tools are custom solutions, not designed to be integrated into an existing model-driven development architecture. In many cases, existing metamodels or code generators are hard to be reused. Moreover, although most DSM providers criticize UML-based approaches, the reality is that UML is so widespead today that it simply cannot be ignored. Thus, along with domain-specific code generators/serializers, support for UML export is necessary as well.

2.2 Basis of comparison

As already discussed in 2.1, the usability of domain-specific visual languages in Model-Driven Systems Engineering depends on how well DSM can be integrated into development processes. This, in turn, stems from two key factors: (i) what aspects of language engineering can be used to customize the DSM layer, and (ii) how can the DSM tools be seamlessly integrated into the existing toolchain.

Although domain-specific modeling is not a new idea, a breakthrough such as UML's rapid adoption is still to come. However, there has been a consideable rise in interest as influential software development houses realised the potential of DSM.

In the next section of the thesis, I give a brief cross-sectional view of the current state of the art in DSM technology. I proceed roughly in a cronological order, comparing variouos approaches using the following evaluation system:

2.2.1 Language engineering criteria

1. **Concrete syntax**

   (a) Automated support for specifying the visual appearance of language elements, using **visual editors** or templates (pre-defined visual elements for common concepts can significantly speed up the creation of domain-specific editors.)

   (b) **Multi-domain visualization**: Support for visualization of model elements from different domains in a single view. This is important because in many cases complex systems need to be modeled in multiple domains, and it can be convenient to visualize the various aspects in a single diagram.
   An extended conceptualization of multi-domain modeling means that the same logical instance of a model element can have attributes in multiple domains, meaning that *multi-domain models* can be constructed and altered in multiple domain-specific editors. For a more detailed explanation see Sec. 5.2.

   (c) **Diagram modeling support**
   The most important aspect of diagram modeling, apart from the possibilities to describe visual appearance in graphical editors, is the conceptual separation of *logical models* and *diagrams*. As an example, in a typical UML modeling tool, the user sees a tree-view based representation of all model elements, possibly arranged by some logic (e.g. projects, solutions, or any kind of hierarchy). Diagrams are constructed by dragging these model elements onto a canvas, or by using creation tools and drawing on the canvas directly. However, the *user* decides what to visualize on a diagram: model elements can exist without being represented on a diagram at all.
   Diagram-model mapping means support for multiple concrete syntax representations, possibly using a custom mapping of model elements to diagram elements using some flexible rule definiton language. This means that instead of sticking to the "node → figure, edge → arrow" concept, the modeling software allows for more complex mappings, e.g. displaying aggregate information in special diagram elements. This technique requires a *metamodel-level* separation of the diagram (concrete syntax) and logical (abstract syntax) models, by introducing a separate visualisation layer, where diagrams are stored independently from logical models. However, since diagrams still represent a projection of the logical modelspace, the editor must impose certain rules that guarantee that there will be no inconsistency between models and their graphical representation. For more details, see Chaper 6.

2. **Abstract syntax**
   The core metamodeling approach of the modeling environment is important, since (i) it should be as concise as possible to support mathematically precise transformations; (ii) while being as flexible as possible to support integration of arbitrary (non-MOF/UML based) models.

3. **Well-formedness rules**

   (a) Static constraints: Modeling environments generally support simple static constraints such as multiplicity or type-correctness at relationship endpoints.

   (b) Language-specific constraints: These typically specify higher-level constraints such as *"every model element has a unique name"* etc. These constraints can be described using a constraint descripton language or a special model transformation from the modeling language to boolean values.

   (c) Enforcement: Well-formedness constraints can be either enforced in an on-line fashion, while the user is editing the model (in some cases, this is called *syntax-driven editing*, whereby syntactically incorrect models cannot be constructed). The other method, *batch mode evaluation* means that users are free to create models, and the satisfaction of well-formedness constraints is evaluated at the user's request - e.g. with the model elements that violate constraints being highlighted.

4. **Dynamic (operational) semantics**
   Support for model simulation at editing time. For example, if the system under design has distinct *states*, a simulator could be used to visualize how the system variables change as a state transition occurs.

5. **Transformations (Denotational/Translational semantics)**

   (a) Model-to-model:
   Support for translation between multiple domains. In the Multi-Domain Systems Engineering approach, systems can be modeled in multiple domains simultaneously. This requires translations between models representing various domains.
   Note that model simulation is also model-to-model transformation.

   (b) Model-to-code: Code generators
   Many programs provide template engines and a programming interface to traverse the model space and generate formatted code output.

2.2.2 Integration criteria

1. **Persistence**
   Support for storing models in relational databases, external model containers, various standard XMI/XML formats.

2. **Documentation**
   Support for automated documentation generation, in various standars (UML for models, JavaDoc for generated code, for example).

3. **Advanced features**
   Support for the integration of DSM technology into the model-based development process using run-time dynamic techniques, for examle a Web Service interface, or a relational database backend.

2.2.3 Architectural properties

1. **Generated editors or runtime framework**
   Many DSM frameworks use code generation to create domain-specific editors, while others present a domain-specific view at run-time. The former approach is generally easier to implement (and may be somewhat faster), while the latter is considered more agile, especially for the language engineer, since changes in the domain metamodel immediately effect the behaviour of the domain-specific editor without waiting for a (long) code regeneration to finish.

2. **Editor capabilites**
   This category is hard to approach objectively. Apart from standard graphical editing features, such as move/resize/zoom/property grid, many tools support more advanced techniques.

3. **Accessibility**
   In this context, accesibility means how the tool can be tailored to the needs of a specific domain. While in most cases a (visual) modeling-based approach is used for the definition of at least the abstract syntax, some solutions provide extensive Application Programming Interfaces (API) for the definition of code generators, simulators, etc. These APIs, while definitely flexible, can also be considered rather cumbersome, especially in contrast to a pure modeling-based approach.

2.2.4 Typical workflow

In this part, the typical workflow to create a domain-specific editor and an application code generator is described (as performed by the language engieer).

The results of the evaluation are summarized on Table 2.1.

2.3 Commercial products

2.3.1 MetaCase

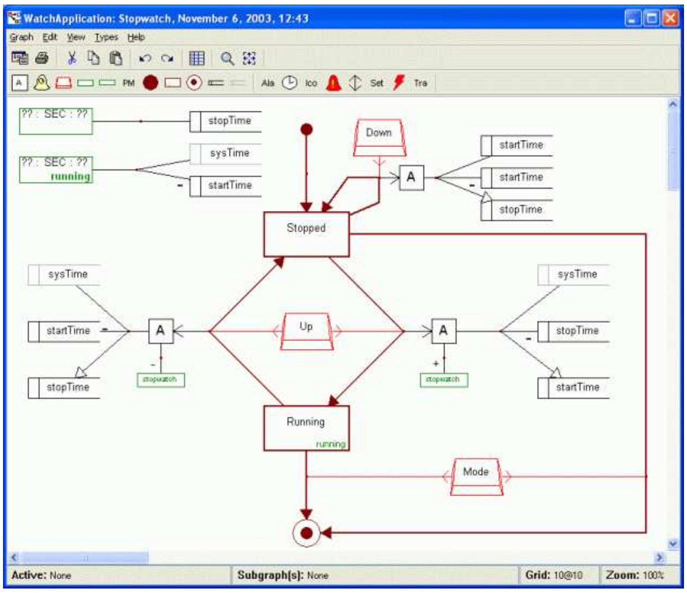
Note: this short review of MetaCase technology is based on the official MetaCase homepage[27], the official MetaEdit+ Technical Summary[28] and various publications by MetaCase employees ([38, 20]).

History MetaCase is a Finnish-American company involved in model-based development software design since 1991. Their main product, *MetaEdit+*, dates back to 1995, and has won multiple awards (Best Application Development Software at CeBIT '95, Finnish National Prize for Innovation in 2000, Best Commercial Tool at Net.Object Days / GPCE 2003). Nowadays, MetaEdit+ is regarded as the leading DSM implementation.

MetaEdit+ can be regarded as a proven, mature product, which makes it unique since all the other implementations examined in this part of the thesis have not progessed beyond a prototype status (with the possible exception of openArchitectureWare).

Architecture MetaEdit+ consists of three major components: (i) the Method Workbench for domain metamodeling, (ii) the MetaEdit+ Framework for modeling, and (iii) the Object Repository server, which serves as the model container. MetaEdit+ is a multi-platform product: every component has been ported to major operating systems including Microsoft Windows, Linux, Solaris, and other commercial Unix variants.

Typical workflow The typical workflow is the following: (1) first, a domain language is designed by the language engineer in the Method Workbench; (2) this language description is imported into the MetaEdit+ Framework through the Object Repository, and used to create domain-specific models by designers; (3) a code generator is constructed using a custom domain-specific textual language; (4) documentation (reports) and application source code is generated from the


WatchApplication: Stopwatch, November 6, 2003, 12:43

Active: None
Subgraph(s): None
Grid: 10@10
Zoom: 100%

Figure 2.1: MetaCase MetaEdit+

models stored in the Object Repository using the pre-defined documentation generator and the custom-made code generator. MetaEdit+ does not use code generation for domain-specific editors; instead, the framework generates an interface from the domain language description at runtime.

Language Engineering support

Concrete syntax As MetaEdit+ is a commercial product with a relatively long history, considerable effort was put into making the tool as user friendly as possible. Thus, this is the category where MetaEdit+ is way ahead: with a built-in library for basic graphical elements, and an intuitive Symbol Editor (for drawing and importing graphical symbols), users can design complex concrete syntax elements without writing any code. Symbol elements can be conditional on property values and can display values calculated by generators, allowing dynamic graphical behavior. Commonly used symbols or parts of symbols can be stored in the Symbol Library for reuse.

Multi domain visualization MetaEdit+ appears to be restricted to editing in one domain per project (but supports multiple views per domain).

Diagram modeling Apart from the traditional graphical views, MetaEdit+ offers the capability to view the same models as matrices, or tables, changing the view according to the users' needs. However, MetaEdit+ lacks a separate diagram modeling layer, because representation is always designed in strong correlation with the abstract syntax (i.e. logical model elements are assigned a visual figure), therefore diagrams not separated from logical models.

Abstract syntax MetaEdit+ uses an own core metamodel, which basically describes a directed graph with typed nodes and edges; nodes are called 'Objects' and edges are called 'Relationships'. Properties of arbitrary types can be assigned to both Objects and Relationships. Type constraints for Relationship endpoints can also be specified.

Well-formedness rules Both static and language-specific well-formedness constraints are supported. With the 'Port Tool', possible interfacing constraints when connecting objects can be defined (although it is rather unclear how complex these constraints can be). MetaEdit+ identifies rules inside and between models. For one modeling language you can define how its concepts can be related to each other and how many connections (of certain type) are allowed between each instance. For example, it may be defined that an instance of 'Initial State' may have only one triggering connection and that the instance must send the same notification event to at least two places. This type of rule forces then all developers using MetaEdit+ to make correct definitions for initial states. It is unclear whether all constraints are enforced during editing in an event-driven manner (in effect making syntactic errors impossible), because it is also mentioned that it is possible to make reports which check the consistency of the models, which suggests a batch-mode type constraint enforcement approach.

Dynamic semantics MetaEdit+ provides an API to read, create, and update model elements, as well as control MetaEdit+ for scripting or simulation support. Moreover, a model-based approach to simulation can also be used: with the 'Graph Tool', "it is possible to manage specifications of whole modeling techniques, such as State Diagram and Component Diagram. Techniques are composed of the Objects, Relationships and Roles defined with other tools, together with bindings and rules on how these can be connected. Different techniques can be integrated with explosions, decompositions and reusable modeling concepts."

Thus, MetaEdit+ provides a mechanism for "subtyping" your concepts with pre-defined dynamic semantics (e.g. State Machines), and the code generator will generate code based on the generic template defined for the dynamic semantics description. However, customized model simulation support is NOT present in MetaEdit+, only some kind of code execution tracking mechanism is provided (which shows the 'Active State', for example, if the generated source code is step-debugged in a separate view). Note that available documentation on this topic is rather unclear.

Transformations

Model-to-model MetaEdit+ was not designed to be a (mathematically precise) model-to-model transformation system. Although the documentation mentions the possibility of creating rules that define how the concepts of one modeling language can be related to concepts in another, all publications suggest that MetaCase's approach is based on the idea that code should be generated directly from domain-specific models, instead of going through several intermediate levels of abstraction (e.g. PIM and PSM in MDA).

Model-to-code MetaEdit+ supports automatic code generation for predefined and user-defined programming languages. The possibilities for automatically generating the code depend on the methods used and target programming languages. Predefined code generators are available for Smalltalk, C++, Java, Delphi (Object Pascal), SQL, CORBA IDL.

In MetaEdit+, a code generator is constructed using a domain-specific textual language[20], based on a graph traversal approach.

Integration All the tools are integrated through the Object Repository, which maintains and enforces the consistency between the tools. Apparently, there is no support for external or relational database model containers. MetaEdit+ offers pre-built reports for model analysis, checking and documentation in Word, RTF, HTML, XML and XMI.

MetaEdit+ allows the user to build sophisticated tool integration between MetaEdit+ and other tools. Alternative tool integration approaches include:

- Programmatic access to model data and MetaEdit+ functions via API
- Model importing and exporting as XML
- Command line parameters for automating MetaEdit+ operations
- Executing external commands via generators

With MetaEdit+, it is possible to make reports which check the consistency of the models, analyse model linkages, create data dictionaries, produce documentation, generate code or configuration information and export models to other programs, such as simulators, version management, external solvers etc. The advanced scripting commands allow the user to print designs in various formats, handle output to several files, and even call external programs.

Advanced features The Object Repository can be integrated into the existing design environment using SOAP/Web Services.

Success stories According to the official MetaCase homepage, MetaEdit+ has already been successfully used in the following target application domains: Mobile devices, Embedded software, Financial applications, Industrial automation, Web applications, Workflow applications, IP Telephony services.

2.3.2 Microsoft DSL Tools

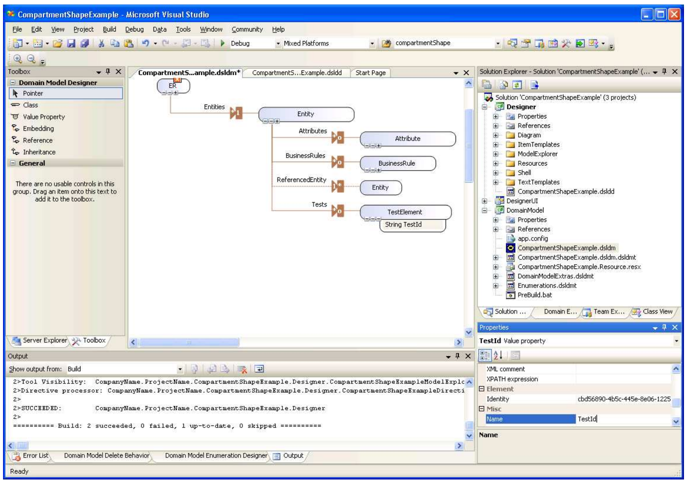
Note: this short review is based on the official Microsoft DSL Tools homepage[30], the official DSL Tools Walkthroughs[31], and various publications by the designers ([17, 18, 14]).

The Microsoft Tools for Domain-Specific Languages is a suite of tools for creating, editing, visualizing, and using domain-specific data for automating the enterprise software development process. These new tools are part of a vision for realizing Software Factories, a new development concept by Microsoft. In a nutshell, a Software Factory is a development environment configured to support the rapid development of a specific type of application, thus, it is, in essence, Microsoft's approach to model-driven software development based on domain-specific modeling.

History In recent years, Microsoft has given indications of increased interest in model-driven software development. Even Bill Gates stated that the most important innovation in the next 10 years is going to be visual modeling tools, "that will reduce software coding by a factor of five."[40]

Microsoft's vision of model-driven software development, *Software Factories*, was laid out in a book[18] in 2004. In a nutshell, this approach argues that the software development industry is in a desperate need of a paradigm shift, because, although the added value in software production is tremendous, the production costs are at an unacceptably high level. Software production needs to be *industrialized*, just like car production was at the beginning of the past century. The key to this, as they conclude, is a model-driven development approach based on domain-specific modeling.

As a result, Microsoft began developing technologies that would turn this vision into reality. The result, a suite called Microsoft DSL Tools, was released to the public in late 2004.



Show output from: Build

```
2>Tool Visibility:  CompanyName.ProjectName.CompartmentShapeExample.Designer.CompartmentShapeExampleModelExplc
2>Directive processor: CompanyName.ProjectName.CompartmentShapeExample.Designer.CompartmentShapeExampleDirecti
2>
2>SUCCEEDED:        CompanyName.ProjectName.CompartmentShapeExample.Designer
2>
========== Build: 2 succeeded, 0 failed, 1 up-to-date, 0 skipped ==========
```

Figure 2.2: Microsoft DSL Tools, September 2005

Architecture The DSL Tools suite integrates into the powerful Visual Studio development platform as a set of development environment plug-ins. The suite of tools is supported by a code framework that makes it easier to define domain models and to construct a custom graphical designer hosted in Visual Studio. The suite consists of:

- A new project wizard for creating a fully configured solution in which you can define a domain model that consists of a designer and a textual artifact generator. Running a completed solution from within Visual Studio opens a test solution in a separate instance of Visual Studio, allowing you to test the designer and artifact generator.

- A format and an updated graphical designer for defining and editing domain models.

- An XML format for creating designer definitions, from which the code for implementing designers is generated. This allows you to define a graphical designer hosted in Visual Studio without any hand coding.

- A set of code generators, which take a domain model definition and a designer definition as input, and produce code that implements both of the components as output. The code generators also validate the domain model and designer definition, and raise errors and warnings accordingly (code generators for generating domain-specific editors).

- A framework for defining template-based artifact generators, which takes data (models) conforming to a domain model as input, and outputs text based on the template. Parameters in the template are substituted using the results of running a C# script embedded in the template (code generators for domain-specific output).

Thus, the DSL Tools suite employs code-generated editors running in a separate Visual Studio instance.

Typical workflow The typical workflow is the following: (1) define the domain metamodel (which is called ”domain model” in the documentation) using the Domain Model Designer, starting from a minimal language template, using the project wizard; (2) create the designer definition; (3) create the text templates for the domain-specific output code generators; (4) define concrete syntax representation using the provided graphical designer; (5) generate code and resources for the domain-specific editor plug-in; (6) launch the created plug-in in a new Visual Studio instance and edit domain-specific models; (7) use the domain-specific output code generator templates to generate application source code.

Language Engineering support

Concrete syntax The DSL Tools suite provides a built-in graphical designer; the suite uses an own metamodel for the description of shapes and decorators, which is rather limited in the September 2005 release.

Multi-domain visualization Apparently, there is no planned support for this feature, as the editors generated by the toolkit are currently meant to be standalone features (so there is no domain editor framework within Visual Studio), bound to a single modeling language. However, support for multiple diagram types per domain is planned.

Diagram modeling The DSL Tools suite supports the mapping of concrete syntax to abstract syntax using a declarative description called *diagram maps*, which are embedded into the designer definition XML file. However, similarly to MetaEdit+, these maps only support simple one-to-one correspondence between visual and logical elements. Therefore, conceptual separation or a separate visualisation modeling layer are absent from the DSL Tools suite.

Abstract syntax There are six predefined templates, and the generator wizard requires the user to choose one as a starting point for the newly defined domain-specific modeling language. However, there is no need to code view classes in C#, as these graphic artifacts can be designed using the provided graphical designer tool. The speciality of the DSL Tools suite is that not only the appearance of the diagram elements can be customized, but the views, palettes, and other components of the generated editor plug-ins as well.

- Minimal Language - A simple template that creates a very small, generic language to build upon, including only two domain concepts, and a notation comprising one box and one line (this is the generic directed graph with assigned types and properties).
- Simple Architecture Chart - A template that includes an example of each of the notational elements currently supported.
- Entity Relationship - A template that can be used to create compartment shapes.
- Activity Diagrams - A template that demonstrates UML activity diagram notation.
- Class Diagrams - A template for UML class diagram notation.
- Use Case Diagrams - A template for UML use cases notation.

According to the documentation, the internal core metamodel is Microsoft's own work, this is also supported by their statements criticizing MOF and XMI [14].

Well-formedness rules At this state of development, only simple static constraints (e.g. type and containment) are supported; only batch mode evaluation is available.

Dynamic semantics There is no current, nor planned support for dynamic semantics.

Transformations There is no current, nor planned support for model-to-model transformations. Model-to-code transformations are supported through a template engine, based on C# scripts.

Integration The DSL Tools suite is integrated into Microsoft's leading development platform, Visual Studio. However, as the DSL Tools suite is essentially a technological demonstration, no other integration options have been implemented (and the documentation is rather self contradictory on this topic).

Persistence Models are stored and exported in XML files.

2.4 VMTS

Note: this short review is based on the official VMTS homepage[51], and a study comparing different prototype model transformation tools[11], co-authored by one of the principal authors of VMTS: Tihamér Levendovszky.

The Visual Modeling and Transformation System (VMTS) is an integrated metamodeling and model transformation system, developed at the Department of Automation and Applied Informatics of the Budapest Univertity of Technology and Economics. For model visualisation and domain-specific modeling, a tool called *Adaptive Modeler* is available. As of May 2006, the current version of the suite is 1.0 Beta.

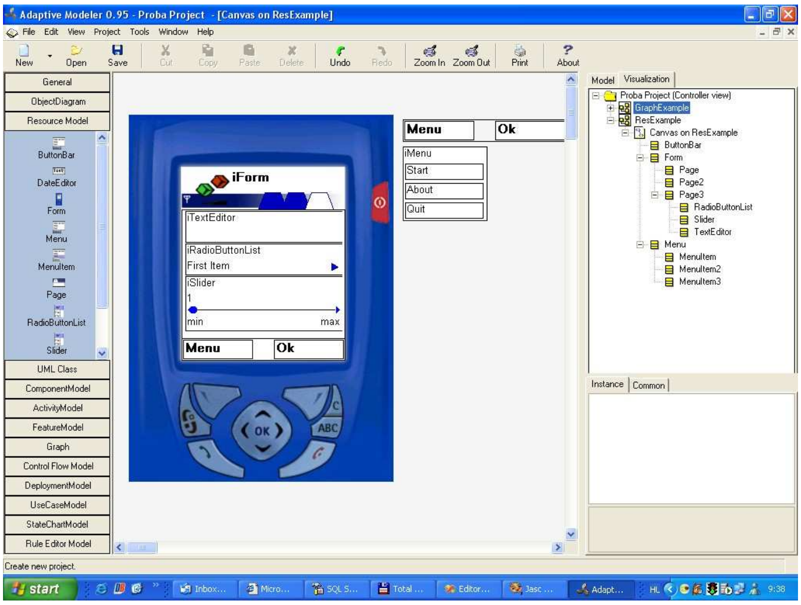


Figure 2.3: VMTS Presentation Framework - Adaptive Modeler 0.95

Architecture The VMTS is a client-server application with a relational database backend. On the server side, the metamodeling core and the rewriting engine reside, while client side consists of the Rule Editor, Modelers and other applications.

Typical workflow As of now, automatic generation of domain-specific editors for VMTS is only partially supported, manual coding is required for visual syntax and editing functionality.

Concrete syntax The VMTS Presentation Framework is a class library which supplies: (i) built-in base classes for the general presentation facilities of shapes (nodes) and lines (edges). (ii) Automatic event handling for the common functionalities such as resizing, moving and selecting model elements. (iii) Automatic serialization for the properties of the model elements. (iv) Sophisticated presentation of attributes, model structure, visualization information and editing features[11].

Abstract syntax VMTS uses Attributed Graph Arhitecture Supporting Inheritance (AGSI) as its core metamodel, which basically describes a directed labeled graph, where nodes can have attributes and inheritance is supported between nodes.

Well-formedness rules OCL constraints (compiled into a .NET assembly) are supported, with static constraints enforced at editing time.

Dynamic semantics None.

Transformations As VMTS was primarily intended to be a metamodeling and model transformation framework, a powerful, graph transformation-based model transformation engine is provided, using a UML-like notation.

Integration VMTS offers a Traversing Model Processor interface, where the model elements appear as regular objects in a programming language, and traversing classes are also provided by the framework. The types of these objects are obtained from the metamodel[11].

2.5 Eclipse

2.5.1 The Eclipse Integrated Development Environment

The Eclipse Project[7] is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It was developed by IBM from 1999, and a few months after the first version was shipped, IBM donated the source code to the Eclipse Foundation.

The Eclipse project consists of many subprojects, the most important being the Eclipse Platform, that defines the set of frameworks and common services that collectively make up ”integration-ware” required to support a comprehensive tool integration platform. These services and frameworks represent the common facilities required by most tool builders, including a standard workbench user interface and project model for managing resources, portable native widget and user interface libraries, automatic resource delta management for incremental compilers and builders, language-independent debug infrastructure, and infrastructure for distributed multi-user versioned resource management.

The Eclipse Platform has an easily-extendable modular architecture, where all functionality is achieved by plugins, running over a low-level core called Platform Runtime. This runtime core is only responsible for loading and connecting the available plugins, every other functionality, such as the editors, views, project management, is handled by plugins. The plugins bundled with Eclipse Platform include general user interface components, a common help system for all Eclipse components, project management and team work support.

2.5.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[8] is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF is also maintained by the Eclipse Foundation within the scope of Eclipse Tools Project. EMF started out as an implementation of the OMG Meta Object Facility (MOF) specification, and evolved into a highly efficient product for model-based software design.

EMF requires a metamodel as an input; it can import metamodels from different sources, such as Rational Rose, XMI or XML Schema documents, or a special syntax can be used to annotate existing Java source files with EMF model properties. Once an EMF model is specified, the built-in code generator can create a corresponding set of Java implementation classes. These generated classes can be edited to add methods and instance variables; additions will be preserved during code-regeneration. In addition to simply increasing productivity, building an application using EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

EMF consists of two fundamental frameworks: the core framework and EMF.Edit. The core framework provides basic generation and runtime support to create Java implementation classes for a model.

EMF has a built-in serialization facility, which enables the developer to save (and load) instances of the model into industry-standard XMI format. EMF also provides a notational and persistence mechanism, on top of which model-manipulating tools can easily be constructed.

EMF.Edit is a framework which includes generic reusable classes for building editors for EMF models. It provides content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard Eclipse user interface views and property sheets. It also provides a command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo for models. Finally, it provides a code generator capable of generating everything needed to build a complete editor plugin for an EMF model. It produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors. The code can then be customized without losing connection to the model. For example, using GEF, a graphical editor can be created.

EMF and EMF.Edit together implement the Model-View-Controller (MVC) design scheme, where EMF serves as the model, EMF.Edit serves as a controller and the user can freely decide how the view is implemented. The connection between the controller and the model uses the aforementioned EMF notification mechanism. EMF.Edit can also automatically generate a tree-view based editor, which integrates into the Eclipse platform as a plugin.

2.5.3 Graphical Editing Framework

The Graphical Editing Framework (GEF) for Eclipse is an open source framework dedicated to providing a rich, consistent graphical editing environment for applications on the Eclipse Platform. In other words, it is a complex system, or graphical library, designed to make developing graphical editors for Eclipse easy and quick. Eclipse is, apart from being a great development environment, a general design tool in the sense that its modular plugin centered design enables the user to integrate graphical modeling tools, interface editors and other GUI-driven components into this rich environment, and have these work together flawlessly. In order to avoid confusing the users with numerous different-looking and working interfaces, and to enable developers to create graphical plugins quickly and efficiently, GEF was developed by the Eclipse Project.

The word *Graphical* in GEF is provided by Draw2D, a lightweight graphical component library. Lightweight means that Draw2D elements (unlike SWT[1] widgets) do not require a separate handle from the operating system window manager. The advantage of a lightweight system is that an entire element hierarchy is seen as a single element from the outside, therefore the management of the elements lies completely in the hands of the lightweight framework. That means an increase of speed in case the number of elements is large. The lightweight library can generally optimize the management of its elements at a greater extent than an ”outsider” window manager could. Draw2D elements are called figures, and any Draw2D figure may contain other figures. The only constraint imposed on a figure hierarchy is that child figures must lie entirely within the boundaries of their parent. Draw2D includes a basic set of predefined figures (polygons, labels, borders, buttons, arrows, ...) but developers might create their own user-drawn figures.

The *Editing* capabilities of GEF are very similar to the Eclipse Platform itself. It uses the request-command abstraction scheme. That means that every user activity (keystrokes, mouse operations) is first translated to requests. Such a request is more abstract than an OS-level mouse of keyboard message. An example of a GEF request would be something like: *Model object Place0 has been dragged from position (15, 21) to position (42, 7).* These requests are received by the model elements involved and translated to one or more commands. A command is responsible for the effective modification of the model.

Actions

Actions are very similar to requests, they represent certain user activity. The major difference between actions and requests are that requests are generated by GEF internally (eg. when the mouse is dragged on a GEF-handled area of the screen), while actions can be generated programmatically. Generally graphical editing operations create requests, whereas menu items, toolbar buttons generate actions. Both requests and actions create commands to modify the model.

GEF follows the Model-View-Controller (MVC) pattern. MVC divides the responsibilities of an editor and assigns them to three distinct elements. The model is solely responsible for representing the data being displayed. That includes serialization (if required) and model manipulation. The model itself is not required to be aware that there is a graphical editor built upon it. The view, on the other hand, is only responsible for the visualization of the model. The view does not care about model manipulation, it just simply updates itself from time to time to reflect changes in the model. What forms the bridge between the two is the controller. The controller is responsible both for modifying the model according to user activities, and for updating views when the model changes. The controller is the element, which accepts editing requests and modifies the model. After the model has been modified, the controller asks the affected view elements to refresh themselves. GEF does not care about what kind of model we are using, the views are required to be Draw2D figures and the controller is provided by GEF, it is called *EditPart*.

However from the developers perspective, it is the *Framework* that makes the difference: although one can create a general purpose graphical interface with Draw2D or any other graphical library, GEF was specifically designed for graphical editors. Thus, if the goal is to not only display data, but also be able to manipulate it, one should consider using GEF instead of implementing everything from scratch. GEF provides ready solutions for a list of common editor problems (drag-and-drop, property sheet handling, tree-based visualization, clipboard support, ...) and the parts of GEF are closely related to each other, and been tested together. GEF has already been

[1] Standard Widget Toolkit, the default widget library of Eclipse Platform products.

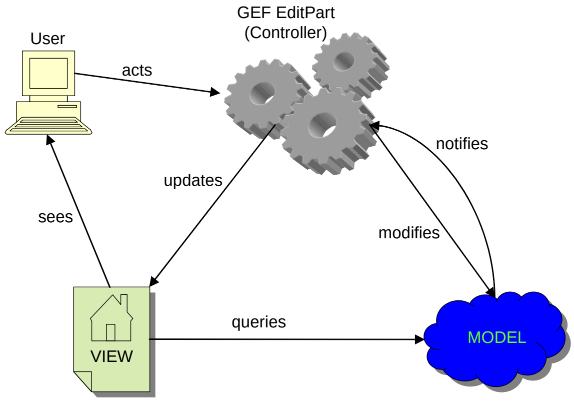


Figure 2.4: MVC scheme inside the Graphical Editing Framework

successfully used for commercial applications, like UML editors, GUI builders and workflow management software.

2.5.4 Domain-specific editors with EMF and GEF

Using EMF and GEF, editors for domain-specific visual languages can be implemented - although not very easily. EMF has a lot of automated features, but connecting a GEF-based visualisation layer to an EMF model is a tough job, certainly not suited for non-programmers/domain experts, as demonstrated by [20, 32].

Typical workflow The typical workflow for creating an EMF-GEF-based editor could be the following: (1) first, the domain metamodel is defined in some UML tool; (2) this metamodel is imported into EMF, and EMF generates all the necessary implementation classes with notification and serialisation support, etc; (3 - optional) using EMF.Edit, a tree-view based editor can be generated - at this point, a usable, tree-based domain-specific editor, in the form of an Eclipse plugin, is ready to be used (and it was automatically generated); (4) the necessary classes for GEF are implemented manually; (5) a code generator is implemented manually, for instance with the powerful Java Emitter Templates API (JET)[9].

Concrete syntax Automated support is only available for a tree-view based editor, everything else has to be implemented manually. Draw2D includes a fairly rich set of common graphics primitives.

Abstract syntax EMF uses an own core metamodel called ECore, which is almost identical to Essential MOF (MOF 2.0). Every imported metamodel is mapped onto ECore concepts.

Well-formedness rules Only simple static constraints are supported, as EMF currently lacks OCL support.

Dynamic semantics Since there is no automated support, dynamic semantics, and model transformations in general can only be implemented using EMF's model manipulation API.

Transformations As EMF is only a model container, no model-to-model transformation is supported. Code generation has to be implemented manually, however, the Java Emitter Templates API is a very powerful template engine using a special template language (with JSP-like syntax).

Integration EMF supports XMI serialization, and can import metamodels from XML Schema Descriptions, Rational Rose models, and XMI files.

2.5.5 Eclipse GMF

Even though both EMF and GEF are fairly powerful tools in the hands of a professional Java programmer, the Eclipse developers have realised that, in their current from, these technologies are simply inadequate for rapid language engineering. Thus, an intermediate framework was drafted, which would serve as a generative bridge between EMF and GEF, whereby a diagram definition would be linked to a domain model as input to the generation of a visual editor. This is The Graphical Modeling Framework[15] project, which aims to provide the fundamental infrastructure and components for developing visual design and modeling surfaces in Eclipse.

The Graphical Modeling Framework is currently in a beta stage draft, with a set of project requirements[16], a project plan published on the website, and several downloadable pre releases, which our review is based on.

Architecture GMF uses generated editors, which support the following features: Palette, Properties, Overview (bird's eye view), Zoom, Navigator, Outline, Decorators, Keyboard bindings, Direct editing, Drag and drop, Layout, Support standard graphical editor facilities (actions, rulers, guides) provided by GEF, Provide support for compartments/subcompartments, feedback (in status line) for constraint violation, advice, and Filter views.

Typical workflow The typical workflow to create a domain-specific editor and a code generator is the following: (1) create an *"EMF model"* (which is actually the domain metamodel); (2a) create diagram metamodels; (2b) create diagram metamodels; (2c) create a mapping model between the EMF model and the diagram model; (3) refine the domain metamodel on a graphical interface, add OCL constraints; (4) design the visual representations for diagram elements using a graphical interface; (5) generate the domain-specific visual editor; (6) use the generated Eclipse plug-in for modeling; (6) manually implement a JET-based code generator; (7) generate application code.

Concrete syntax The GMT project plans a full-fledged visual toolkit based on Draw2D primitives, using an intuitive graphical interface integrated into Eclipse.

Multi-domain visualisation GMF employs a separate diagram modeling layer; diagram metamodels are stored using EMF and describe diagrams to the detail level of graphical attributes (size, position etc). The mapping between diagrams and models is facilitated through generated code, which is based on a *mapping metamodel*. The mapping allows for a partial one-to-many mapping of the logical domain into multiple diagram domains, however a connection is always represented by an edge, and a class is always represented by a node. Thus, the conceptual separation is, while extensive compared to other tools, still partial. For more details, see Sec. 6.2.

Diagram modeling support A diagram may contain multiple references to a single domain model element, potentially with each having a different diagram representation. A diagramming metamodel is provided to allow for diagram definitions. A mapping metamodel is provided to allow for diagram to domain model mapping definitions.

Abstract syntax The GMF allows for the creation of a new EMF model using a graphic editor, leveraging an ECore modeling surface.

Well-formedness rules Diagram and/or domain models may have constraints added which need to be manifested as feedback in the graphical editor. For example, a constraint indicating that circular relationships are not allowed should be indicated in the UI while attempting to make such a connection (editing time enforcement). Constraints may be defined in the diagram and/or domain models that are more appropriately checked in a batch mode, as is done with the EMF validation framework. GMF should allow for constraints to be enforced using this or similar framework. GMF will provide support for the Object Constraint Language (OCL).

Dynamic semantics None.

Transformations Only model-to-code generation using JET.

Integration

Persistence Diagrams are stored in files using XML/XMI capability of EMF. Alternative persistence mechanisms are planned to be made available via extension points. Export: Printing, Images; UML Diagram export support via Diagram Interchange Specification; Team integration support.

2.5.6 openArchitectureWare

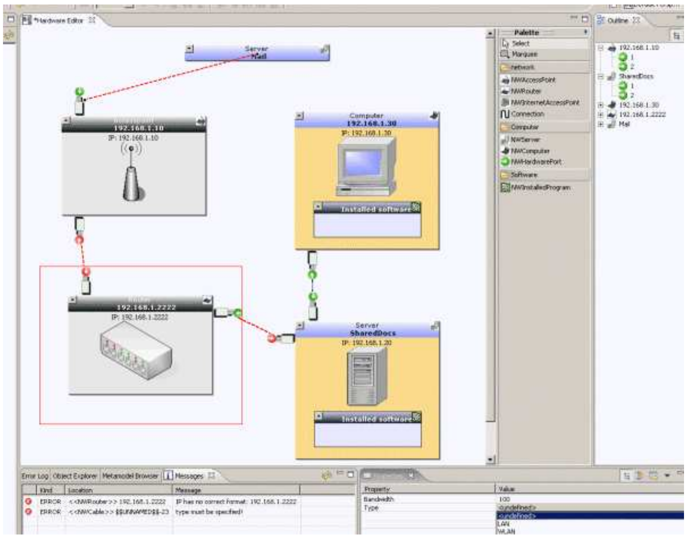
Note: this short review is based on the official openArchitectureWare homepage[35], and a publication by the principal authors, Markus Völter and Bernd Kolb[25].

The openArchitectureWare framework is a suite of tools and components assisting model-driven software development. It is an open source project with around 10 active developers.

History Originally, the framework was developed as a commercial product by a German company called *b+w GmbH*[3]. As of 14 September 2003, the b+w Generator Framework is developed as an open source project, hosted at SourceForge. Since the original code commit, many features have been added. The openArchitectureWare framework has been successfully used in a wide spectrum of industrial applications.

Architecture The openArchitectureWare framework (oAW) is a comprehensive tool suite for model-driven software development. The project is made up of several components (all written in Java):

- Core: A modular MDA/MDD generator framework supporting arbitrary import formats, meta models, and output formats.



Kind	Location	Message
ERROR	<<NWRouter>> 192.168.1.2222	IP has no correct format: 192.168.1.2222
ERROR	<<NWCable>> $$UNNAMED$$-23	type must be specified!

Property	Value
Bandwidth	100
Type	<undefined> <undefined> LAN WLAN

Figure 2.5: openArchitectureWare

- Utilities: A lot of useful utilities for the generator.
- Eclipse Plugin: Eclipse IDE support for openArchitectureWare.
- Metamodel Generator: Automating the manual development of Java metaclasses.
- Feature Modeling: Integrating openArchitectureWare with the *pure::variants* feature modeling tool.
- GEF Editors: Generating GEF Editors for oaW models.
- Visio Integration: Reading visio models into oAW.

Typical workflow The typical workflow for creating a domain-specific visual editor and code generator in oAW is the following:

1. The metamodel is created, either by manually coding Java classes or using the Metamodel Generator (which can generate these Java classes from a UML model).
2. Java classes responsible for visual representation are implemented manually by the language engineer.
3. The visual editor generator creates a GEF-based editor.
4. Models are created using this visual editor.
5. The templates are written in oAW-s own domain-specific template language (there is an intelligent editor available with Code completion and syntax-highlighting support available).
6. An existing model is imported and instantiated, which can be validated against the domain metamodel.
7. The Generator will create an Eclipse editor, based on the domain metamodel.
8. Code can be generated using this editor and the templates.

Concrete syntax

Automated support Although there is a thin component library for GEF visualisation classes available, concrete syntax representation classes have to be implemented manually.

Multi-domain visualisation Not supported. All editors generated in oAW are logically bound to one metamodel.

Diagram-model mapping As the graphical editor generator is in a very early stage of development, there is absolutely no automated support for a custom diagram-model mapping. The generated GEF editors are rather simple, there is no conceptual separation between diagrams and models.

Abstract syntax openArchitectureWare uses a very flexible approach, as there is no fixed core metamodel, instead any Java class can be used as a metamodel element. This has obvious advantages, but can be cumbersome because knowledge of the Java programming language is required even for a basic metamodel definiton. Thus, a Metamodel Generator feature was soon added, which can generate classes based on a UML model, in which case, stereotypes and tagges values are required (there is a pre-defined base UML model).

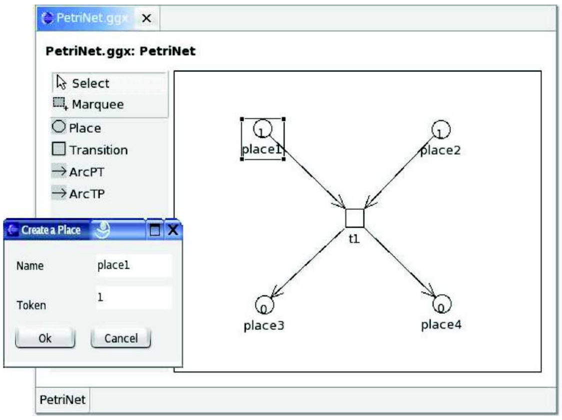
Well-formedness rules Static constraints from UML models are supported. The static constraints can only be evaluated in batch mode. There is preliminary support for declarative metamodel constraints.

Dynamic semantics There is no support for dynamic semantics.

Transformations Only model-to-code transformations are supported, using the powerful oAW template engine and a domain-specific textual language for creating templates.

Integration

- UML tool support
  - MagicDraw
  - Enterprise Architect
  - Poseidon
  - Together/J
- Generator Utilities
  - ANT integration
  - Integration with JUnit
  - Event-Based Modelmodifiers to update dependent parts of a model
  - various frontends (instantiators), e.g. a generic XML reader, an instiator to read Microsoft Visio models
  - a generic XML writer
  - Invokers to handle ”cross-cutting concerns” in the generator



Name place1

Token 1

Figure 2.6: Petri net editor generated with Tiger

2.5.7 Tiger

Note: this section is based on the official Tiger website[43], the User Documentation[44], and a publication by the authors[12].

The Tiger Project (Transformation-based generation of modeling environments) is a tool environment that allows to generate an Eclipse editor plugin based on the Graphical Editing Framework (GEF) from a formal, graph-transformation based visual language specification.

At the current stage of development, the Tiger framework can generate GEF-based diagram editors using combined domain and diagram metamodels, with a tool called Attributed Graph Grammar System (AGG-Engine)[1]. Tiger's approach is quite interesting for two reasons: (i) some of GEF's graphical concepts (such as drawing primitives, layout constraints) have been included in the metamodel and thus Tiger is able to generate code which uses GEF's advanced graph layout algorithms to generate diagrams in an elegant fashion; (ii) in domain-specific editors generated with Tiger, all editing actions are translated into high-level graph-transformation rules which are executed within the model container - in other tools, basic editing functionality uses API calls. The authors call this *syntax-directed editing*, which basically means the on-line enforcement of well-formedness constraints, in effect making the creation of syntactically incorrect models impossible. Note, however, that for simple static constraints this can be and actually is guaranteed in most other tools using the traditional approach.

Architecture The Tiger project uses generated Eclipse plug-in editors, which currently provide very basic editing capabilities. Models are stored in the AGG graph-transformation core.

Typical workflow The typical workflow to create a visual editor is the following: (1) create a Visual Language specification (which includes both abstract and concrete syntax) using the provided Java classes (tiger.vlspec) - this currently requires manual coding; (2) load the Visual Language into a separate software component and edit the syntax grammar; (3) use the tiger.generator component to generate the GEF-based editor.

Concrete syntax Tiger provides a basic set of *ShapeFigures* which can be assigned to model elements (these include Ellipse, Circle, RoundedRectangle, RectangleFigure, Polygon). Additional attributes like color and size can be assigned to these ShapeFigures. Layout constraints (e.g. *”a token is inside a place”*) can be defined between ShapeFigures using Java language constructs.

Abstract syntax Tiger's core metamodel describes a directed labeled graph. Nodes are called *NodeSymbolType*, edges are called *EdgeSymbolType*; attributes (*AttributeType*) can be assigned to both edges and nodes. In the Tiger core metamodel, nodes and edges need to be connected through links (*LinkType* - these are the definitons for simple type constraints for edges.

Well-formedness rules Tiger uses *grammar syntax* to capture both static and language-specific well-formedness rules (these are basically graph transformation rules).

Dynamic semantics Although support is planned, currently the specification and execution of dynamic semantics is not supported.

Transformations As Tiger is based on a graph transformation engine, there is a possiblity to support both model-to-model and model-to-code transformations; at the current stage, however, none of them are.

Integration Not applicable due to the early stage of development; however as Tiger generates Eclipse plug-ins,all standard integration features of Eclipse (e.g. Team Synchronization support) are available.

2.6 Summary

The summary of our reviews can be seen on Table 2.1.

	Language Engineering											Architecture			Integration		
	Concrete syntax				Abstract Syntax	Well-formedness			Dynamic Semantics	Transformations							
			3														
	1	2	3/1	3/2		4	5	6		7	8	9	10	11	12	13	14
MetaEdit+	yes	no	yes	?	custom	yes	yes	online batch	no	?	yes	Run-time	Prof.	API	own	yes	SOAP
DSL Tools	yes	no	yes	yes	custom	yes	no	batch	no	no	yes	Generated	Prof.	API	XML	no	Visual Studio
oAW	no	no	no	no	Java class	yes	?	batch	no	no	yes	Generated	Basic	API	XML	yes	Eclipse
VMTS	no	?	yes	?	AGSI	yes	yes	?	no	GT	yes	Non-gen.	Basic	API	Rel. DBMS	?	no
EMF/GEF	no	*	*	*	ECore	yes	no	*	*	*	JET	*	*	API	XMI	*	Eclipse
GMF	yes	yes	yes	yes	ECore	yes	yes	online batch	no	no	JET	Generated	Prof.	API	XMI	?	Eclipse
Tiger	No**	?	no	no	AGG	yes	yes	Online ?	plan	GT	yes	Generated	basic	?	?	no	Eclipse
Viatra2 DSM	plan	yes	yes	yes	VPM	yes	yes	online batch	yes	GT ASM	yes	Run-time	plan	API	own	yes	Eclipse

Table 2.1: The summary of our research

Legend to Table 2.1 See also 2.2 for detailed explanation.
1 - graphical syntax designer; 2 - multi-domain design; 3/1 - diagram-model conceptual separation; 3/2 - arbitrary diagram-model mapping; 4 - static constraints; 5 - language-specific constraints; 6 - constraint enforcement; 7 - model-to-model transformations; 8 - model-to-code transformations; 9 - runtime framework or generated editors; 10 - editing capabilities; 11 - accessbility; 12 - persistence; 13 - documentation generation; 14 - integration (advanced features)
'*': as EMF/GEF are not suited for automatic editor generation, most of the categories are left blank because they all depend on the individual implementations.
'**': although Tiger has no automated support for the interactive design of concrete syntax representation, an easy to use (albeit limited in capabilities) library is provided.

Evaluation

By looking at Table 2.1, the following conclusions can be drawn:

- *Concrete syntax*
  Easy-to-use graphical syntax designers require a lot of effort to implement well. In commercial products, such as MetaEdit+ or DSL Tools, this is considered a very imporant feature because it *"sells"* the software better, however, in research projects this is usually one of the last development targets.

- *Abstract syntax*
  All projects use a core metamodel which describes a typed, labeled graph with attributes (in some cases, this core metamodel is an ECore or MOF model). This is fairly straightforward because visual languages are, in essence, typed and labeled graphs with attributes.

- *Concrete syntax - abstract syntax mapping*
  Although the usual separation of models and diagrams, which is found in most modeling tools, is supported by most of the applications (most notably MetaEdit+ and Microsoft DSL Tools), the language engineers hands are tied because nothing more than specifying the set of elements displayable on a given diagram type, and their concrete appearance, can be customised. The exception is GMF: it is the most modern approach, because diagrams are handled almost independently from models (even at the metamodel level), however GMF is fairly conservative in using the full potential of its approach because the mapping mechanism is still limited to the connection - edge, class - node paradigm.

- *Constraints*
  For simple constraints, all approaches use an on-line evaluation scheme, while languge-specific constraints, if supported, are usually handled by a batch-mode OCL evaluator.

- *Dynamic semantics / Simulation*
  It is apparent that - with the exception of Tiger - no project supports (or even plans to support) dynamic semantics modeling. In some cases, this feature could be added using the application programming interface; the lack of a model-based approach to dynamic semantics specification can be explained by the fact that it would require a model transformation infrastructure, which only Tiger and VMTS use.

- *Model-to-model transformations*
  Model-to-model transformations, if supported, are based on a graph transformation technique.

- *Model-to-code transformations*
  Code generation is usually implemented with a template engine (e.g. JET), or a custom domain-specific programming language, suited for easy modelspace traversal and formatted code output.

- *Architecture*
  - Most approaches use code-generated domain-specific editors because they are usually easier to implement (although usually limited in capabilities, e.g. no multi-domain support).
  - With the exception of professional products, currently all projects provide fairly limited editing cabilities, in spite of the powerful integrated environments these editors run in.
  - Persistence: most projects use in-memory modelspace containers, with the exception of VMTS (RDBMS backend), and MetaEdit+ (custom Object Repository server). All products support serialization and model export into some standard XMI or XML format.

- *Integration*
  While many projects support (on even rely on) external (meta)models for editor code generation, some require manual Java coding, or non-automatic model conversion by a professional. Documentation generation, similarly to visual syntax editors, is unique to commercial products (MetaEdit+, DSL Tools). Many projects integrate into popular Integrated Development Environments (Visual Studio, Eclipse), which can be considered a must since nowadays most software developer houses use these even for the smallest projets.

2.7 Our approach

It is clear from the review that while many initiatives provide support for some aspects of language engineering, a *comprehensive* domain-specific modeling environment that integrates **all** aspects of language engineering and provides efficient ways of domain integration is not yet existent. The *Multi-Doman Systems Engineering* approach requires all of the above conditions to be met.

To provide this all-round solution, we created a domain-specific modeling framework built on top of a powerful model transformation architecture: the ViatraDSM framework. With this approach, all aspects (with the exception of concrete syntax) of language engineering can be handled within the same (programming) domain (i.e. a high-level domain specific language suitable for defining metamodels and model transformations).

In our approach,

- the precise specification of domain-specific visual languages can be described using a high-level formal language;
- the abstract syntax of these languages can be defined using visual and precise metamodeling techniques;
- modeling constraints, model transformations, model simulation and code generation can be uniformly specified by a combination of graph transformation and abstract state machines.

An overview of our approach, the ViatraDSM framework, is presented on Fig 2.7.

The implementation criteria for our system were as follows (see also the last row of Table 2.1):

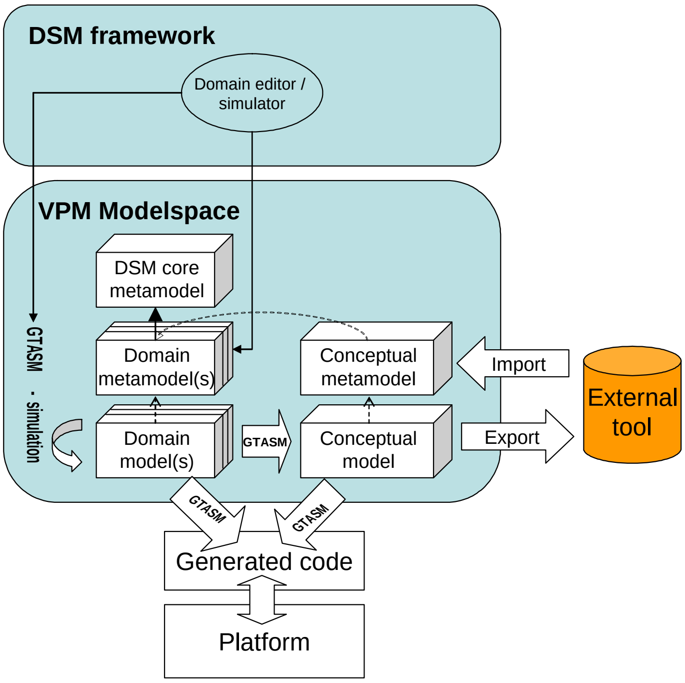


Figure 2.7: Our framework in the DSM approach

Architecture The system consists of two main components: (i) a model transformation framework; and (ii) a domain-specific modeling environment. The model transformation framework provides the necessary infrastructure model persistence, model-to-model and model-to-code transformations, and model import/export. The domain-specific modeling environment presents domain-specific views of the integrated modelspace, and provides rich editing and simulation facilities, based on the services of the model transformation framework.

Runtime domain-specific editors Our approach uses a runtime framework for supporting domain-specific editors, because this facilitates a more rapid development process for domain-specific editors and simulators. This approach also makes supporting a multi-domain modeling environment easier.

Editing capabilities The system is integrated into the Eclipse development environment. The domain-specific editors make use of the standard workbench features (Properties view, undo-redo support, copy-cut-paste support) as well as some advanced techniques (intelligent text completion and validation).

Accessibility In our approach, the domain-specific editors can be tailored to the needs of the target application domain in various ways:

- an Application Programming Interface (API) is provided for the implementation of concrete syntax, based on the GEF and Draw2D libraries;
- a set of domain-specific programming languages (VIATRA2 languages, see 3 for details) for specifying modeling constraints, model transformations, model simulation and code generation;

- the system can also adapt to other model containers and model transformations environments as necessary.

Concrete syntax

Visual editors We use a highly customizable and platform independent graphical application programming interface (GEF), which enables language engineers to create arbitrary concrete syntax representations. We do not consider the automated visual support for concrete syntax design a high priority, but our flexible implementation allows for the integration of easy-to-use visual syntax designers at a later stage of development.

Multi-domain visualisation We consider multi-domain modeling and visualisation crucial, therefore our approach provides full support for the integration of multiple domain-specific editors into a single environment, with the possibility of switching between various views instantly. Moreover, our design allows for model and diagram elements from different domains be handled within the same editor (e.g. a graphical interdomain model transformation designer). For a more detailed discussion of this issue, see Chapter 5.

Diagram modeling support We consider the full conceptual separation of diagrams and models a must, because a tree-based (or even textual) modeling environment can be faster to work with in many scenarios. Moreover, the performance of the whole system can be improved drastically if the user is free to decide what to visualise graphically. Furthermore, our approach implements a separated diagram modeling layer, which supports the metamodel-level design of diagrams, which are mapped to logical models through a mapping mechanism. The mapping can be implemented using Java code, based on well-defined APIs, or using declarative techniques (mapping metamodel), combined with native VIATRA2 transformation programs, executed in an event-driven fashion. For more details, see Chapter 6.

Default concrete syntax representations While our approach relies on manual coding for many aspects concerning the definition of concrete syntax, we provide support for a default, UML-like graphical notation for models so that domain-specific editors can be developed rapidly without any manual coding.

Abstract syntax Our approach is based on the VIATRA2 framework, which provides a flexible core metamodeling language (VPM, 3.1.1). We define a core metamodel for domain-specific visual languages using VPM concepts, which is used by the runtime domain-specific editor framework to provide model editing capabilities to various concrete domain editors and simulators. As the system is to be integrated into an existing model-based development environment, existing domain metamodels can be imported using the native import facilities of the VIATRA2 framework (see 3.3), which can then be mapped automatically to our core domain metamodel, thereby facilitating the generation of domain-specific editors for domains that are external to the system.

Well-formedness rules As the model transformation language of the VIATRA2 framework is powerful enough to express arbitrary static and language-specific constraints in an intuitive way (as demonstrated by the example in 7, our approach uses read-only transformations evaluated in a batch-mode fashion for the enforcement of complex constraints. Static constraints that can be handles by the VPM metamodel (containment, type correctness, multiplicity) are enforced in an on-line fashion.

Dynamic semantics As the model transformation language of the VIATRA2 framework is expressive and intuitive enough for the specification of dynamic semantics (7), and the interpreter-based execution of these model transformations is sufficiently efficient, our approach provides full support for interactive model simulation.

Transformations As demonstrated in Chapter 3, the VIATRA2 framework provides a robust and efficient environment for the specification and execution of model-to-model and model-to-code transformations. In 7, we give a simple example for model-to-code transformations.

Integration

Persistence Our approach is based on the VIATRA2 modelspace container, which provides support for a wide range of output formats, including tagged formats (XML variants) and textual languages.

Documentation With the powerful transformation capabilities of VIATRA2, development documentation can be generated on-the-fly from models of various abstraction levels.

Flexibility As our domain-specific modeling framework can be adapted to any model container, the code generation facilities of EMF (JET) and other technologies can also be integrated into our concept.

Typical workflow A language engineer would have to perform the following tasks to create a domain-specific editor, simulator, and code generator:

1. Create a domain metamodel using an external tool, the VPM editor of our modeling environment, or the VIATRA2 Textual Metamodeling Language.
2. Project this metamodel onto the core domain metamodel of the DSM framework (by specifying an appropriate transformation in the VIATRA2 Textual Command Language, or designing one with the model transformation editor).
3. The DSM framework will instantly provide a domain-specific editor with a tree view.
4. Create a diagram metamodel, a mapping model, a mapping implementation (either by coding or using the mapping metamodel), and a the DSM framework will provide a basic UML-like graphical notation for diagrams.
5. Using the API of the DSM framework, and Draw2D primitives, create a customized concrete syntax representation.
6. Define dynamic semantics using the VIATRA2 Textual Command Language, or the model transformation editor.
7. Define a code generator using the VIATRA2 Textual Command Language, or the model transformation editor.

Note that the language engineer only has to learn *one* expressive domain-specific language for *all* aspects of language engineering.

2.8 Example: Petri Net

In the thesis, the following simple but descriptive domain-specific visual language example will be used for demonstrating the feasibility of our approach. In Chapter 7, a complete description of building a domain-specific modeling tool based on the ViatraDSM framework's facilities is given.

Petri nets (Fig. 2.8) (abbreviated as PN) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. From a system modelling point of view, a Petri net model is frequently used for correctness, dependability and performance analysis in early stages of design. Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary number of Tokens. A token distribution defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by InArcs) and add a token to all output places (as defined by OutArcs).

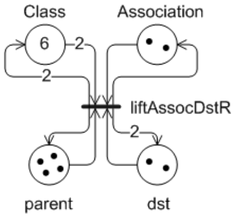


Figure 2.8: A sample Petri net with place capacities and arc weights

Chapter 3

Interpreter-based model transformation in VIATRA2

3.1 Metamodeling: Definition of Abstract Syntax

In this section, I conceptually follow [4].

3.1.1 Visual and Precise Metamodeling

Currently, most widely used metamodeling languages (e.g. ECore) are derived (with slight variations) from the Meta Object Facility (MOF) [29] metamodeling standard issued by the OMG. However, as stated in [46], the MOF standard fails to support multi-level metamodeling, which is typically a critical aspect for integrating different technological spaces where different metamodeling paradigms (e.g. EMF, XML Schemas) are used.

Therefore, the VPM (Visual and Precise Metamodeling) [46] metamodeling approach was chosen in the VIATRA2 framework, which can support different metamodeling paradigms by supporting multi-level metamodeling with explicit and generalized instance-of relations.

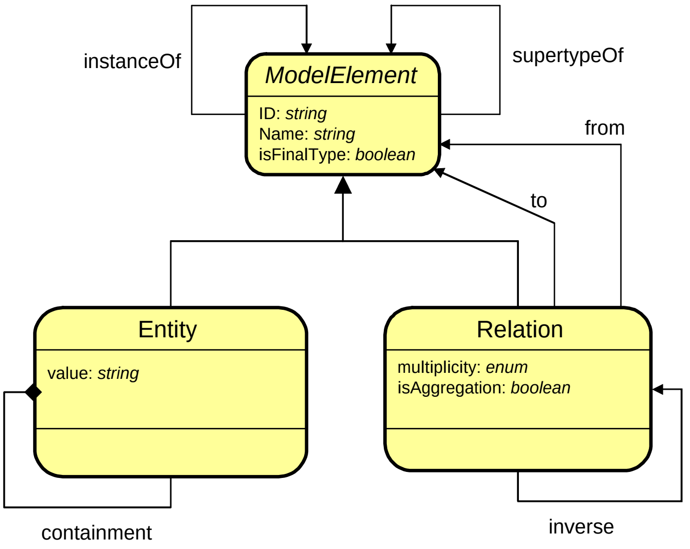


Figure 3.1: The VPM Metamodel

The VPM language consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). *Entities* represent basic concepts of a (modeling) domain, while *relations* represent the relationships between other model elements[1]. Furthermore, entities may also have an associated value which is a string that contains application-specific data.

Model elements are arranged into a strict containment hierarchy, which constitutes the VPM model space. Within a container entity, each model element has a unique local name, but each model element also has a globally unique identifier which is called a fully qualified name (FQN).

Fully qualified names are constructed in a different way for entities and relations:

- an Entity's fully qualified name is the fully qualified name of its parent and the name of the entity concatenated (separated with a dot).
- a Relation's fully qualified name is the fully qualified name of source and the name of the relation concatenated (separated with a dot).
- There is an entity with no parent: the *root entity* is the root of name hierarchy. The fully qualified names of the children of the root entity equal the name of the child entity.

The construction of the fully qualified name imposes an important constraint on the VPM modelspace: the containment hierarchy for entities must not contain loops, and for every relation, it must be true that a finite traversal along the source endpoints ends up at an entity (otherwise, the fully qualified name would be infinite). This constraint is enforced by the runtime VPM core implementation.

All elements have a globally unique ID, which can not change during the life cycle of the model element (in contrast, names are free to change).

There are two special relationships between model elements: the *supertypeOf* (inheritance, generalization) relation represents binary superclass-subclass relationships (like the UML generalization concept), while the *instanceOf* relation represents type-instance relationships (between meta-levels). By using an explicit *instanceOf* relationship, metamodels and models can be stored in the same model space in a compact way.

The formal transitivity rules of instantiation and inheritance are the following:

$$\begin{gathered} instanceOf(a,b) \wedge subtypeOf(b,c) \Rightarrow instanceOf(a,c) \\ subtypeOf(a,b) \wedge subtypeOf(b,c) \Rightarrow subtypeOf(a,c) \\ instanceOf(a,b) \wedge instanceOf(b,c) \not\Rightarrow instanceOf(a,c) \end{gathered}$$

If the *isFinalType* attribute of a VPM element is set to *true*, then only instances of that model element can be created (i.e. the model element cannot be subtyped).

Relations have additional properties:

(i) Property *isAggregation* tells whether the given relation represents an aggregation in the metamodel, when an instance of the relation implies that the target element of the relation instance also contains the source element.

(ii) The *inverse* relation points to the inverse of the relation (if any). In a UML analogy, a relation can be considered as an association end, and thus the inverse of a relation denotes the other association end along an association.

[1] Most typically, relations lead between two entities to impose a directed graph structure on VPM models, but the source and/or the target end of relations can also be relations.

(iii) Relations also have *multiplicities*, which impose a restrictions on the model structure. Allowed multiplicity kinds in VPM are one-to-one, one-to-many, many-to-one, and many-to-many. This information can be used by the pattern matcher search plan generator.

3.1.2 The VTML language

In VIATRA2, the textual metamodeling language supporting VPM is called VTML (Viatra Textual Metamodeling Language). The technicalities of VTML are demonstrated in Fig. 3.2 on a simplified UML metamodel presented originally in the model transformation benchmark of [11].

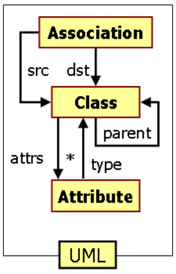


Figure 3.2: Sample UML metamodel

The VTML equivalent of the metamodel is as follows.

```
entity(UML)
{
        entity(Class);
        entity(Association);
        entity(Attribute);
        relation(src,Association,Class);
        relation(dst,Association,Class);
        relation(parent,Class,Class);
        relation(attrs,Class,Attribute);
        multiplicity(attrs,many-to-many);
        relation(type,Attribute,Class);
}
```

The basic elements of the language are the element declarations defined as Prolog-like facts. An entity can be declared in the form *<type>(<name>)*, where *type* is the type of the given entity and *name* is the name of the new entity. Type declarations are mandatory, because all entities must have a type. If an entity has no definite type, it is instantiated from the basic VPM *entity* model element. As entities may contain other model elements, the containment is done similarly to the C language, where the program blocks are marked with braces ({}). Here, the contained elements are represented in a block surrounded by braces after the container entity.

A relation can be defined similarly, but the source and target model elements must also be marked. The syntax of relation definition is the following: *<type>(<name>, <source>, <target>)*. A relation is always contained by its source entity.

The containment hierarchy defines namespaces in the model space. This enables the definition of the fully qualified name (FQN) of model elements. The FQN is equal to the list of containers of a given model element from the model space root to the element, separated by dots. For example, the FQN of the entity Association in the example is *UML.Association*, while the FQN of the relation src is *UML.Association.src*. The local (short) name of a model element must be unique in its container, this also ensures the uniqueness of FQNs.

Special relationships can be represented by the keywords *supertypeOf*, and *subtypeOf* for generalization, and *typeOf*, and *instanceOf* for instantiation. The syntax is the following: *<relationship>(*

<supplier>, <client>). For example, *typeOf(UML.Class, Dog)* defines that the entity Dog is an instance of the metamodel element UML.Class. This way, a model element may have multiple types to support multi-domain modeling.

The VTML language has also a *namespace import* definition that can be used to import namespaces for the given VTML file. Model elements that are in the imported namespaces can be referred to using their local names instead of the default fully qualified names for user's convenience. Namespace import has the following syntax: *import <namespace>;* For example, if we want to create an instance model using the simplified UML metamodel above, we can use the *import UML;* command to import the UML namespace. After that, elements of the metamodel (like *UML.Association*) can be referred to using their local names (as *Association*).

3.2 The VTCL language

Transformation descriptions in VIATRA2 consist of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [10] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [5] rules can be used for the description of control structures.

The language that is created to implement all these concepts is the Viatra Textual Command Language (VTCL). This language is primarily textual, but it will soon be extended by a graphical editor that will support the graphical definition of model transformations.

3.2.1 Graph patterns

Graph patterns, negative patterns

Graph patterns are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model.

A model (i.e. part of the model space) can satisfy a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique presented in [45].

In the following example, a simple pattern can be fulfilled by class instances that do not have parent classes.

```
/* C is a class without parents and with non-empty name */
pattern isTopClass(C) =
{
        UML.Class(C);
        neg pattern negCondition(C) =
        {
                UML.Class(C);
                UML.Class.parent(P,C,CP); UML.Class(CP);
        }
    check (name(C)!="")
}
```

Patterns are defined using the *pattern* keyword. Patterns may have parameters that are listed after the pattern name. The basic pattern body contains model element and relationship definitions, which are identical to the VTML language constructs.

The keyword neg marks a subpattern that is embedded into the current one to represent a negative condition for the original pattern. The negative pattern in the example can be satisfied, if there is a class (CP) for the class in the parameter (C) that is the parent of C. If this condition can be satisfied, the outer (positive) pattern matching will fail. Thus the pattern matches to top-most classes in parent hierarchy.

There are also *check* conditions that are Boolean formulae which must be satisfied in order to make the pattern true. In our example, we check whether the name of the class is empty. The pattern can be matched to classes with non-empty names only.

A unique feature of the VTCL pattern language among graph transformation tools is that negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [39].

Pattern calls, OR-patterns, recursive patterns

In VTCL, a pattern may call another pattern using the *find* keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled.

Alternate bodies can be defined for a pattern by simply creating multiple blocks after the pattern name and parameter definition, and connecting them with the *or* keyword. In this case, the pattern is fulfilled if at least one of its bodies can be fulfilled. The two features (pattern call and alternate (OR) bodies) can be used together for the definition of *recursive pattern*. In a typical recursive pattern, one of the bodies contains a recursive call to itself, and the other defines the stop condition for the recursion. The following example illustrates the usage of recursion.

```
// Parent is an ancestor (transitive parent) of Child pattern
ancestorOf(Parent,Child) =
{
        UML.Class(ParentClass);
        UML.Class.parent(X,Child,Parent);
        UML.Class(Child);
} or
{
        UML.Class(Parent);
        UML.Class.parent(X,C,Parent); UML.Class(C);
        find parentOf(C,Child); // pattern call
        UML.Class(Child);
}
```

A class *Parent* is the parent of an other class *Child*, if it is a direct parent of the child class (first body), or it has a direct child (C), which is the parent of the child class (second body). The pattern uses recursion for traversing multi-level parent-child relationships, and uses multiple bodies to create a halt condition (base case) for the recursion.

The semantics of graph patterns

When a predefined graph pattern is called using the *find* keyword, this means that a substitution for the free (unbound) parameters of the specified graph pattern has to be found that satisfies the pattern. if it has no defined value. If there are bound variables passed as parameters, they are treated as additional constraints, and they remain substituted (bound) throughout the pattern matching process. By default, the free variables will be substituted by *existential quantification*, which means that only one (non-deterministically selected) matching will be generated. If a variable is universally quantified by the external *forall* construct (see Sec. 3.2.3), the matching will be done (in parallel) for all possible values of the given variable.

3.2.2 Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [10], which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its left-hand side (LHS) pattern with an image of its right-hand side (RHS) pattern.

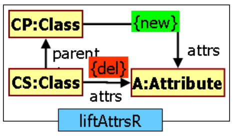


Figure 3.3: Sample graph transformation rule

The sample graph transformation rule in Figure 3.3 defines a refactoring step of lifting an attribute from child to parent classes. This means that if the child class has an attribute, it will be lifted to the parent.

The VTCL language allows both popular notation for defining graph transformation rules. The first syntax of a GT rule specification corresponds to the traditional notation: it contains a *precondition* pattern for the LHS, and a *postcondition* pattern that defines the RHS of the rule. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in RHS are created, and other model elements remain unchanged.

```
gtrule liftAttrsR(in CP, in CS, in A) =
{
        precondition pattern cond(CP,CS,A,Attr) =
        {
                UML.Class(CP);
                UML.Class(CS);
                UML.Class.parent(Par,CS,CP);
                UML.Attribute(A);
                UML.Class.attrs(Attr,CS,A);
        }
        postcondition pattern rhs(CP,CS,A,Attr) =
        {
                UML.Class(CP);
                UML.Class(CS);
                UML.Class.parent(Par,CS,CP);
                UML.Attribute(A);
                UML.Class.attrs(Attr2,CP,A);
        }
}
```

The graph transformations rules are defined using the *gtrule* keyword, and they are allowed to have directed (in/out/inout) parameters. The LHS and RHS patterns share information on matchings by parameter passing.

The second format directly corresponds to the graphical (FUJABA [33]) notation as shown in the following example.

```
gtrule liftAttrsR(in CP, in CS, in A) =
{
        condition pattern cond(CP,CS,A) =
        {
                UML.Class(CP);
                UML.Class(CS);
                UML.Class.parent(Par,CS,CP);
                UML.Attribute(A);
                del UML.Class.attrs(Attr,CS,A);
                new UML.Class.attrs(Attr2,CP,A);
        }
}
```

The rule contains a simple pattern (marked with the keyword *condition*), that jointly defines the left hand side (LHS) of the graph transformation rule, and the actions to be carried out. Pattern elements marked with the keyword *new* are created after a matching for the LHS is succeeded (and therefore do not participate in the pattern matching), and elements marked with the keyword *del* are deleted after pattern matching.

In both cases, further actions can be initiated by calling any ASM instructions within the *action* part of a GT rule, e.g. to report debug information or to generate code.

There is also a third format of graph transformation definition that is more likely to the procedural programming languages. The rule contains a precondition (LHS), like the previous one, but instead of defining the RHS pattern we have to define the actions to be executed. The actions can be any ASM instructions (see Section 3.2.3). The actions that are defined after the *action* keyword are executed sequentially. It is important to note that the action section can also be used with the other two forms of graph transformation definition, for example to create debug outputs or generate code.

```
gtrule liftAttrsR(in CP, in CS, in A) =
{
        precondition pattern cond(CP,CS,A,Attr) =
        {
                UML.Class(CP);
                UML.Class(CS);
                UML.Class.parent(Par,CS,CP);
                UML.Attribute(A);
                UML.Class.attrs(Attr,CS,A);
        }
        action
        {
                new(UML.Class.attrs(Attr2,CP,A));
                delete(Attr);
        }
}
```

The interpreter of the VIATRA2framework supports all these formats simultaneously, so developers can choose the rule format that is more suitable for them.

Generic and meta-transformations

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic and meta-transformations, which are built on the multilevel metamodeling support. For instance, we may generalize rule *liftAttrsR* as lifting something (e.g. an Attribute) one level up along a certain relation (e.g. parent). The following example is the generic equivalent of the previous GT rule parameterized by types taken from arbitrary metamodels during execution time.

```
gtrule liftUp(in CP, in CS, in A,
              in ClsE, in AttE, in ParR, in AttR) =
{
   condition pattern transClose(CP,CS,A,
             ClsE, AttE, ParR, AttR) =
   {
      // Pattern on the meta-level
      entity(ClsE);
      entity(AttE);
      relation(ParR,ClsE,ClsE);
      relation(AttR,ClsE,AttE);
      // Pattern on the model-level
      entity(CP);
      // Dynamic type checking
      instanceOf(CP,ClsE);
      entity(CS);
      instanceOf(CS,ClsE);
      entity(A);
      instanceOf(A,AttE);
      relation(Par,CS,CP);
      instanceOf(Par,ParR);
      del relation(Attr,CS,A);
      del instanceOf(Attr,AttR);
      new relation(Attr2,CP,A);
      new instanceOf(Attr2,AttR);

   }
}
```

Compared to *liftAttrsR*, this generic rule has four additional input parameters: (i) *ClsE* for the type of the nodes containing the thing to be lifted (*Class* previously), (ii) *AttE* for the type of nodes to be lifted (*Attribute* previously), and (iii) *ParR* (ex-*parent*) and (iv) *AttR* (ex-*attrs*) for the corresponding for edge types.

When interpreting this generic pattern, the VIATRA engine first instantiates the type parameters (*ClsE*, *ParR*, etc.) and then queries the instances of these types. As a result, the same rule can be applied in various modeling languages.

Invoking graph transformation rules

To execute graph transformation rules they have to be invoked from a transformation program. The basic invocation is done using the *apply* keyword. In this case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters. A rule can be executed for all possible matches (in parallel) by quantifying some of the input parameters using the *forall* construct. The following example illustrates some possible invocations of our sample rule.

```
// simple execution of a GT rule
// all variables must be bound
apply liftAttrsR(Class1,Class2,Attrib);

// calling the rule for all attributes of a class
// variables Class1 and Class2 must be bound
forall A do apply liftAttrsR(Class1,Class2,Attrib);

// calling the rule for all possible matches
forall C1, C2, A do apply liftAttrsR(C1,C2,A);
```

3.2.3 Control Structure

To control the execution order and mode of graph transformation the VTCL language includes some concepts that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, we included the basic set of Abstract State Machine (ASM) language constructs [5] that have formal semantics and correspond to the constructs in conventional programming languages.

The basic elements of an ASM program are the rules (that are analogous with methods in OO languages), variables, and ASM functions. ASM functions are special mathematical functions, which store values in arrays. These values can be updated from the ASM program. These functions are called *dynamic*. There are also *static* functions, which means that they cannot change their values. For example, the basic mathematical functions (+,-,*,/) are static.

In VTCL, a special class of functions, called *native function*s, is also defined. Native functions are user-defined Java methods that can be called from the transformations. These methods can access any Java library (including database access, network functions, and so on), and also the VIATRA2 model space. This allows the implementation of complex calculations during the execution of model transformations.

ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (*seq*), rule calls to other ASM rules (*call*), variable declarations and updates (*let* and *update* constructs) and *if-then-else* structures, non-deterministically selected (*random*) and executed rules (*choose*), iterative execution (applying a rule as long as possible *iterate*), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (*forall*).

These basic instructions, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules are also given as ASM programs. The following example demonstrates the main control structures.

```
pattern isClass(C) =
{
   //simple pattern that recognizes classes
   UML.Class(C);
}
rule main() = seq
{
   //Print out some text
   print("The transformation begins...");
   //Call a GT rule for all matches
   forall C1, C2, A do apply liftAttrsR(C1,C2,A);
   //Call other rule
   call printFormatted(123);
   //Iterate through all classes
   forall Cl with find isClass(Cl) do seq
   {
      print("Found a class: "+name(Cl));
   }
   //Write to log
   log(info,"transformation done");
}
rule printFormatted(in C) =
{
   //Print out the value
   print("Value is  : "+C);
}
```

3.3 VIATRA2 Architectural overview

3.3.1 The VIATRA2 framework

The VIATRA2 system is a standalone model container and transformation framework, which can be integrated into the Eclipse IDE as a plug-in. In stand-alone mode, the VIATRA2 system runs as a console application with a command-line console.

Within the Eclipse environment, additional integration components are available:

- a tree-view modelspace editor component, supporting the standard *Properties* view and undo-redo functionality;
- an Eclipse *view* which provides an interface to the import/export/parser facilities;
- a Code output view component to visualize the textual output generated by code generators.

The current implementation of the system allows for multiple *framework* instances within a single Eclipse workbench, thereby enabling users to work with multiple VPM model spaces (and editors) simultaneously.

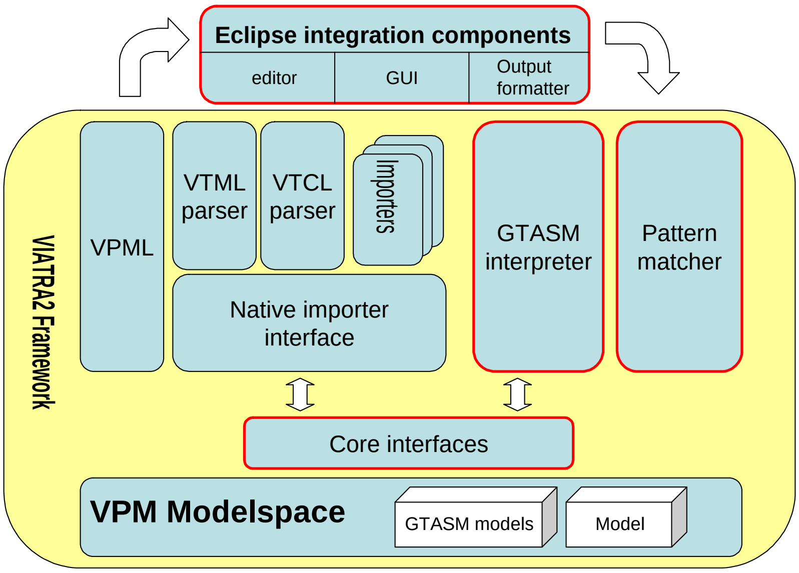


Figure 3.4: The architecture of the VIATRA2 framework

As it can be seen on Fig. 3.4, the internal structure of the VIATRA2 framework can be split up into four major components:

1. VPM modelspace container and VPM core interfaces
2. Pattern matcher
3. GTASM interpreter
4. Import/export facilities

Our development teamwork consisted of the implementation and optimization of the VPM core, the pattern matcher, and the GTASM interpreter (marked with red on Fig 3.4).

VPM Core The VPM Core implementation defines a low-level, simple interface. This interface ensures the integrity of the model. All other components, including the editors and importers, use this interface for queries and modifications. The VPM Core also supports a notification mechanism, an arbitrary depth undo/redo interface, and a simple global locking mechanism to provide preliminary support for concurrent modifications and asynchronous transformations.

Import/export facilities To facilitate the integration of the VIATRA2 framework into an existing model-driven development infrastructure, the *native importer* interface provides support for the construction of import plug-ins which read native formats and instantiate models in the VPM modelspace. The VIATRA2 language (VTML and VTCL) parsers are also implemented as native importers (by András Balogh).

Pattern matcher Pattern matching is an experimental subject. The efficiency of the whole model transformation system highly depends on the efficiency of model storage and pattern matching. Therefore, we designed the VIATRA2 system so that it can be used as a framework of future pattern matchers. There are experiments with different matchers (such as [48]). If they were inserted into the VIATRA2 framework then all features of the GTASM language would automatically work with that implementation (eg. recursive pattern matching and merging sub-patterns).

3.3.2 The GTASM interpreter

The model transformation interpreter is a complex system (see Fig 3.5). The modular implementation makes it possible to have modules with diverse implementations.

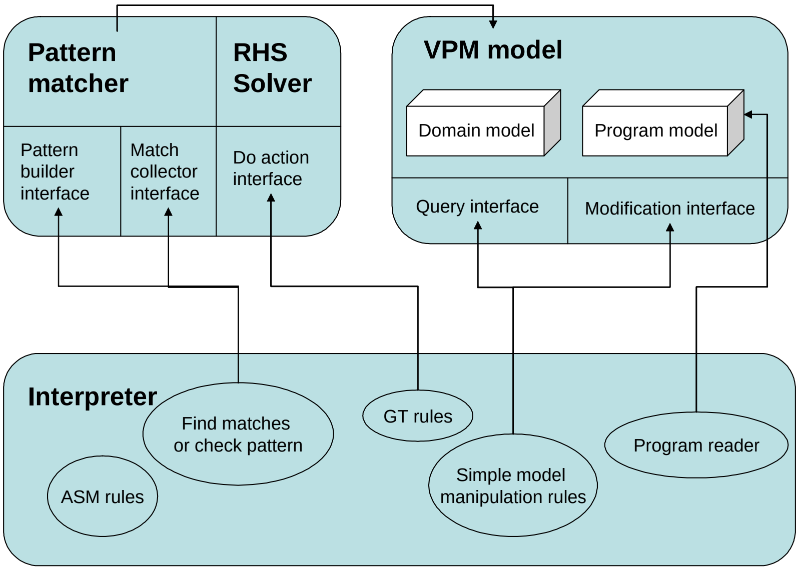


Figure 3.5: The architecture of the VIATRA2 GTASM interpeter

The pattern matcher interface

A higher level VPM model query interface is the pattern matcher. This interface requires a model of patterns and methods to manipulate them.

The interpreter defines a constraint based pattern definition. Over this model the pattern matcher interface defines a pattern builder. The pattern builder accepts variables, constants and constraints between them.

For each variable its type must be specified. There are three types of variables:

1. *"find all"* output variables (to be bound to all matches)
2. *"find one"* output variables (to be bound to the first possible match)
3. input variables (caller must bind to a model element before calling pattern match)

The pattern matcher will return all matches where the *"find all"* variables are matched to different model elements.

The Java interface of pattern matcher allows the following types of constraints:

- constraint on one variable that defines the main type of the element (entity or relation)
- constraint on two variables (these constraint types cover all types of relations between model elements in a VPM model)
- client defined constraints on one or more variables: these constraints are checked by a method provided by the client of the module

The constraint based pattern definition is handy for client programs and for pattern matcher implementations as well. The user defined constraints make the system easily configurable.

The RHS solver interface

Graph transformation rules have a right hand side (RHS) pattern. This pattern is the goal of the transformation. The graph transformation technique finds all matches of the left hand side (LHS) pattern and changes them to be like the RHS pattern. The RHS solver module computes the difference between the LHS and RHS modules and applies the required action on the matching of the LHS pattern.

The RHS solver module interface accepts the two patterns of the same model as the pattern matcher. The additional information required for building a list of actions is to bind the variables of LHS to the variables of RHS.

3.4 Summary

In this section, I introduced the VIATRA2 framework, which supplies the modeling and transformation infrastructure of our *Multi-Domain Systems Engineering* approach. I gave the detailed description of our research and development efforts concerning the implementation and optimization of the framework.

In the next section, I describe our implementation of the domain-specific modeling framework based on VIATRA2, and how the examples provided in this section can be used on an intuitive graphical user interface.

Chapter 4

The ViatraDSM Framework

In this chapter, I introduce the ViatraDSM framework, a tool supporting the construction of dynamic and visual modeling languages, built on top of the model transformation facilities of the VIATRA2 framework. This tool was developed by Dávid Vágó and myself; an early version was presented as a Scientific Students' Association report in November 2005. Since then, it has matured greatly: numerous features, such as extensive simulation support, conceptually separated diagrams, multi-domain modeling support, declarative mapping between logical models and diagrams have been developed. In this section, I provide a general overview of the ViatraDSM framework and discuss its most important features in detail. Some aspects, which are the main contribution of my master's thesis, such as multi-domain modeling and diagram-logical model mapping, are discussed in more detail in chapters 5 and 6.

4.1 Architecture

Figure 4.1 shows the overall architecture of our ViatraDSM framework. At first sight it looks complex and hard to understand. In the next few sections I will elaborate the concepts of that figure, giving detailed descriptions of every major component of our framework. But before going into details about that architecture, I will examine what other architectures could have been possible, and why we chose this particular version.

4.1.1 Editor generation or runtime framework?

The first question which arises when creating a DSM framework is how actual domain specific editors will function. There are two major approaches, editor generation and runtime framework. Editor generation means that based on the metamodels, the code of the domain specific editors are generated by the DSM tool. This approach is used for example by the Eclipse Modeling Framework (EMF). Using EMF, the domain metamodel is specified using an UML diagram or an XML schema definition. From the definition the EMF generates both Java classes, which implement elements of the metamodel, and a simple tree-structured editor, which allows the editing of models.

The other approach is a runtime framework, which means that there is a general (domain independent) model editor, which takes the metamodel as one of its inputs, and uses that metamodel to validate model editing actions. An example of this second approach could be the MetaEdit+ tool.

Consider the Petri net example to see the difference between these approaches. If we model Petri nets using EMF, the framework will generate Java classes such as `Place`, `Transition`, which implement the metamodel (eg. class `Place` will have methods like `getCapacity()`). A simple Eclipse-based editor is also generated, which allows the editing of models. The domain rules (eg.

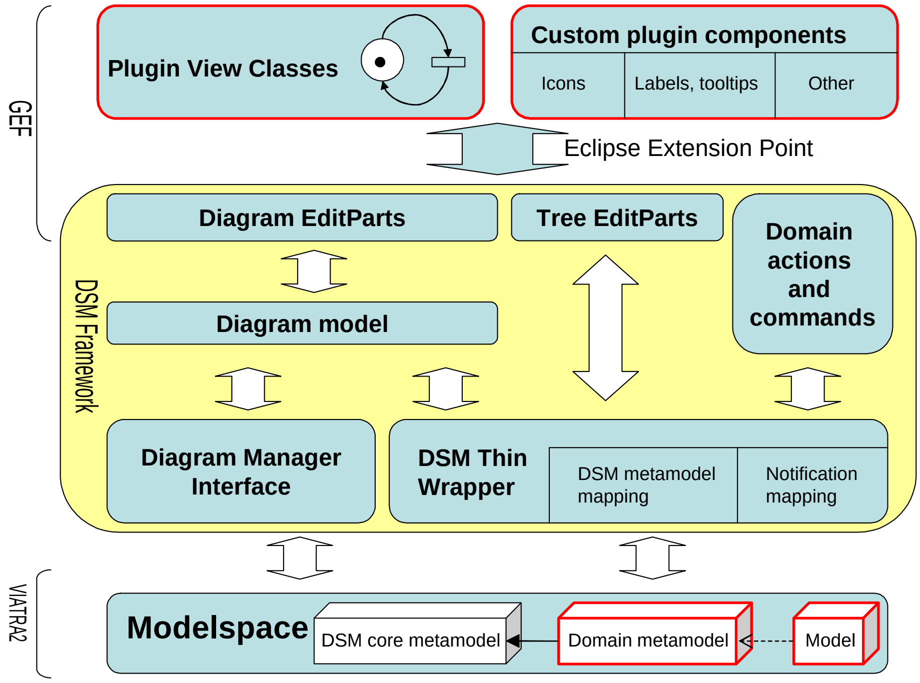


Figure 4.1: Detailed architecture of the DSM framework

places have capacities, transitions do not) cannot be violated, since the structure of the generated classes cannot be changed (generated class `Transition` will not have a `getCapacity()` method, whereas class `Place` will).

On the other hand, if Petri nets are modeled using MetaEdit+, we first design the metamodel and domain rules within that tool. Upon the creation and editing of models, the tool will not generate a separate editor, but it will constantly check the provided metamodel to decide whether a certain editing action is allowed or not. For example in case of Petri nets, when we want to add capacity to a given model element, the editor will check on-the-fly in the metamodel, whether the selected element can have a property called `capacity` or not. Giving a different example for comparison, the difference between the two approaches is similar to the difference between compiled (EMF) and interpreted (MetaEdit+) program execution.

Before designing our DSM framework, we studied both approaches, evaluating the advantages and disadvantages. Editor generation (just like compiled program execution in general) has an advantage in terms of speed. The metamodel is processed only once, when the implementation classes and editor are generated. After code generation, the metamodel might be thrown away, since everything it describes (model structure and constraints) is reflected in the generated code. That gives also one of the most important weakness of editor generation, the difficulty of tracking metamodel changes.

A domain language is not a static concept, it evolves and changes to keep up with changing business processes. That means the domain language has to be updated from time to time. When we have our domain specific editors generated, every (even the smallest) change in the metamodel must be followed by the regeneration of the entire editor. That means a problem, if some custom functionality has been added to the original editor after code generation. Since a DSM framework must be general and should accept a large scale of different metamodels, generated editors tend to be simple, providing only basic editing functions. Therefore a language engineer usually adds more advanced features to the generated editor code to increase its usability and thus its effect on

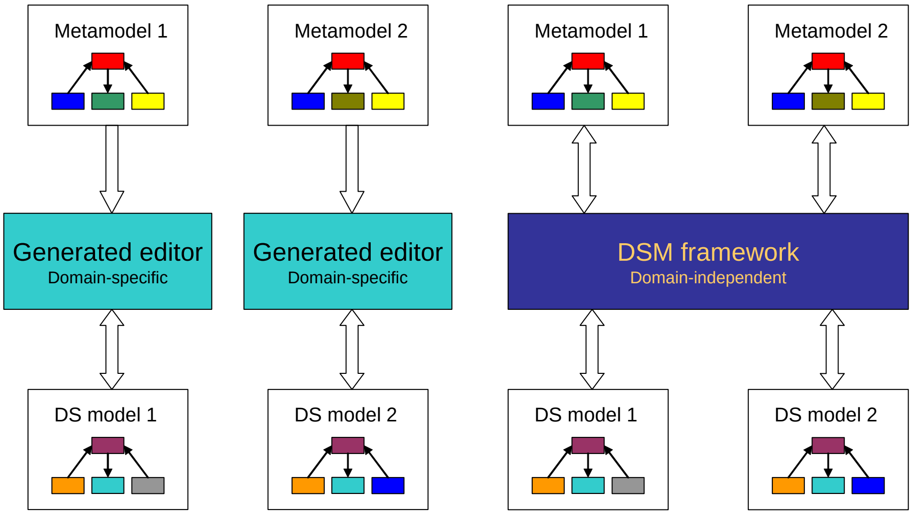


Figure 4.2: Difference between editor generation and runtime framework

productivity. However when the metamodel changes, a new editor will be generated and all the extra features have to be added again to the freshly generated editor code. EMF has some mechanism to detect user-made (in this case language engineer) changes in the generated code, and keep the changed part intact when regenerating code, but that requires placing special tags in the source code. Therefore it is uncomfortable and more importantly, if not handled correctly, it might cause serious and hardly detectable faulty behaviour in generated editors.

On the other hand, a runtime framework approach can get rid of this problem by having a domain independent general editor and making use of the domain metamodel during editing. When the metamodel changes, code need not be generated, just the editor must be told to use the updated metamodel. Just like the difference between fat and thin client software. For fat clients (generated editors), an update must be distributed to each client (adding all extra features again), providing numerous steps for errors. For thin clients (runtime framework), only the software on the server side must be updated (change the metamodel), and the clients will automatically adapt to the modified system. The drawback of this runtime framework approach is reduced speed, since the metamodel must be checked at every editing action to ensure the fulfillment of domain constraints.

After looking at both approaches closely, we decided to use the runtime framework method in our DSM tool implementation. It is true, that the constant processing of the metamodel requires some additional time at editing, but editing is in general carried out by a human user, therefore the time required by checking domain constraints is negligible compared to the time needed by the user to press keys and drag the mouse. However this time penalty has a greater impact on the performance while running automated transformations. But with advanced techniques like metamodel caching and notification collecting (see section 4.2.2 on implementation for details) that problem can also be surmounted.

On the architectural diagram (figure 4.1) this runtime framework approach is reflected by the element *DSM metamodel mapping*. That component is responsible for mapping a domain metamodel inside the VIATRA2 model space to the internal metamodel representation, which is then used to verify the fulfillment of domain constraints during editing. A detailed explanation of that component will be given in section 4.2.2.

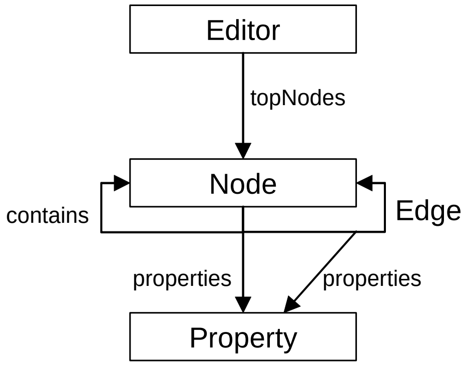


Figure 4.3: Core domain metamodel of our DSM framework

4.1.2 Domain specific graphical representation

To support user-made graphical elements, there are two major ways, drawing and coding. In case of drawing, the DSM tool provides a simple drawing module where the user can create the required element by combining shapes, texts, colors and other graphical primitives. The alternative way, coding, requires the user to effectively write some code, using the programming interface (API) of the graphical library used by the DSM tool. The major differences between the two alternatives are the level of user influence and the required user effort. Drawing is clearly simpler for the users. A few mouse clicks and keystrokes require way less work then writing code using an API previously unknown to the user. But with drawing, we only have the set of basic elements provided by the DSM tool vendor, and that may greatly limit our possibilities. Writing code requires more work from the user, but he will have a total control of what will appear in the editor. Looking back at our mouse trap example, building the figure of a mouse from separate lines, circles and polygons (say these are the only elements provided by the DSM tool) requires much more work than writing a few lines of code, which displays a bitmap image.

Our choice about graphical model representation is reflected by the component *Plugin View Classes* on figure 4.1. These view classes are the short pieces of code a language engineer has to write to provide a custom (domain specific) graphical representation for model elements. Details about these view classes will be given in section 4.2.3.

4.1.3 Modeling

Metamodel structure

In this part, I introduce the core domain metamodel, which defines the set of elements the language engineer may use to build up the metamodel of his own domain.

Figure 4.3 shows the structure of the core domain metamodel; in the following paragraphs, the elements of this metamodel are described in detail.

Editor is the topmost element in every domain. There exists only one single instance of *Editor*, it serves as a model container for each domain. The relation *topNodes* links the topmost nodes to this single container.

Node represents an entity in the domain metamodel. Nodes may arbitrarily be nested into each other by the *contains* relation. The only constraint about containment is that every node must have exactly one parent (topmost nodes have the *Editor* as their parent).

Edge is a relation that links two nodes together. At present, the four multiplicity constraints supported natively by the VPM core (one to one, one to many, many to one, many to many) can be assigned to edges.

Properties are simple (name, value) pairs, which may be assigned to any model element. On diagram 4.3, only one *Property* entity and two *properties* relations are present for simplicity, but in the actual implementation both have various subtypes. The entity *Property* has three subtypes, *TextProperty* is a property, whose value can be any textual data (domain plugins may implement custom property validators to restrict the value). *ColorProperty* is a property, which represents a color in RGB format. It is used mainly as a property of custom diagram elements. Finally, *MultiProperty* is a property, whose value may be only one of a predefined set of possible values (eg. a boolean property which may have only `true` or `false` as its value). The relation *properties* has two subtypes both for nodes and edges. Any defined property can either be required or optional (having relation names *requiredProperties* and *optionalProperties* respectively). When a new model element is created, its required properties are created automatically, and a default value is assigned. Optional properties can be added and removed from a model element at any time.

These four concepts are the basic elements a language engineer may use to build up a custom domain metamodel. Exactly one *Editor* element must be used as a top-level model container. Arbitrary hierarchies of nodes might be contained in that *Editor* container. Edges may run between any two nodes, and both nodes and edges may have any number of required or optional properties. Anything which is built using these abstract elements and the few aforementioned rules is a valid domain metamodel.

However in a mathematical sense, the domain metamodel is not an instance, but a subtype of the core domain metamodel. That means the element `Place` of a Petri net metamodel is not the instance of the core metamodel element `Node` rather its subtype. It makes sense, since the concept *Place* is not a node itself, but it is a special kind of node. So strictly mathematically, the core metamodel and domain metamodels are at the same meta-level, but they describe domain specific concepts at a different level of detail.

On figure 4.4, an example domain metamodel can be seen, a simplified metamodel for Petri nets. The names inside the square brackets give the core metamodel supertypes of each element. There is one *Editor* on the top, and the only topmost nodes may be Petri nets. Each Petri net may contain *Places* and *Transitions*. Places may contain *Tokens* and may have a property called *capacity*. There are two kind of edges, *OutArc*, which goes from a Place to a Transitions, whereas *InArc* is the opposite, and edge from a Transition to a Place. Both kind of edges may have a property entitled *weight*.

Summary The core domain metamodel defines basic concepts about abstract syntax, the domain metamodels refine these concepts to create domain specific elements, finally domain specific models are simply instances of the domain metamodel.

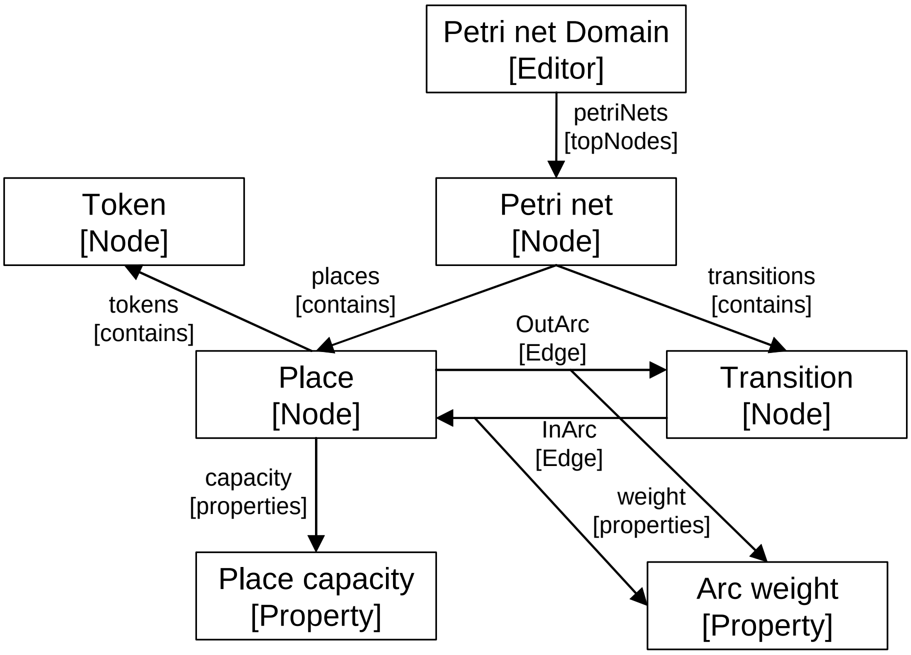


Figure 4.4: Example of a domain metamodel (Petri net domain)

4.1.4 Diagrams

Diagrams are graphical representations of certain domain specific model elements. However diagrams themselves can be regarded as some kind of domain specific models in the diagram domain. For example, if we design a Petri net, that will be a model in the Petri net domain, however if we draw that Petri net using circles, rectangles and arrow, that picture will be a model in the Petri net diagram domain. There is a *one-way mapping* between domain specific models and domain specific diagrams. Every diagram defines a model (or model part) by itself, however a model does not necessarily define a diagram. In other words, diagrams are projections of the logical model.

The ViatraDSM framework supports such separation of models and diagrams. Each domain can have its own diagrams, each domain specific diagram is described by a domain diagram metamodel. One domain may contain multiple diagram metamodels, which is helpful, when a given model can be depicted in various formats. For example in the domain of UML, the same complex UML model might be represented as a class diagram, a sequence diagram, or maybe some other format. With multiple diagram metamodels for each domain, creating various kinds of diagrams from the same model is a straightforward job.

In Chapter 6, the separation of diagrams and logical models is described in more detail.

Diagram metamodel structure

A domain specific diagram is described by a domain diagram metamodel. Such a metamodel defines the structure of a diagram, describes what kind of model elements can be displayed on the particular diagram, and how those elements should be displayed.

For example, a diagram metamodel for a UML class diagram could state that this kind of diagram may display classes, associations and class fields in a structure, where fields are displayed within the classes and associations are displayed as some kind of connection between classes. It

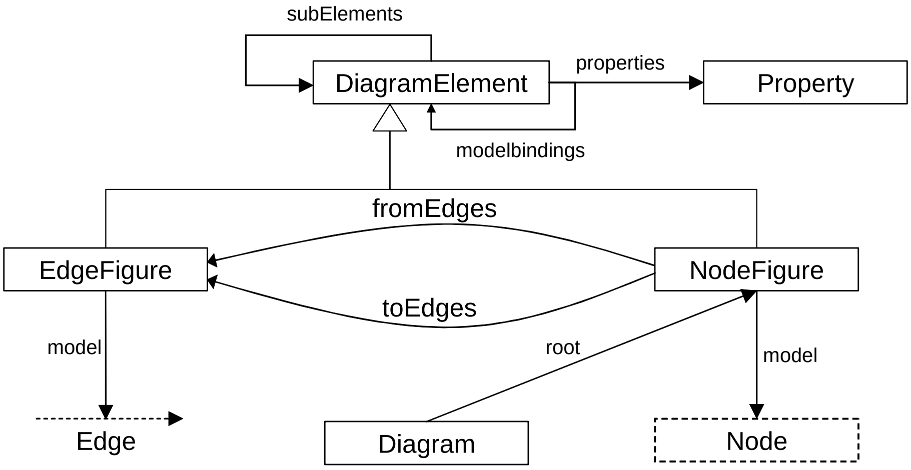


Figure 4.5: Diagram metamodel of the DSM framework.

is important to note that a diagram metamodel does not say anything about the actual graphical representation of the model element. In the case of the example above, it does not say that classes are boxes and fields are lines of text, it just states that fields are displayed within the classes.

In order to support various domain diagram metamodels, some basic concepts have to be defined, out of which diagram metamodels may be constructed. That set of basic concepts is called the core diagram metamodel (see Fig. 4.5).

Diagram is the element which represents a single diagram in the ViatraDSM framework. It is not bound to any model objects, it just joins the elements of the diagram into a single entity. It is the graphical representation of the diagram as a whole.

DiagramElement represents a graphical element on the diagram. It has two subtypes, *NodeFigure* and *EdgeFigure*. A DiagramElement may contain subelements, and have an arbitrary number of properties. Diagram properties are always stored as string key-value pairs, because it depends on the plugin-specific implementation how they will be used in the graphical rendering process.

EdgeFigures are the graphical representation of edges in the domain specific model. They may contain a reference (the relation *model*) to the *Edge* model object they represent.

NodeFigures represent the nodes of the model. Just like *EdgeFigures*, they may also linked to the model object they represent. Generally (but not necessarily) the children of a *NodeFigure* are the figures representing the children of its model object. Additionally, NodeFigures reference their EdgeFigures throgh two relations (*fromEdges* and *toEdges*).

The *root* relation Every Diagram has a root element, which is displayed as the Diagram's background. This root element can be changed to support hierarchical zooming. For more details, see 4.1.4.

The special role of the *modelbindigs* relation The *modelbindings* relation is a special feature of the diagram metamodel. Since not all DiagramElements (NodeFigures, EdgeFigures) necessarily contain references to logical model elements, the ViatraDSM framework supports DiagramElements without logical model bindings (e.g. decorators, notes, etc). However, there may be cases where such an element is *indirectly* connected to a logical element, through another DiagramElement which has a direct model reference. For instance, the designer may wish to assign several floating text labels to a Petri net Place, which display the various static and dynamic properties of that place (capacity, number of tokens), and can be edited and moved around freely. Such a floating text label requires a separate diagram metamodel element (because, on the GEF level, a separate EditPart is needed), however it has no direct model binding. In such cases, a *modelbindings* relation should be added between the NodeFigure of the place and the NodeFigure of the floating text label, which the framework uses to determine which place should be involved in the modification of the logical model once something on the diagram is modified.

Note: strictly speaking, using this technique is *optional*. If the tool designer wishes to implement diagram-to-logical model mapping using a custom mapper interfaces implementation (e.g. by manually coding the relevant functions of the plugin), these modelbindings relations can be omitted. The *modelbindings* relation is only needed if there is no connection between the "modelless" DiagramElement and its "model-bound" companion that is visible on the diagram model level and the plugin designer wishes to use the built-in diagram-to-logical model mapping functionality of the ViatraDSM framework.

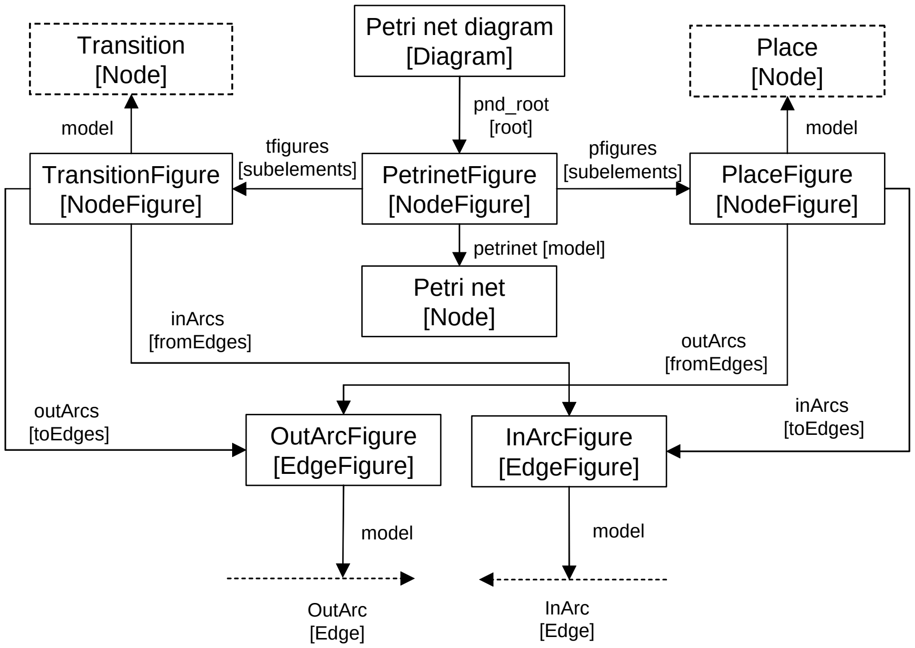


Figure 4.6: Example of a diagram metamodel (Petri net domain)

On figure 4.6 an example of a domain specific diagram metamodel can be seen. We can see that there is one logical *Diagram* entity, and one default root node, which serves as a container for graphical elements. There are two kinds of *NodeFigures*, one for the places and the other for the transition. In the case of edges, *InArcs* and *OutArcs* are represented with different figures.

One might notice the absence of the model element *Token* from this diagram metamodel. This

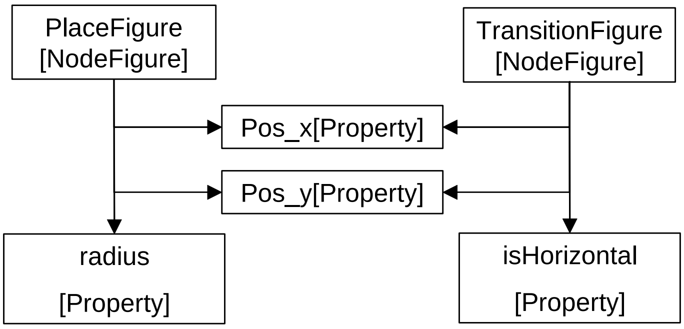


Figure 4.7: Petri net domain diagram metamodel - properties

is because in this particular example, Tokens have no direct graphical representation, they will appear as numbers written inside the Place's figure.

In this particular example, three diagram properties are defined (Fig 4.7): position coordinates (x,y) for both transitions and places, a radius for places, and an orientation for transitions (which determines whether the particular TransitionFigure instance should behave as a horizontal or vertical rectangular figure).

Changeable root concept

Model elements on a complex, highly hierarchical diagram can be very difficult to find. Logical zooming enables the user to ”zoom into” any selected node, and display only the contents of that node. It is similar to the current directory concept of file systems. Without logical zoom, it would be hardly possible to find anything on a diagram containing hundreds of elements.

In the diagram metamodel (Fig. 4.5) the relation *root* always points to the actual node the user has zoomed into. The reason why this relation is included in the metamodel (compared to being stored in some local variable during editing) is that when a diagram is saved, we would like to retain that zooming information. Therefore if the users saves a diagram and loads it back later on, the zoom settings will not be lost.

4.2 Implementation

In this section, I will introduce the concept of a *domain plugin*, and show how the ViatraDSM framework and these domain plugins interact to form a multi domain editor. Following that, the default implementation of a domain plugin will be discussed.

4.2.1 DSM framework and domain plugins

The ViatraDSM framework is built on top of Eclipse, it is an Eclipse plugin, which implements a multi page editor. A multi page editor is one of the default editors of Eclipse, it is basically a container for multiple editors, its actual appearance will be shown in section 4.4. In this case, the different pages of that container will correspond to the editors of the different domains. A multi domain editor still separates the domains at the editing phase, multi domain functionality only means that multi domain diagrams may be created, and multi domain transformations may be executed.

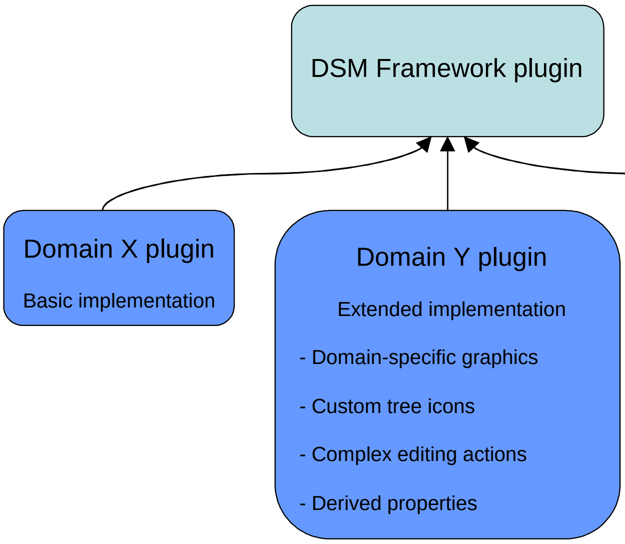


Figure 4.8: Plugin-oriented architecture of the DSM framework

In order to support multiple domains at the same time, the ViatraDSM framework must have a mechanism to detect what are the domains, which are present. When the user starts our DSM framework, it should somehow find information about the available domains. Our solution for that problem was a plugin-oriented architecture. Our DSM framework itself does not know about any domains, but separate domains can make the framework aware of themselves by connecting to the framework and providing information. A *domain plugin* is an Eclipse plugin, which implements a certain domain. So when the framework starts, it does nothing, but waits for separate domain plugins to log in. Figure 4.8 shows this plugin-oriented architecture.

A domain plugin may log in to the DSM framework by providing four different pieces of information. First, it must supply the domain metamodel. Second, a domain plugin must also provide the root of the actual domain specific model. Each domain has a single root element, from which every other model element of the domain is accessible by navigating through containment relations, edges and properties. Third, a domain plugin must also provide a list of domain diagram metamodels, and the available diagrams (diagram models), which correspond to these metamodels. Finally, the domain plugin must provide its own domain specific graphical editor, since our framework only support a simple, tree-structured visualization of models.

The ViatraDSM framework contains a default domain plugin implementation, which uses VIATRA2 as a model (and metamodel) container and also as a diagram model (and metamodel) container. It also provides a GEF-based editor with some basic graphical figures (rectangular boxes and arrows). Generally a language engineer does not need to create the entire domain plugin himself, but can use our extensible implementation.

As it can be seen on figure 4.8, the DSM framework can be extended to provide custom icons in the tree view, add optional, derived properties, and so on. Our default domain plugin implementation will use built-in icons, no additional properties, and therefore even if a language engineer does not want to code, he can create a functional domain plugin. But since virtually every small piece of our default implementation may be changed by the language engineer (or even a language engineer may create his own domain plugin from scratch), our framework is completely customizable.

4.2.2 VIATRA2 as the model container

In order for the ViatraDSM framework to function properly, both the core metamodels (both domain and diagram) and the metamodel of the used domains must be present in the initial model space. The models being edited are also stored within the same model space. The first implementation problem we solved was how to map VIATRA2 model objects to core metamodel object, and how to navigate in a domain specific model.

Thin wrapper implementation

Using VIATRA2 references has multiple problems, most importantly because of the lack of type-safety. Copying the entire domain specific model using custom (type-safe) classes raises memory usage and synchronization problems. We decided to go for the middle, and created an implementation, which combines the two, a type-safe implementation which does not create a copy of the model. We call this approach *thin wrapper*. The idea was that we should create interfaces for each core metamodel entity, but the implementation of these interfaces does not copy the model (thus being thin), rather it stores a reference to a VIATRA2 model object (thus being a wrapper), and uses a mapper utility to extract information upon queries. So we have interfaces like `IDSMNode` or `IDSMEdge`, and the implementation of these interfaces stores a VIATRA2 reference. When the `getContainedNodes()` method is called upon an `IDSMNode` object, the implementation uses a mapper utility to extract the information from the VIATRA2 model space.

Using these interfaces introduces type-safety to the implementation, and thus the mapper utility does not have to check the validity of the queries. We moved those checks from run-time to compile-time. This approach also allows us to deal with models outside the VIATRA2 model space, since these core metamodel interfaces might be implemented by a third party to support EMF models for example.

One drawback of this thin wrapper implementation is that it still uses a mapper utility. A lot of processing time is spared due to type-safety, but a thin wrapper still lags behind complete model copy in terms of speed. A function which queries the VIATRA2 model space could not possibly be as fast as just checking a few local variables. We tried a few optimization tricks, the most promising was *metamodel caching*. It is sure that the entire domain specific model cannot be copied, such a model might contain hundreds of complete Petri nets for example. However we may copy the domain metamodel. No matter how large the model becomes, the metamodel remains the same (generally small) size. Why would that increase the performance when the queries are run on model element? Simply because most model queries begin with a metamodel query as a first step. For example, if we want to read all the children nodes of a Petri net instance, first we have to read the metamodel to find out that a Petri net has two kinds of children, places and transitions. Without optimization, reading the children of a node requires two queries into the VIATRA2 model space, one about the metamodel and a second about the model itself. If we cache the metamodel within our DSM framework, we can spare one of those queries. Caching the metamodel requires a negligible memory compared to model sizes, but the speed gain approximates twofold.

Notification support

Using a thin wrapper with metamodel caching is a rather efficient way of navigating in the model along core metamodel relations. However, as it can be seen on figure 2.4, in a GEF-based editor, the model must be capable of notifying its EditPart. These notification messages are sent to the EditPart when something about the model changes (eg. a new place is added to the Petri net or simply one of the edge weights are changed). This notification mechanism is necessary because the model can be modified not only by the user, but by a transformation (simulation) run by the VIATRA2 engine.

VIATRA2 has built-in notification support, so for every VIATRA2 model object (`IModelElement`) listeners can be registered. The registered listeners will be notified when that model object is changed.

The major difficulty in implementing this notification mechanism was that one domain specific element (eg. a node) is made up of multiple VIATRA2 model elements. Thus an event at DSM-level (eg. a new child node has been created) are made up by a sequence of multiple VIATRA2-level notifications. The creation of a new child node for example requires four basic VIATRA2 operations to be carried out in a defined order.

Our solution was very similar to the thin wrapper conception. We created two notification dispatcher classes (for the diagram and logical models - represented by the “Notification mapping” component of the Thin Wrappers on Fig. 4.8), which have the sole responsibility of turning VIATRA2 notifications to DSM notifications (like the thin wrapper turned domain specific queries into VIATRA2-specific ones). We defined a pattern for each DSM notification. Such a pattern describes what VIATRA2 notifications should occur and in what order to induce a DSM event. An example for such a pattern could be: if a new entity (at VPM-level) is created, then it becomes instance of a domain metamodel element, and then a relation is created linking this entity to another model object of the same domain, this sequence means that at DSM-level a new child node has been created. The notification dispatcher listens to every VIATRA2 model objects in a given domain, and checks the incoming VIATRA2notification sequence for these patterns, and when it finds one, it emits the corresponding DSM notification.

Extensibility

In the previous paragraphs I summarized how the handling of domain specific models can be performed when those models are stored within the VIATRA2 model space. However, the ViatraDSM framework is *not* bound to VIATRA2 . Since the model space is accessed through a thin wrapper layer, an EMF-based (or even a relational database backed-based) implementation would be possible.

Implementing these domain specific model interfaces requires lots of work and great care nevertheless. Basically there are four major components to implement. First, metamodel and model query functions (get children nodes, and others) must be implemented. Second, model manipulation function (add new child node, etc.) must also be implemented. Third, notification support required by GEF EditParts must be added to model objects. Finally, a simple domain manager class should be implemented. That domain manager is responsible for initializing the domain model, finding the model and metamodel root elements and coordinating the work of the other three components.

Model representation summary

For the representation of metamodels and models, the ViatraDSM framework gives complete freedom to the language engineer. The framework accesses and modifies the models through a well-defined set of interfaces. Every domain plugin must contain an implementation of these interfaces. A language engineer has the freedom to implement these interfaces himself, thus completely hiding the metamodel and model representation from the DSM framework.

However, in most of the cases, the default implementation can be used (subclassed), which uses the VIATRA2 model space for storing metamodels and models. This VIATRA2 implementation tries to be both fast and memory-saving, therefore it uses a *thin wrapper* conception, which combines the low memory usage of the wrapper-less approach (using VIATRA2 reference inside the DSM framework) and the fast processing speed of ”fat” wrappers (copying the entire domain specific model). Our implementation also supports the notification mechanism required by GEF by converting incoming VIATRA2 notifications to DSM-specific ones based on a few simple rules. Both the thin wrapper and the notification dispatcher includes certain optimization techniques which enable a smoother run of complex transformations and simulations.

4.2.3 Graphical representation

The default domain plugin implementation also provides functions for graphical model representation and editing. As a domain specific graphical editor, it provides a GEF-based extensible editor, where actual graphical figures may be easily replaced by any custom figure.

VIATRA2 as the diagram container

The diagram models are also accessed through a similar thin wrapper layer like the logical models. The main advantage of this approach is that this enables VIATRA2 transformations to run on diagram models (making, for instance, transformation-based mapping between the diagram and its logical counterpart possible - for details, see 6).

GEF-based extensible editor

Our implementation includes a GEF-based (see 2.5.3) editor to edit these domain specific diagrams. In the MVC scheme of GEF, the model may be any arbitrary Java class, the only requirement is that model change notifications should be supported. The views are required to be Draw2D figures, whereas the controller, called EditPart, is supplied by GEF. These GEF EditParts contain the essence of GEF functionality, they transform basic user actions (keyboard and mouse events) to a higher abstraction level, called requests, which are then transformed again to commands, which are the placeholders for model manipulation activities.

Basic view implementation The ViatraDSM framework provides basic view class for NodeFigures and EdgeFigures (and a white canvas as the default diagram root). Our *NodeFigure* is a resizable rectangular box with the name of the represented model element in the header. The basic *EdgeFigure* is a simple arrow leading between nodes. You can see screenshots of these figures in section 4.4.

Extensibility

The various domain plugins may connect to the framework through various Java interfaces. They must implement the `IViatraDSMEditor` interface, which inherits methods from the following interfaces:

- `IDSMOutlineContributor` This interface is responsible for supporting various ways to customize the appearance of the tree view (tree labels, tree icons, tree color, tree comparator for a custom ordering of objects, DSM notification support, and plugins can even contribute to the tree's context menu).
- `IDSMPropertyContributor` Plugins may provide custom Property Descriptors for various logical and diagram properties through this interface. This is useful if a special editor (such as a File selector, for instance) is required for a given property.
- `IDSMDomainContributor` Plugins provide basic information about the domain they are attached to through this interface. Additionally, the simulation/code generation engine of the ViatraDSM framework uses this interface to determine what kind of simulation transformations/code generators are provided by the plugin.
- `IDSMGraphicalEditingContributor` This is the interface though which plugins provide helper classes for graphical editing. The most important of these is the `IViewFactory` instance. Additionally, plugins may provide custom creation tool icons which are displayed on the graphical palette.
- `IEditorPart` This is a standard interface of the Eclipse workbench, every editor instance must implement this.

Additional methods defined on `IViatraDSMEditor` provide support for the handling of various workbench events, such as selection changed, focus gained and lost, save initiated.

The `IViewFactory` interface provides support for the creation of view classes based on the metamodel specification (i.e. the ViatraDSM framework passes a metamodel instance to the plugin, and gets a view class in return).

4.3 Transformations, simulation and code generation

An essential concept in our DSM framework is the support for transformations, both model transformations and code generators. Basically every existing DSM tool provides code generation functions, but only a few support model transformations. The advantage of model transformations is that using them, the models can be simulated with the DSM tool. Simulations is just one use of model transformation (however the most frequent use), but model transformations are more powerful than just a plain simulation engine. Using model transformations, automated model manipulation becomes possible. An example could be automated optimization, which we have already mentioned as an example in the introduction of this chapter.

4.3.1 Requirements of transformation support

The ViatraDSM framework has a built-in transformation engine, which utilizes the facilities of the underlying VIATRA2 framework. This transformation engine was developed by Dávid Vágó, and thus in this section I only outline the various possibilities of model simulation (and code generation). On the user interface, simulation execution appears as a separate element in the tree's context menu; it supports interactive simulation (meaning that the user can determine on non-deterministic points how to continue) and diagram can track the state of the simulation engine as well.

4.3.2 Describing transformations

Since the VIATRA2 framework is used for model transformation, simulation has to be specified by graph transformation rules and abstract state machine programs. These technologies have already been introduced in chapter 3. Transformations may be divided into two groups, the ones which does not affect the model structure, and the ones which effectually modify the model. The first group contains mostly code generators, but model checkers also belong there. Code generators are transformations, which simply walk through the entire model, and at each model element, they emit some kind of textual information. Most probably at the end of the round trip, these snippets of texts assemble into some kind of meaningful source code of some kind. See 7 for an example simulator, and 7 for an example code generator, a transformation which generates PNML code from a Petri net model.

Model checkers are transformations, whose task is to check the fulfillment of certain wellformedness rules. Certain wellformedness rules be expressed at metamodeling level (for example our metamodel of Petri nets contained the wellformedness rule that two places may not be connected), but certain others cannot (or would require heavy modification of the metamodel). To check complex wellformedness rules, we can design a transformation, which walks throught the entire model, just like a code generator, and checks the given rules for every model element. One example for wellformedness rules, which may not (in our actual implementation) be expressed in metamodels are relation multiplicities. If we want to introduce multiplicity constraints in our domain specific editor, we must build a transformation, which check the entire model, and verifies multiplicities.

4.3.3 Running transformations

The ViatraDSM framework supports the possibility to assign certain transformations to model elements, and thus the execution of these transformations may be initiated from the user interface. The framework currently provides support for two kinds of model transformations: simulation and the checking of language-specific well-formedness constraints.

The results of a model-checking transformation can be observed in Eclipse's Error Log view (as supported by the VIATRA2 framework), while simulations can be tracked using ViatraDSM's user interface (as supported by the notification mechanism). Figures 4.9-4.11 show the three successive steps in an example Petri net simulation.

The framework also provides support for influencing the execution of the simulation. For example in the Petri net example, if we simply launch the simulation, we have no means to decide whether the top or the bottom transition will fire. To provide complex debugging services, we provide a mechanism which enables the user to choose any arbitrary execution path himself whenever the simulation arrives at a decision point.

On figure 4.12 we give an example of how such a guided simulation is executed. It shows the first step of a Petri net simulation, where the possible execution path (fireable transitions) are marked with red, and the user could choose the desired one by simple clicking on the chosen one.

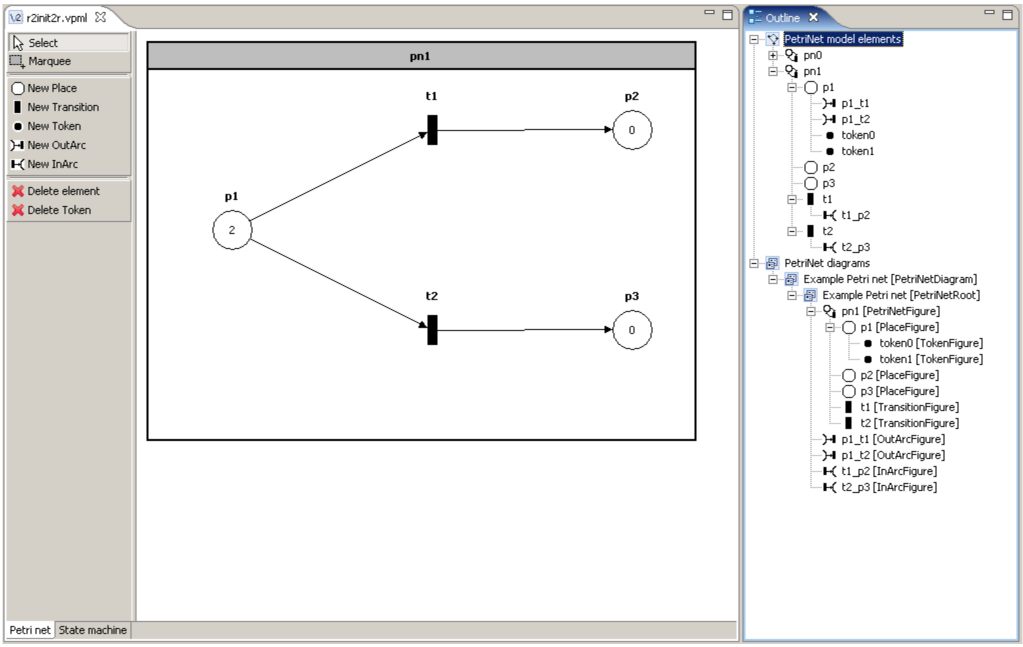


Figure 4.9: Three steps of a Petri net simulation in the DSM framework (step 1)

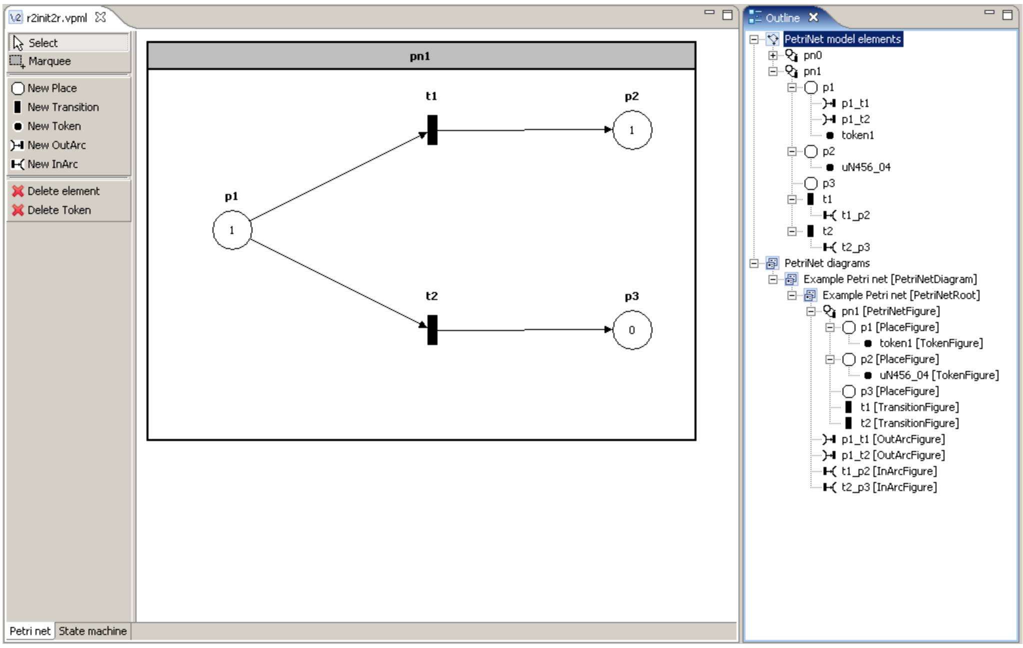


Figure 4.10: Three steps of a Petri net simulation in the DSM framework (step 2)

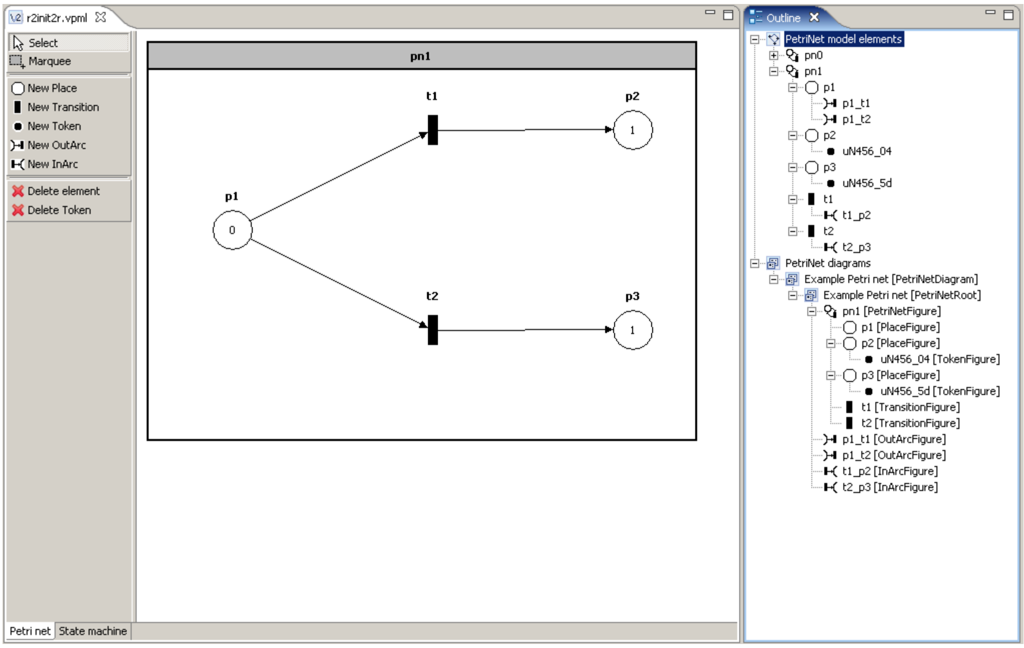


Figure 4.11: Three steps of a Petri net simulation in the DSM framework (step 3)

4.4 User Interface

The user interface of a modeling tool is a key point in success, since developer productivity can only be increased, if the GUI is simple, clear and easy to use.

On figure 4.13 the entire DSM framework is shown as it appears inside the Eclipse platform. The appearance of our framework can be divided into three major components. In the middle we see the graphical representation of the domain specific model. On the right, in an Eclipse view entitled *Outline* we see a tree-structured representation of our model and our diagrams. Finally at the bottom, we see the Eclipse *Properties* view, where model and diagram properties may be changed.

Another important part are the little tabs at the bottom of the graphical editor. There are two tabs there on this picture, *Petri net* and *State machine*. These are the domain, which are created within the DSM framework. Unlike most of the existing DSM tools, our framework supports multi domain editing (see 5), where models from different domain can be edited at the same time, multi domain diagrams can be drawn or heterogeneous systems may be simulated. With those little tabs we may instantly jump from one domain to another. User-initiated editing operations do not affect domains other than the selected one, but multi domain transformations may cause model from multiple domain to change.

4.4.1 Logical model view

The logical model in can be found in the *Outline* view in the form of a tree. The tree view has two roots, one for the logical modelspace and one for the diagrams (which can be viewed, but not edited).

Every item in the tree represents one model element. Each node is contained by its parent node, and edges are contained by their source node. In the example on figure 4.17 we have two

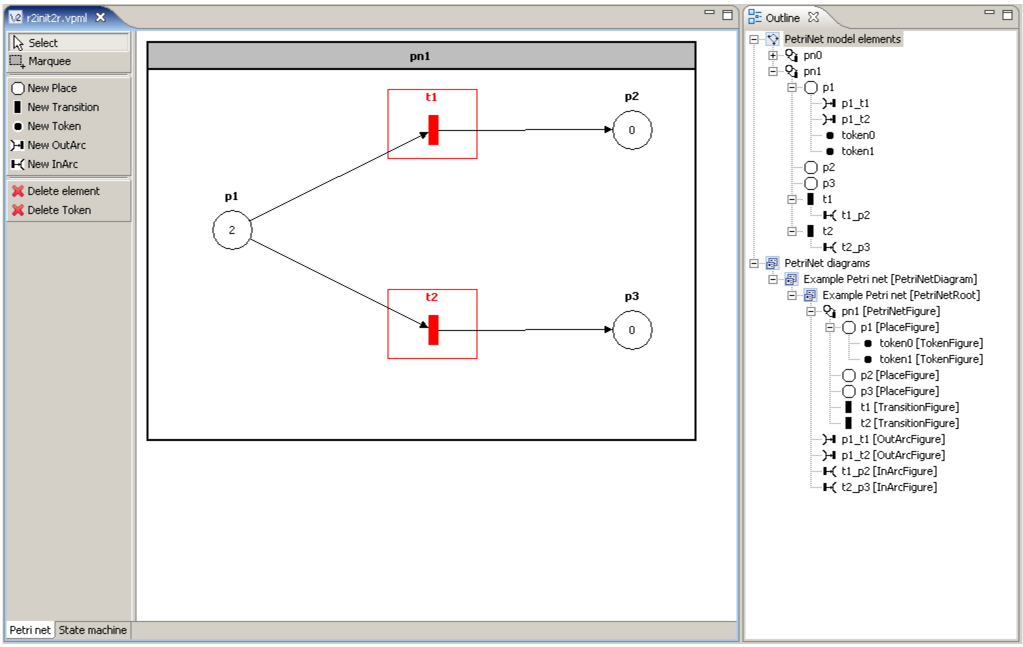


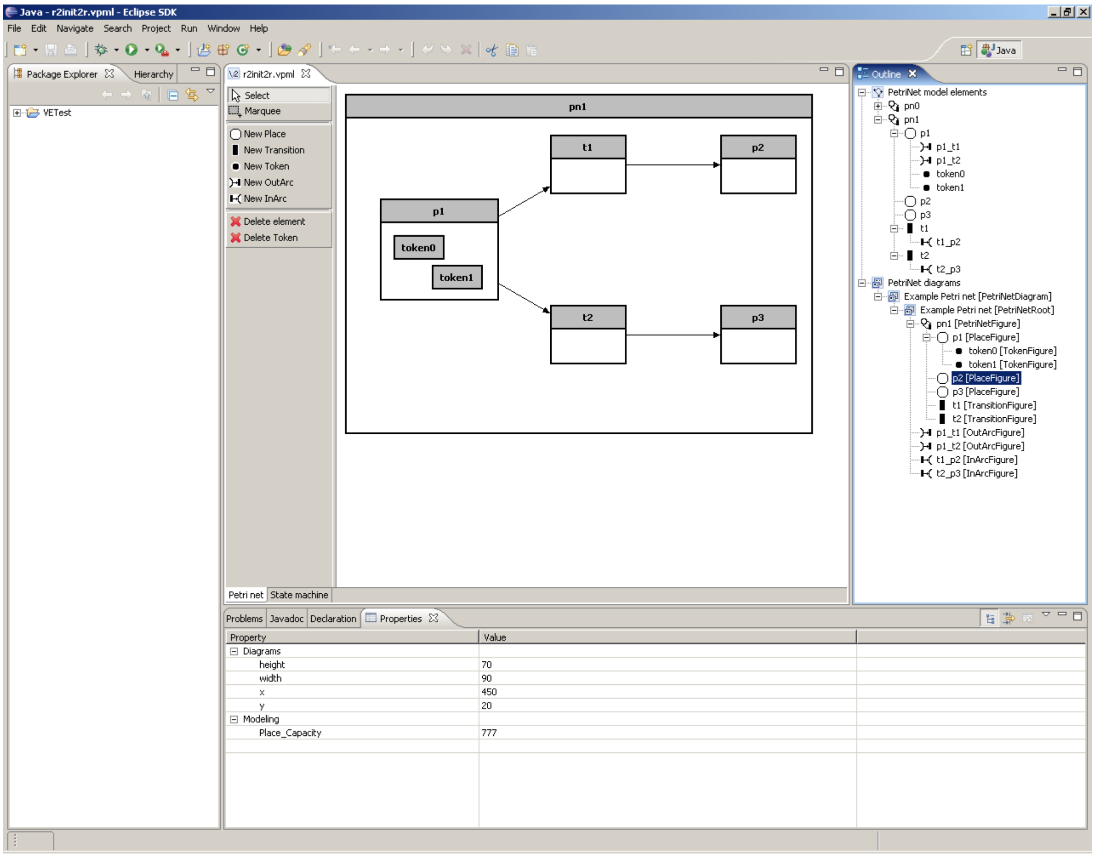
Figure 4.12: Guided simulation of Petri nets

Petri nets (*pn0* and *pn1*), and the latter has three places, two tokens (both in place *p1*) and two transition and some arcs linking them together.

To edit the model in the tree view, the user has to use the context menu. Figure 4.14 shows two examples, the left one appears, when a Petri net is selected, the right one appears, when a place is selected. The context menu will contain only the appropriate actions for the selected element. You can see on the example, that Petri nets have places and transition as possible children, whereas places may contain tokens or outarcs (this is called *syntax-driven editing*).

For the creation and deletion of model elements, the user may use the context menu (actually deletion also works with the *Del* key), but it is also possible to move elements from their parent into an other element. That can be done by simple drag-and-drop, the framework will check whether the operation is valid (eg. dragging tokens into a transition would be invalid). Clipboard operations (copy, cut and paste) are also supported, and every action (or action sequence), which modifies the model may be undone step by step. Each supported domain has its own clipboard and undo stack, therefore the user may comfortably work with multiple domains at the same time. Undoing transformations (thus also simulations) is supported, too. The undo and redo operations, along with the clipboard functions are available through the Eclipse toolbar (see figure 4.15).

An other important feature in terms of editing is the property sheet (figure 4.16). This view is provided by the Eclipse platform, and editors and other views may populate it with any set of properties. As it can be seen in the example, it displays both modeling and diagram properties. Every *Property* in our domain metamodel will be displayed on this property sheet, and the value (which is in general an ordinary text value) can be edited there. Naturally, property change actions can also be undone. For edges, the source and target endpoints may also be edited through the property sheet (the other way of changing them is drag-and-drop in the tree view).



Property	Value
Diagrams	
height	70
width	90
x	450
y	20
Modeling	
Place_Capacity	777

Figure 4.13: The graphical user interface of the ViatraDSM framework

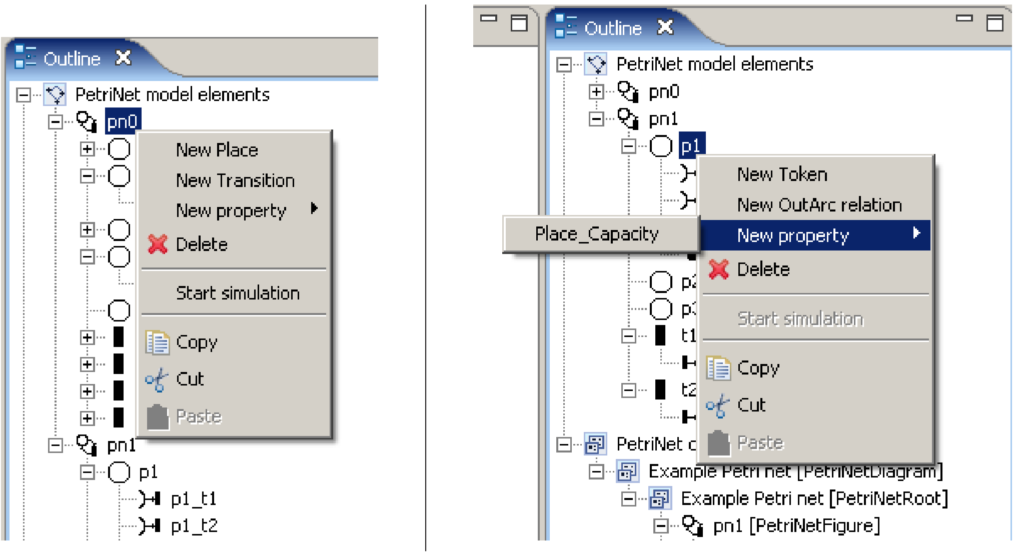


Figure 4.14: Example of context menus (left: PetriNet, right: Place)

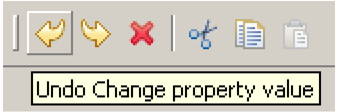
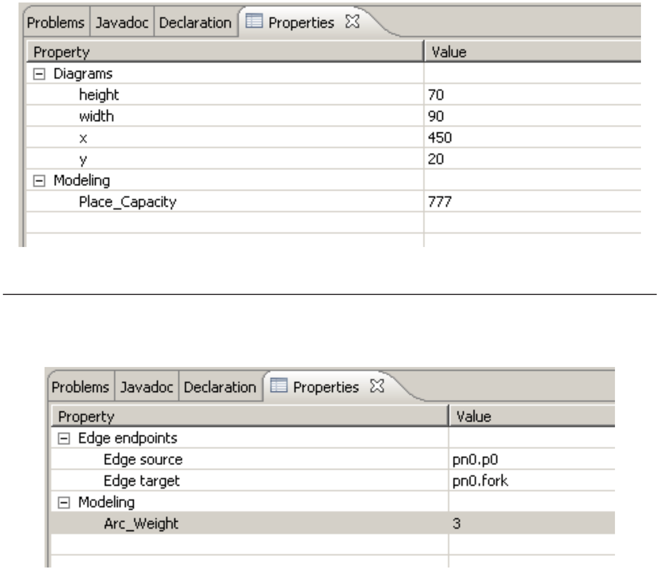


Figure 4.15: Toolbar for undo/redo, delete and clipboard operations.



Property	Value
⊟ Diagrams	
height	70
width	90
x	450
y	20
⊟ Modeling	
Place_Capacity	777

Property	Value
⊟ Edge endpoints	
Edge source	pn0.p0
Edge target	pn0.fork
⊟ Modeling	
Arc_Weight	3

Figure 4.16: Property sheet views (top: Place, bottom: OutArc)

4.4.2 Diagrams

A developer may access every possible editing action through the property sheet and the tree view, but diagrams offer a more spectacular way of modifying the model. An example graphical editor is shown on figure 4.17. On the left it contains a palette, which list the possible graphical editing actions. Its main part is the actual graphical display of certain model elements.

The ViatraDSM framework (like many other modeling tools) *separates the concepts of model and diagrams*. The model includes every element, which is present in our domain, whereas a diagram only displays a subset of these elements. This introduces certain ambiguity, for instance, when the user deletes something on the graphical view, the question arises whether it should be only removed from the diagram, or also deleted from the model. Despite this small problem, separating the model and the diagrams is a good idea, since it lets users show only parts of the model. A complex web-based application might be modeled with a statechart, which contains thousands of states and transitions. If that entire statechart were represented as one single diagram, it would quite confusing to a designer (we have had plenty of feedback regarding this problem for the first VIATRA2 editor, which was also written by Dávid Vágó and myself). Separating the model from the diagrams enables to have a large, complex model, but create diagrams only of certain small submodels, in case of our web application example, the few dozen states, which are responsible for user authentication for example.

Graphical diagrams are also displayed in the bottom half of the tree view. The main motive behind this is that graphical elements can be found easier (especially considering the changeable root functionality) in a tree layout, that on a visual diagram. The property sheet can also be useful in graphical editing, because for example using the Property view the user may precisely position and resize diagram objects (seee Fig. 4.16).

A more detailed elaboration on Diagrams, and the separation between diagrams and logical models can be found in Chapter 6.

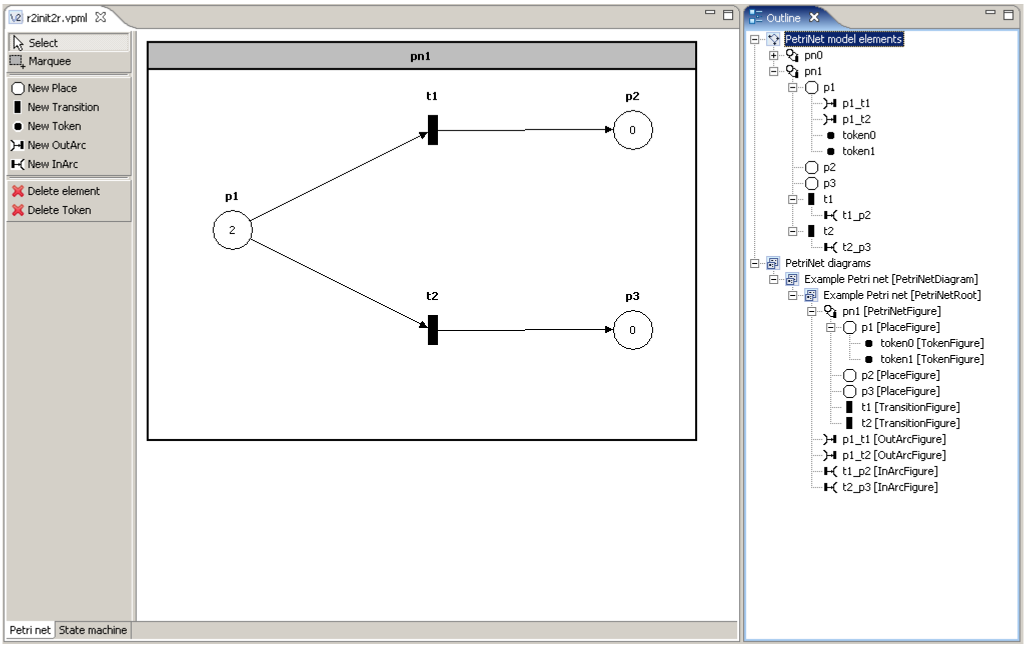


Figure 4.17: Petri net editor with custom graphical elements

Chapter 5

Multi domain modeling

In this chapter, I discuss the ViatraDSM framework's novel approach for integrating domain-specific modeling languages across multiple domains. First, I describe (in Sec. 5.4) light-weight techniques for integrating multiple domain-specific modeling languages into a consistent system model in addition to traditional model transformation based domain integration. Using these novel techniques, integration of DSMLs can be specified very easily compared with designing a complex model transformation for the same problem in many cases. Then I present (in Sec. 5.5) how these concepts can be used to create multi-domain models in the ViatraDSM framework.

5.1 Introduction

The ViatraDSM framework supports the multi-domain integration of domain-specific modeling languages (DSMLs) in the following ways: (i) *subclassing* between metamodel elements of two DSMLs, which is an extension of the UML profiling mechanism for an arbitrary "host" DSML (not only UML); (ii) *multiple instantiation (typing)* [46] where a model element may be typed over multiple domain metamodels; and (iii) traditional *model transformations* as provided by the VIATRA2 transformation framework. For all these solutions, the models of the different DSMLs can be edited separately by the domain engineers, however, the underlying system model is kept synchronized.

5.2 Domain integration

Complex systems are usually modeled from different perspectives, for example, UML offers several diagram types for software design, e.g. class and deployment diagrams for structural modeling, activity and sequence diagrams for dynamic behaviour etc. Thus the separation of concerns is essential in maintaining the clarity and accuracy of complex models.

The various modeling perspectives should be linked together in the underlying system model in a consistent way. This means, in simple terms, that if something is changed in one of the modeling domains (e.g. a method name is changed in a UML class diagram), this change has to be automatically reflected in all other modeling domains (e.g. by changing the name in all sequence diagrams as well).

Thus, *domain integration* means retaining the advantages offered by various domain-specific tools, and providing automated support for generating a global and coherent system model from small domain-specific submodels.

The traditional approach to domain integration is to use separate domain-specific tools for various source domains and *import* their output into the target domain (see Sec. 5.4.1). This is generally unidirectional, i.e. if a change in the target domain should be reflected in the source domain, the only option is to manually figure out what needs to be changed in the source domain and edit the models manually, or to write a second transformation.

These off-line (import/export based) transformational integration approaches can be slow, especially for large models, even if automated tool support is available. A lightweight solution is to integrate all (meta)models into a single language, and use stereotype annotation (tagging) to assign model elements to various domains (Fig. 5.1). This approach is used by the profiling mechanism of UML. Whilst some tool support exists for defining and applying UML profiles, this technique lacks certain advantages offered by flexible domain-specific editors as it is annoyingly easy to construct ill-formed models with inappropriate stereotypes.

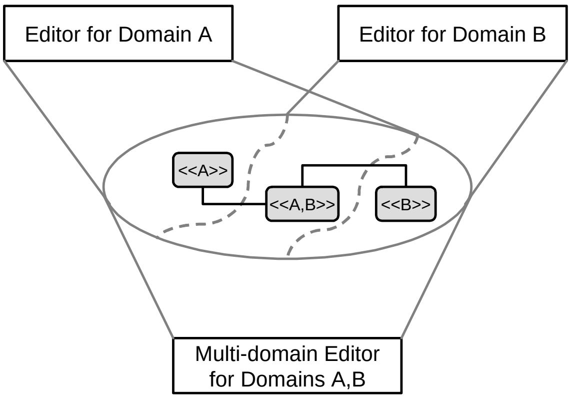


Figure 5.1: Integrated multi-domain modeling

However, the metamodels in standard UML Profiles already incorporate deep knowledge of a domain gathered from top domain experts, which is required for constructing editors and languages for that domain. Unfortunately, these existing metamodels are frequently ignored when constructing a new language in a DSM framework. In the upcoming example, I demonstrate that the metamodels of UML profiles are highly reusable when constructing domain-specific languages.

5.3 Example: Enterprise Security Policies and UML's Performance Profile

In this section, a simple example will be used to demonstrate how a custom domain-specific language can be projected into an analysis domain corresponding to the UML Performance Profile.

The domain-specific modeling language is built from scratch by using a generic modeling environment. In this case, the **Enterprise Security Policy** domain (see Fig. 5.2; which is an extended version of the example presented in [37] as a complex case study) describes a simple enterprise security and surveillance system installation and its security procedures (note: several classes have properties which are omitted in Fig. 5.2 for the sake of simplicity).

The Enterprise Security Policy domain describes both deployment and procedural aspects of enterprise surveillance systems. At the top of the containment hierarchy is the Installation entity, to which Buildings, a Network and several Processes can be assigned. A Building consists of Rooms, where Devices can be placed. Devices have deployment and operational cost properties

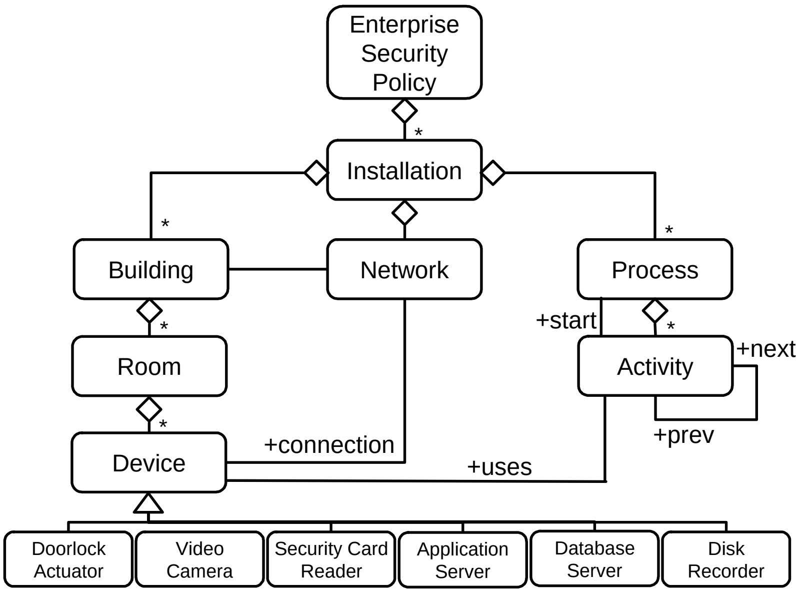


Figure 5.2: Enterprise Security Policy domain metamodel

which can be used for financial analysis and optimization.

The buildings are connected to the network infrastructure; individual devices can be assigned individual connection properties (not shown in Fig. 5.2). Security policy Processes consist of Activities, which can make use of several devices.

Since the application domain is capable of describing processes, it is a natural requirement to analyze the performance and throughput of the system in a production environment. For modeling performance related aspects, we use the standard **UML Performance profile**[34]. As a consequence, the modeling environment should be extended to allow the projection of security policy models into the UML Performance domain (Fig. 5.3).

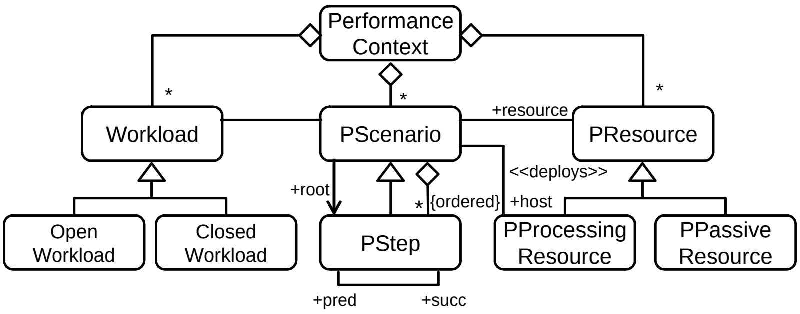


Figure 5.3: UML Performance domain metamodel (extract)

The UML Performance domain describes Performance Contexts, which consist of Workloads, Performance Scenarios, and Performance Resources. A scenario is made up of multiple, ordered Performance Steps, which can make use of either Passive, or Processing resources. A scenario can be analyzed with either Open, or Closed workloads.

5.4 Concepts for multi-domain integration

In this section, I discuss three modeling techniques for multi-domain modeling integration based on (i) *subclassing* between metamodel elements of two languages, which is an extension of the UML profiling mechanism for arbitrary ”host” languages (and not only UML); (ii) *multiple instantiation (typing)* where a model element may be typed over multiple domain metamodels; and (iii) traditional *model transformations*. In all these cases, the models of the different DSMLs can be constructed separately by the domain engineers, however, the underlying system model is kept integrated automatically by appropriate mechanisms.

5.4.1 Transformation-based integration

Transformation-based domain integration (Fig. 5.4) means that models from different domains are mapped using model transformation techniques. In the general case, such a transformation takes a valid model of the source domain, and produces its counterpart in the target domain. The equivalence of the source and target models are preferably guaranteed by the formal verification of the transformation.

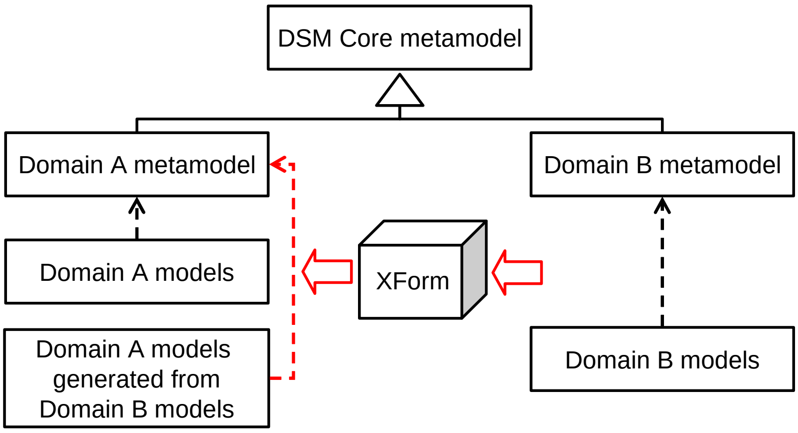


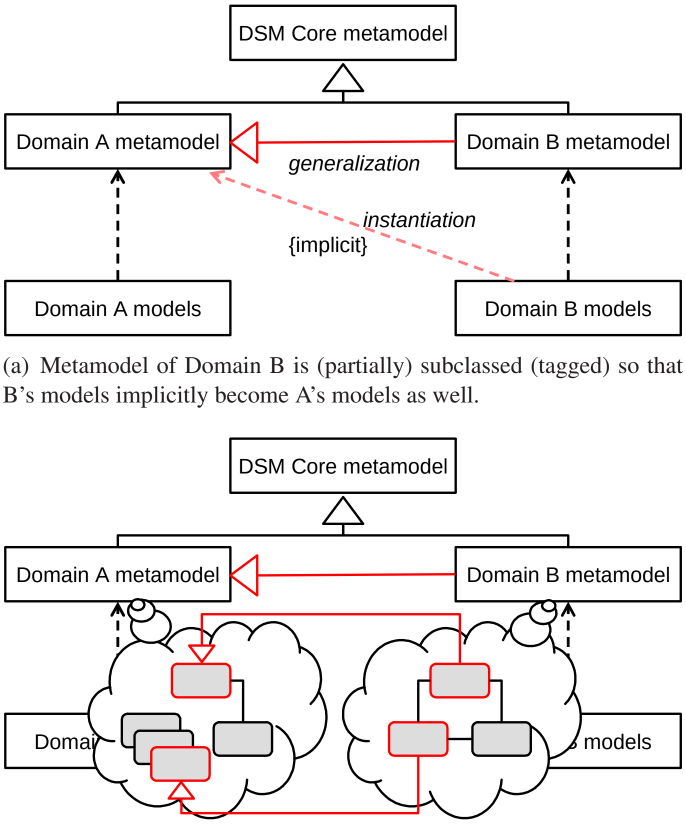
Figure 5.4: Transformation-based integration

The transformation-based integration is the most generic approach to multi-domain modeling. Moreover, bidirectional translations are also possible in certain cases. However, for complex models, regenerating the whole target model after a small change in the source model can significantly slow down the design.

Therefore, the main challenge of this approach is to provide support for *incremental* transformations which run transparently in the background while the user is editing the model. While incremental transformations are of immense importance in the upcoming QVT standard, graph transformation-based approaches provide much better support for integrating multiple domains or views.

5.4.2 Metamodel-level integration by subclassing

However, in many cases, the transformation-based approach can be considered too heavyweight concerning the amount of work required for specifying the links between two domains. This is particularly true when the source and the target domain metamodels are structurally ’similar’. This similarity can be expressed with a (partial) *subclass* relation between the domain metamodels (Fig. 5.5).



(a) Metamodel of Domain B is (partially) subclassed (tagged) so that B's models implicitly become A's models as well.

(b) Partial subclassing (tagging) means that not every metamodel element in Domain B is required to be tagged with a supertype from the metamodel of Domain A.

Figure 5.5: Metamodel-level integration (subclassing)

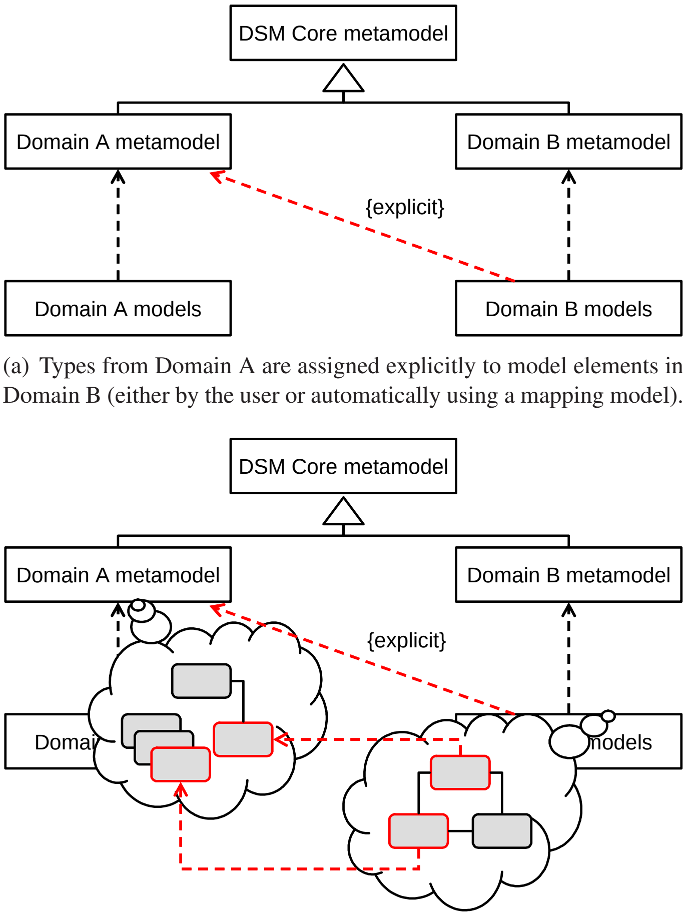
This technique is a generalization of UML profiling mechanism for an arbitrary ”host” domain [46]. In Fig. 5.5, element ”tagging” means the assignment of an additional supertype to the source domain metamodel's certain elements. This is possible if the underlying metamodeling framework supports multiple inheritance.

This way, model instances of the source domain implicitly become instances of the target domain; however, care must be taken to ensure that they are *valid* instances, i.e. both static and language-specific constraints must hold in both domains. In the simplest case, this may require some further associations be added to the model (which are only required for the target domain); however, in general, this is also a model transformation problem. The key difference is that typically only simple transformations are required here which only perform ”corrections” on the models if they are not well-formed (instead of full translations as in the general case). In practice, this means that they are easier to design and implement.

5.4.3 Model-level integration by multiple instantiation

The model-level integration approach (see Fig. 5.6) requires that a class in target domain can be assigned explicitly as a type for a model element in the source domain. This assignment can be carried out by user interaction or automatically.

The key difference between the model- and metamodel-level approaches is that by mapping metamodels it is ensured that *all* model instances will be mapped onto the target domain, while



(a) Types from Domain A are assigned explicitly to model elements in Domain B (either by the user or automatically using a mapping model).

(b) This mapping can be partial, because in most cases not all elements from Domain B are relevant in Domain A.

Figure 5.6: Model-level integration (multiple instantiation)

model-level mapping is more customizable in the sense that the user decides what to project.

A major requirement for model-level domain integration against the underlying metamodeling framework is a support for multiple instantiation [46], i.e. to allow to assign a type from each domain to an instance-level model element.

5.5 Multi-domain modeling in ViatraDSM: An example

The example of Sec. 5.3 consists of two domains, the Enterprise Security Policy (ESP) designer, and UML's Performance profile. In this case, the goal is to provide a way for a performance modeling expert to view the models created by the enterprise security experts directly from the native UML Performance view based upon the multi-domain modeling support of ViatraDSM.

Since the domain metamodels are structurally similar, a combination of the metamodel-level and transformation-based domain integration techniques will be used. One could also opt for a model-level integration approach, however, in this specific case primary system design is carried out in the ESP domain, and the UML Performance view is only used for the assignment of performance-specific model properties.

First, the two editor metamodels for both of the domains are constructed. Next, the mapping from the ESP domain to UML Performance is defined. The ESP domain incorporates both structural (Building-Network-Room-Device) and dynamic views (Processes). The following mapping is used:

- Installation → PerformanceContext
- Device → PResource
  - ApplicationCPU → PProcessingResource
  - DatabaseCPU → PProcessingResource
  - DiskWriter → PPassiveResource
  - DoorlockActuator → PPassiveResource
  - SecurityCardReader → PPassiveResource
  - VideoCamera → PPassiveResource
- Process → PScenario
- Activity → PStep

The metamodel-level mapping ensures that every model instance constructed in the ESP domain will be visible in the UML Performance domain as well (naturally, this holds for the mapped model elements). Furthermore, all constraints have to be fulfilled in both domains. In this case, additional constraints are required to express that the containment structure of the language metamodels is different.

In Fig. 5.7, the metamodel-level integration technique with transformation-based integration is combined. When the containment tree structure of the source domain cannot be mapped directly to the target domain by subclassing, additional model transformation steps may be required.

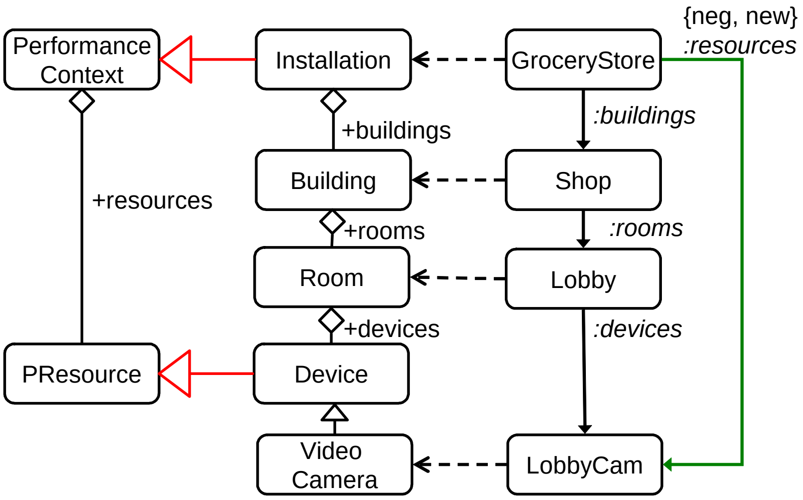


Figure 5.7: Multi-domain models

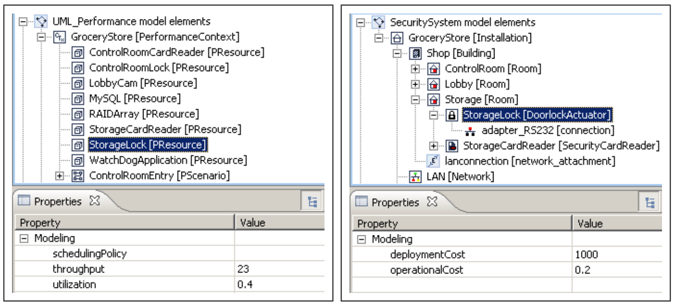
In this particular case, **Device** instances associated to a given **Installation** can be reached in three navigational steps in the ESP domain, however in the UML Performance Profile **PResources** are directly contained by **Performance Context** instances. This means that for already existing Device instances, such as LobbyCam being contained as a valid PResource by the GroceryStore performance context, a new relation of type **resources** must be added (marked by new keyword in Fig. 5.7). This simple ”correction” is implemented by the following VIATRA2-native model transformation program:

```
namespace DSM.machines.UML_Performace;

import DSM.metamodel.SecuritySystem.SecuritySystemEditor;
import DSM.metamodel.UML_Performance.UMLPerformanceEditor;

machine resourceCorrector
{
  gtrule deviceConnector(in I, in D) =
  {
    precondition pattern lhs(PC,D) =
    {
      // describe the model in the
      // SecuritySystem domain
      Installation(I)
      {
        Installation.Building(B)
        {
          Installation.Building.Room(R)
          {
            Installation.Device(D);
          }
        }
      }
      // describe the model in the
      // PerformanceContext domain
      PerformanceContext(PC)
      {
        PerformanceContext.PResource(PRes);
      }
      // check that the instances matched
      // are indeed the same
      check(I==PC)
      check(PRes==D)
      // check that the presources relation
      // has not been added previously
      neg pattern p(PC,D) =
      {
        PerformanceContext(PC);
        Installation.Device(D);
        PerformanceContext.presources(R,PC,D);
      }
    }
    postcondition pattern rhs(PC,D) =
    {
      PerformanceContext(PC);
      Installation.Device(D);
      // add the presources relation
      PerformanceContext.presources(R,PC,D);
    }
  }
  rule main() = seq
  {
    // perform the transformation for all models
    forall Installation, Device below ’DSM’.’model’
      with apply deviceConnector(Installation,Device) do skip;
  }
}
```

The end result can be seen on Fig. 5.8. The screenshot extracts show the same physical model instance from two different perspectives. As the user is editing the model in the ESP domain, the changes are instantly visible in the UML Performance domain. However, these are domain-specific views, thus in each editor only those details are visible which are relevant for the given visual language.



```
⊟ UML_Performance model elements
  ⊟ GroceryStore [PerformanceContext]
      ControlRoomCardReader [PResource]
      ControlRoomLock [PResource]
      LobbyCam [PResource]
      MySQL [PResource]
      RAIDArray [PResource]
      StorageCardReader [PResource]
      StorageLock [PResource]
      WatchDogApplication [PResource]
    ⊞ ControlRoomEntry [PScenario]
```

Properties

Property	Value
⊟ Modeling	
schedulingPolicy	
throughput	23
utilization	0.4

(a) UML Performance

```
⊟ SecuritySystem model elements
  ⊟ GroceryStore [Installation]
    ⊟ Shop [Building]
      ⊞ ControlRoom [Room]
      ⊞ Lobby [Room]
      ⊟ Storage [Room]
        ⊟ StorageLock [DoorlockActuator]
            adapter_RS232 [connection]
        ⊞ StorageCardReader [SecurityCardReader]
      lanconnection [network_attachment]
    LAN [Network]
```

Properties

Property	Value
⊟ Modeling	
deploymentCost	1000
operationalCost	0.2

(b) Enterprise Security Policy

Figure 5.8: Domain-specific editors: the same model element in two different domains

Chapter 6

Mapping between abstract and concrete syntax

6.1 Introduction

The importance of arbitrary abstract-to-concrete syntax mapping in visual languages is best supported by the simplicity argument: abstract syntax representation tends to be too complicated and difficult to look through for the human user. The reason for this phenomenon lies in the fact that most of the information in the abstract syntax representation is encoded in a structural fashion, while concrete syntax is usually more compact. For instance, a String property in a Class instance might be represented as a relation pointing from the Class entity to an instance of `datatypes.String` in abstract syntax, and a simple label on the graphical node representing the Class in concrete syntax (see Fig. 6.1).

As discussed in Chapter 2, traditional domain-specific modeling environments such as MetaEdit+ offer only a single modeling layer which means that the language engineer is limited to assigning a graphical representation to each metamodel element, based on their role (node or edge). In contrast, in recent initiatives, such as Eclipse's General Modeling Framework[15], the modeling and diagram layers are conceptually separated, meaning that diagrams are stored on a separate modeling layer which is mapped onto the logical model using generated Java code (based on a declarative mapping model). However, this conceptual separation is only partial, because GMF only allows for logical nodes being mapped to diagram nodes, and logical edges to diagram edges. Thus, the language engineer can use this technique to create diagrams which present a projection of the logical modelspace, however there is no support for more useful constructs such as aggre-

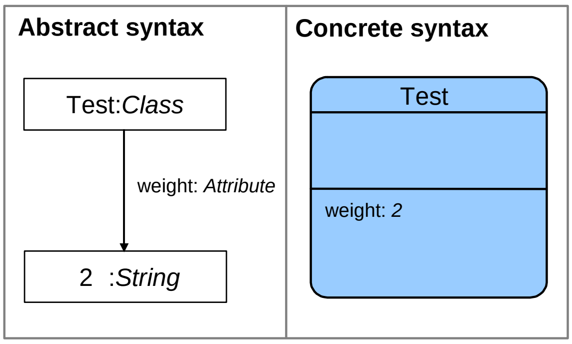


Figure 6.1: Abstract vs. concrete syntax representation

gations (information obtained from many model elements mapped to a single diagram element).

The problem of mapping between abstract and concrete syntaxes of a visual language is similar to the problem in the domain of textual languages (as discussed in [52, 22]). The important difference is that in textual languages a separate "visualisation" modeling layer between the abstract syntax representation (e.g. an Abstract Syntax Tree) and the visual source code representation is absent. In visual languages, diagrams display a "projection", or aspect of the logical modelspace; its equivalent in textual languages would be an aspect-oriented editor which would enable the developer to only see certain aspects of the source code.

The most important research result presented in this thesis is declarative support for the arbitrary mapping between abstract and concrete syntax (logical and diagram models) in the ViatraDSM framework. The most important features of the implementation are:

- The mapping technique allows for arbitrary bidirectional mapping between abstract and concrete syntax models (logical and diagram models), and provides declarative support based on a mapping metamodel and VIATRA2 transformations.
- Diagram and logical models share the same modelspace, making VIATRA2-native GTASM transformations between abstract and concrete syntax models possible.
- The mapping interfaces follow the plugin architecture of the ViatraDSM framework by allowing the plugin author to design and implement mapping in both directions in a flexible way (e.g. by using pure Java code, or mostly declarative techniques).
- The mapping metamodel allows for future automatic generation based on a concrete syntax metamodel, further minimizing the amount of manual coding required to build a domain-specific visual editor.

6.2 Architecture

ViatraDSM and GMF share the following modeling layer concepts (the terminology is taken from official GMF documentation):

1. **domain (language) metamodel**, which defines the logical concepts of the language (abstract syntax);
2. **diagram metamodel**, which defines a diagram metamodel, including the graphical appearance of model elements (concrete syntax);
3. **mapping definition**, to create a bridge between the domain metamodel and its visual representation;
4. **tooling definition**, which defines how the user can edit models.

In the case of ViatraDSM, a separate tooling definition is only in a planning stage, because currently the framework provides a creation tool for every diagram metaelement, and the selection tool for standard editing actions (e.g. deleting, moving, resizing, reparenting, retargeting edges). These possibilites can be extended by allowing the toolsmith to define custom tools which perform custom editing actions (a special tool for increasing a property's value, for example).

It is important to note the difference between the *model level* and the *metamodel level* separation of logical models and diagrams.

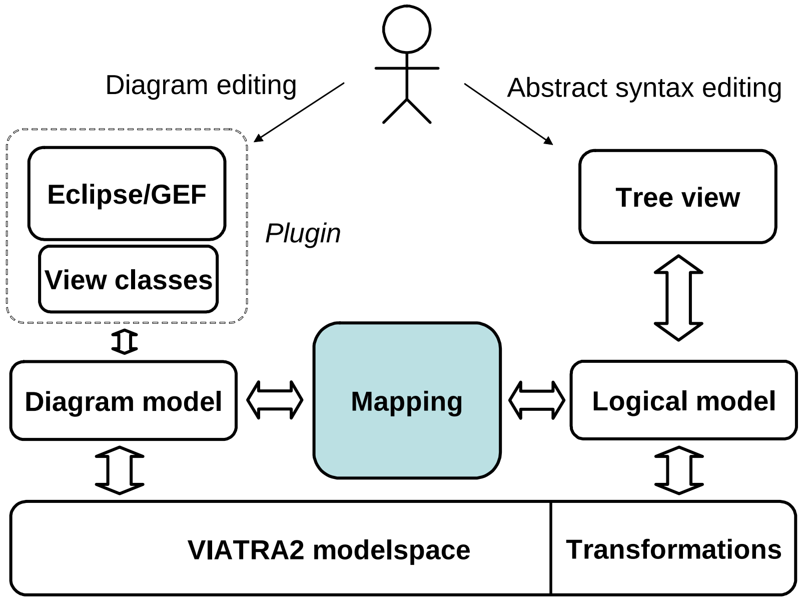


Figure 6.2: Mapping architecture

- On the **model level**, most tools separate the two layers by giving the user (almost) complete control over what parts of the logical model are visualized. This means, that the user either creates logical models by editing the model on a diagram, or visualizes existing elements by dragging them onto a diagram, for instance. However, diagrams strictly follow the structure of the logical model, i.e. a single logical entity is always represented by a single graphical entity. This makes the graphical editing easier to implement, because the graphical representation can be directly mapped onto the logical domain.

- The **metamodel level** separation, as implemented by both ViatraDSM and GMF, enables the language engineer to define visualization independently. Logical and diagram metamodels are constructed separately. This means that the GUI-driven editing only affects the diagram models directly, and the required changes to the logical model are generated by a mapping interface.

The mapping between the logical and diagram models can be approached in two ways: either by completely separating the two layers and implementing **bidirectional synchronization** between them. This is the broadest conceptualization of abstract syntax-concrete syntax separation, however, due to its generality, this approach can be difficult to implement for the toolsmith.

The other approach, **bidirectional mapping** makes use of the fact that the user can only apply a limited and well-defined set of modifications to the diagram through the graphical interface. Therefore, this approach maps the editing actions of the user to the logical model (see Fig. 6.3). As the user is editing the models visually, the user's editing actions are transformed into compund commands that affect the logical and diagram models.

This mapping interface is also responsible for reflecting the changes of the logical models in diagrams. Logical models can be altered independently of the diagram model in two main ways: (i) the user can edit the logical models using the tree view, (ii) and the transformation engine can execute simulation steps, or other transformation actions. The changes of the logical model are coded into notification objects which are distributed according to the event-listener design pattern. The mapper interface supplies a listener, which receives these notification objects, and maps them

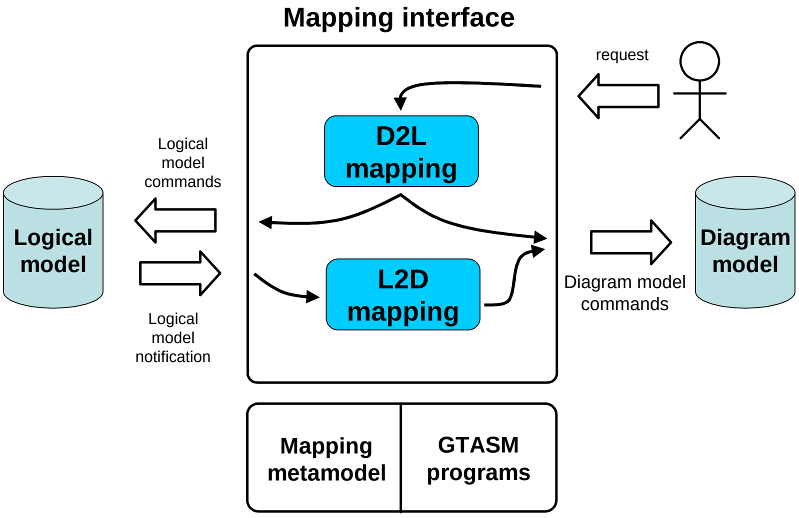


Figure 6.3: Mapping interface

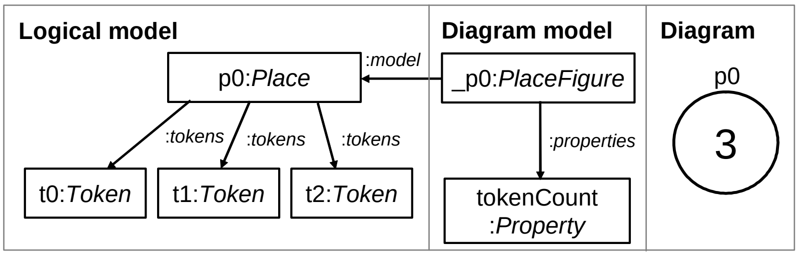


Figure 6.4: Presentation layers in the ViatraDSM framework

to valid diagram-specific commands which alter the diagram model and thus eventually change the appearance of diagrams.

The specification of this bidirectional mapping mechanism should ideally be fully declarative, in order to avoid the necessity of writing complicated and critically important code. GMF solves this problem by employing an EMF-based *mapping definition*, which plays a crucial role in the code generation process. This mapping definition, however, has inherent limitations. For example, GMF only allows the mapping of EClasses to nodes, and EReferences to edges. That is, it imposes a restriction on how diagram definitions can be assigned to logical models.

Although GMF allows for a partial representation of the logical model (meaning that not every logical feature has a graphical counterpart), it is rather limited in providing support for more advanced mappings. For instance, *attributes* or *properties* are frequently used in diagrams for the expression of the amount of model subelements assigned to a container. In a Petri net editor, for example, *place* nodes can contain *token* nodes. It is common to visualize the amount of tokens assigned to a place by displaying a single integer attribute as a decorator of the place's graphical representation (see Fig. 6.4). With the GMF approach, it's not possible to establish such a logical link between the amount of tokens assigned to the place, and its attribute.

Thus, with ViatraDSM mapping facilities, my most important goal was to overcome this restriction, first by designing an effective mapping programming interface, and by providing a declarative "bridge" to this interface using the mapping metamodel and VIATRA2 transformations.

In the next section, the three approaches to mapping implementation are discussed in detail.

6.3 Techniques

The implementation behind the mapping interface on Fig. 6.2 can be constructed in three (plus one) ways:

- by using **manually written code** (utilizing a well-defined application programming interface);
- by constructing a mapping model based on the mapping metamodel, which is interpreted by the ViatraDSM framework, and controls the translation of editing requests to commands and model notification objects to diagram commands;
- as a special command class, GTASM transformations can be invoked in both directions; the changes made by these transformations are automatically mapped into the DSM modelspace by the framework's notification layer.
- Due to the flexible API design, all of these approaches can be intermixed according to the needs of the plugin engineer.

6.3.1 The presentation layer of ViatraDSM

Before discussing the implementation details of the mapping mechanism, I give a short insight into the internals of GEF and the presentation classes of the ViatraDSM framework.

The ViatraDSM framework's presentation layer is based on the Graphical Editing Framework's (see 2.5.3) Model-View-Controller scheme. In GEF, the model may any kind of Java class, whereas the view and controller classes must implement the `IFigure` and `EditPart` interfaces. The GEF controller, called **EditPart**, is a *heavy* controller, meaning that almost all of the rendering and editing functionality is delegated to the controller classes.

Generally, an EditPart takes care of the following tasks:

- maintaining references to the model and view objects;
- instantiating the view class;
- listening to the model's changes and refreshing the view;
- handling user interaction.

GEF's default model-to-diagram mapping

Upon the initialization of the graphical editor, GEF creates an EditPart tree, based on the model, which corresponds to the containment hierarchy of figures on the diagram. More precisely, the model is recursively queried through the `getModelChildren()` method of the EditParts, which must return a list of the EditPart's model elements' children. Although it is possible to customize what parts of the model are displayed on the diagram by applying various tricks to the `getModelChildren()` implementation, this is not very practical, because the model is still traversed in a top-down depth-first search, making an arbitrary model-to-diagram mapping impossible.

This means the GEF was designed to only display diagrams which strongly correspond to the model's structure. Therefore, a separate modeling layer for diagrams has to be introduced in order to allow for an arbitrary mapping between logical models and diagrams. However, this requires a mapping mechanism which makes appropriate alterations to the logical model as the user is editing the diagrams, and tracks the links between the logical model and its appearance on the diagram.

The EditParts are included in the ViatraDSM framework, and all plugin-dependent functionality (view instantiation, view refresh) are handled through the various interfaces described in Section 4.2.3 (IDSMGraphicalEditingContributor, IViewFactory).

User interaction

In GEF, the handling of user events (*Requests*) is implemented in the following way:

1. First, a Request is issued by GEF, or by a custom Tool (which may be selected from the Palette).
2. Next, the target EditPart's `understandsRequest()` method is called, which queries the EditPart whether is supports the given Request or not.
3. If the answer was positive, the Request is transformed into a *Command* instance through the EditParts `performRequest()` method; the Command contains code which can modify the model using information obtained from the Request object.
4. Finally, the Command is placed onto the *CommandStack*, which supports undo-redo operations, and is executed by the framework.

This pattern has a major drawback: implementing all the cases for various requests would clutter the source code of EditParts too much. Moreover, typically the same code is shared among various types of EditParts, however introducing custom superclasses for this purpose is not always possible. Therefore, GEF introduced the notion of *EditPolicies*, which are delegate classes to handle the Request - Command mapping. EditPolicies can be attached to EditParts by overriding the `createEditPolicies()` method.

EditPolicies are associated to *Roles*, which define the context of the operation. The most important pre-defined roles are the following (denoted with their internal string IDs):

- Component role. Component editpolicies handle the deletion of model elements.
- Connection bendpoints role. This role handles the creation and deletion of connection bendpoints.
- Connection endpoints role. If the endpoints of a connection (EdgeFigure) are movable (on the target NodeFigure), this editpolicy creates the appropriate commands.
- Connection role. This role is responsible for the deletion of connections.
- Container role. A container editpolicy handles the creation and moving of nodes.
- Graphical node role. This role handles the creation and retargeting/resourcing of connections between source and target nodes.
- Layout role. The layout role is responsible for the setting of layout constraints based on information obtained from a Constraint object, provided by GEF (and the `LayoutManager` class assotiated to the Diagram).

- Direct edit role. Direct editing requests, where a user edits some attributes directly on the diagram, using a *cell editor* (a small editor box, e.g. a textbox that pops up for editing the name of a diagram element if the user performs a double click), are handled though the direct edit editpolicies.

Implementation details

In ViatraDSM, most of the editpolicies described above are provided by the framework. They are the entry point of the mapping mechanism, because the general structure of an editpolicy implementation looks like this:

```
public Command createCommand(Request r)
{
        // create a compound command, which will
        // encompass both diagram-specific and
        // logical model-specific commands

        CompoundCommand compound = new CompoundCommand();

        << create diagram model-specific commands,
              add them to the compound >>
        << invoke the mapper, and query for a
            logical model-specific command >>
        << if it is valid, add it to the compound >>

        return compound;
}
```

Some editpolicies cannot be provided by the framework, because they depend on how certain diagram-specific properties are handled by the view classes. These include:

- LayoutEditPolicy: as Diagram properties are string key-value pairs, it is best to leave how constraints are encoded into property values in the hands of the plugin author.
- DirectEditPolicy: as direct editing can affect various parts of figures, even those which have no diagram model element assigned, this editpolicy has to be implemented by the plugin author as well.
- BendpointEditPolicy: as bendpoints can be stored in strings in a multitude of ways, this editpolicy is also delegated to the plugin.

However, default implementations of these classes are provided for the *Basic* family of view components, which is part of the ViatraDSM framework.

6.3.2 Java interfaces

In this section, the Java interfaces of the mapping mechanism are discussed in detail.

The core interface is called `IDSMMapper`, which is referenced by the `IDSMGraphicalEditingContributor` interface. That is implemented by all editor plugins, thus an IDSMMapper instance must be provided by all domain specific editors. The IDSMMapper interface provides access to two companion interfaces: `IDSMCommandMapper` and `IDSMNotificationMapper`. More precisely, the instances of these two can be queried; they correspond to the two mapping components (directions) on Fig. 6.3.

Diagram-to-model mapping

Diagram-to-model mapping can be implemented using the `IDSMCommandMapper` interface. The methods define a *stateless* request-command mapper, meaning that the mapper has no access to previously issued requests or executed commands. This may seem like a major restriction, however in practice special editing action sequences are rarely used in graphical editors (e.g. requiring the user to perform certain actions in a given order to achieve a single modification of the model). However, previously executed commands *can* be accessed if the plugin author provides a custom implementation of this interface, because the Command Stack of the editor may be visible to the Mapper if the author enables that.

The interface contains methods for all the different scenarios that may involve the modification of the logical model, i.e. deletion and creation of diagram elements, resourcing and retargeting of edges, moving, layouting of nodes, and direct edit events as well. These methods receive references to the request object, optional information placeholders (e.g. constraints), and the thin wrapper adaptors of diagram (meta)elements. It is important to note that the commands provided by the IDSMCommandMapper instance are executed **after** the modifications have been committed to the diagram model. This is a design decision: this approach makes it easier to implement the mapping with model transformations because the modified diagram model is already in the model space when the transformation executes.

Model-to-diagram mapping

Model-to-diagram mapping can be implemented using the `IDSMNotificationMapper` interface. An IDSMNotificationMapper instance can be constructed in two ways:

- *Stateful* implementation. By creating a DSM logical model notification listener which receives all notification levels from the ViatraDSM framework's notification dispatcher object and translates them into Commands valid on the diagram model;

- *Stateless* implementation. In this case, the framework calls the `getDiagramModification()` method of the IDSMNotificationMapper instance after each emitted dsm logical model notification event, and executes the returned command on the diagram model.

It should be noted that the stateful approach is considered to be experimental, and there exists no declarative support for it (yet). The reason for this is because the DSM notification layer currently does not support transactions, and thus at the present stage transformations cannot issue `transaction begin` and `transaction end` messages which would enable the stateful mapper to generate commands based on a pre-defined pattern of modification sequences.

More generally, the main challenge of implementing the logical model → diagram mapping is the fact that while we can take advantage of the fact that the diagram model may only be modified using a well-determined set of user actions (requests) in the case of diagram-model mapping, in the reverse direction this is not true. In other words, the logical model may be modified by transformations (simulation) and abstract syntax editing by the user as well, and in order to achieve *truly* arbitrary model-to-diagram mapping, any sequence of notification objects may be mapped to a given command (sequence) which modifies the diagram.

Trigger-based stateless approach It is possible to implement arbitrary logical model-to-diagram mapping within the constraints of the stateless implementation, although with a performance penalty. The *trigger-based* approach uses certain types of notification events as triggers which scan the logical modelspace, and look for a given pattern around the notification source. If that has been found, they emit a commandsequence for the model (this can be generalised to the whole modelspace, although the principle of locality should be employed to the maximum to ensure acceptable performace).

6.3.3 The Mapping metamodel

The mapping metamodel forms the basis of the second technique mentioned in Sec. 6.3. As previously established, the *stateless* mapping is basically the generation of a command (sequence) based on a request, the target model elements, and some additional contextual information (or, in the case of model-to-diagram mapping, commands are generated based on information contained in notification objects).

In designing metamodel-based declarative support for mapping constructs, the most important question to answer is the level of abstraction. In this case, answering this question was simple, because the context was already given: the metamodels must support the mapping of Requests / Notification objects to Commands. Thus, I designed a Mapping metamodel which consists of multiple submodels. In the following section, these will be discussed in detail.

The Command metamodel

The Command metamodel (Fig. 6.5) describes an ordered sequence of Command objects, the CommandSequence. Commands are associated to a CommandType classifier, which is actually an extensible enumeration (Create, Delete, Modify). Commands may have an arbitrary number of Parameters, which can be bound to ModelElements.

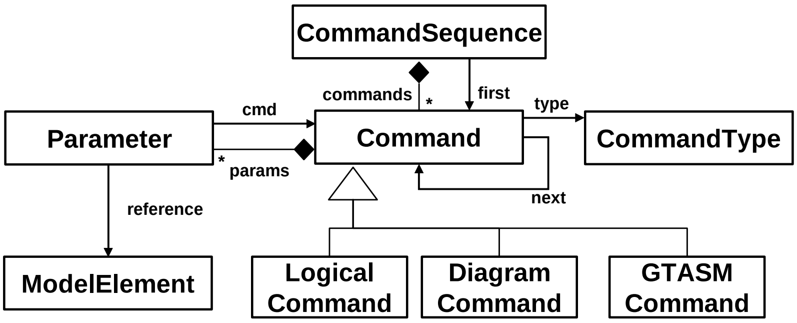


Figure 6.5: Command metamodel

There are three types of commands: Logical Commands, Diagram Commands, and a special class called GTASM Commands. This distinction is required because the ViatraDSM framework implementation uses different base classes for each of these. The GTASM command is a special command class which can pass parameters to a GTASM transformation program and invoke it.

Request and Notification metamodels

The Request and Notification metamodels (Fig. 6.6) describe the contextual information which is used in the generation of commands. Both Request and Notification are typed containers for Context Elements / Notification Elements, which can be associated to Model Elements.

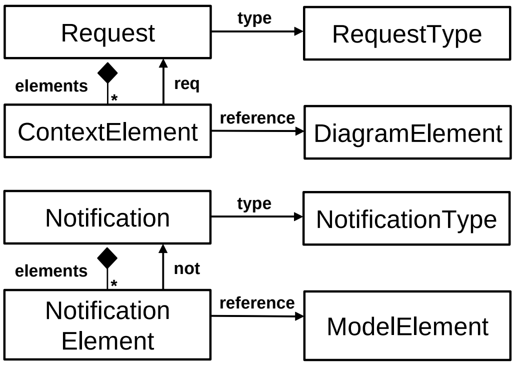


Figure 6.6: Request and Notification metamodels

Both the Request and Notification classes can be easily generated from actual Java objects.

Binding and Trigger metamodels

The Tigger and Binding metamodels (Fig. 6.7) define the "glue" between the commands' Parameters and request/notification Context Elements / Notification Elements. This way, a logical relationship can be established between the information present in a Request / Notification object and a sequence of Commands. The BindingType / TriggerType attribute is a reference to a built-in enumeration of the ViatraDSM framework, which describes how the objects should be mapped on the Java level.

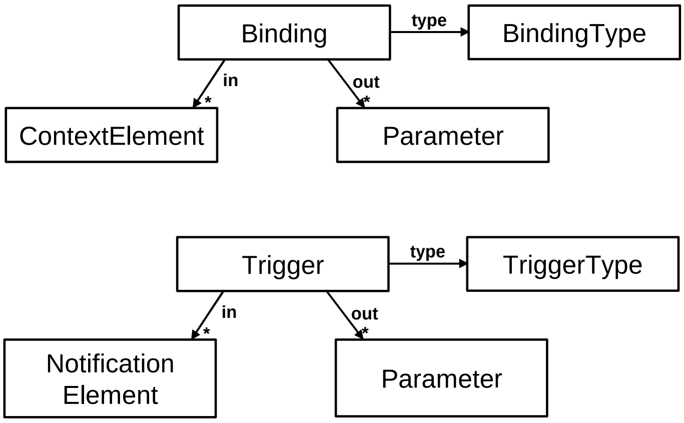


Figure 6.7: Binding/Trigger metamodel

Mapping rules metamodel

The main part of the Mapping metamodel is shown on Fig. 6.8. A concrete Mapping can either be a logical-to-diagram mapping (L2DMapping) or a diagram-to-logical mapping (D2LMapping) construct. Both consist of rules (L2DMappingRule and D2LMappingRule), which map a Request or Notification object to a Command sequence, using as many Bindings or Triggers as necessary.

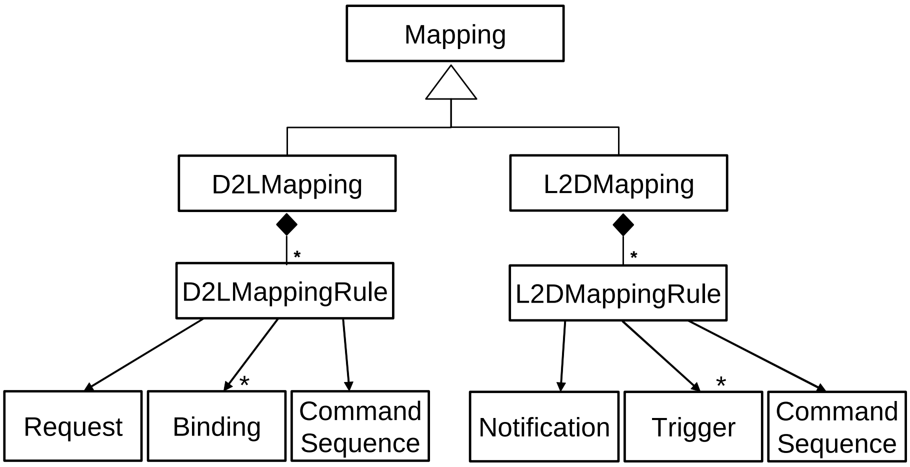


Figure 6.8: Mapping metamodel

6.3.4 Using the mapping metamodel

Some knowledge of the mapping mechanism is required to use the mapping metamodel efficiently. The basic rules are:

1. A mapping is identified by its Request attachment. In other words, only one mapping rule should be defined for a given RequestType - ContextElement(s) pattern. Naturally, this applies to Notification mappings as well.

2. RequestTypes, NotificationTypes, BindingTypes and TriggerTypes are identified by the local name of their representing model element. Some types are provided by the ViatraDSM framework (see sec. 6.3.5), the plugin author can provide custom BindingType and TriggerType implementations (the set of RequestTypes and NotificationTypes cannot be extended, yet).

3. The ModelElements which can be referenced by notification objects and command parameters should be valid metamodel elements in the Diagram and Editor (logical model) domains.

On Fig. 6.9, a simple example for a mapping model is shown. In this case, the creation of an InArcFigure between a TransitionFigure and a PlaceFigure is mapped onto the creation of an InArc between the corresponding Transition and Place instances (note that for simplicity, the appropriate metamodel references from the Target and Source context elements and parameters have been omitted).

Creating a mapping model such as the one shown on Fig. 6.9 requires knowledge of the various requests, commands, and bindings of the ViatraDSM framework. For instance, the designer has to be aware that a *CreateConnection* request has three context elements, a source node figure, a target node figure, and a connection figure type; similarly, a *CreateEdge* command has a source node, a target node, and a connection type; a *ModelBinding* ensures that the Diagram Element referenced by the contextelement will be queried for its **model** association and the resulting logical model element will be assigned as the out parameter.

It would have been possible to include all the various Bindings, Requests, Notifications, Commands in the metamodels. However, the metamodel-based mapping approach was *not* primarily

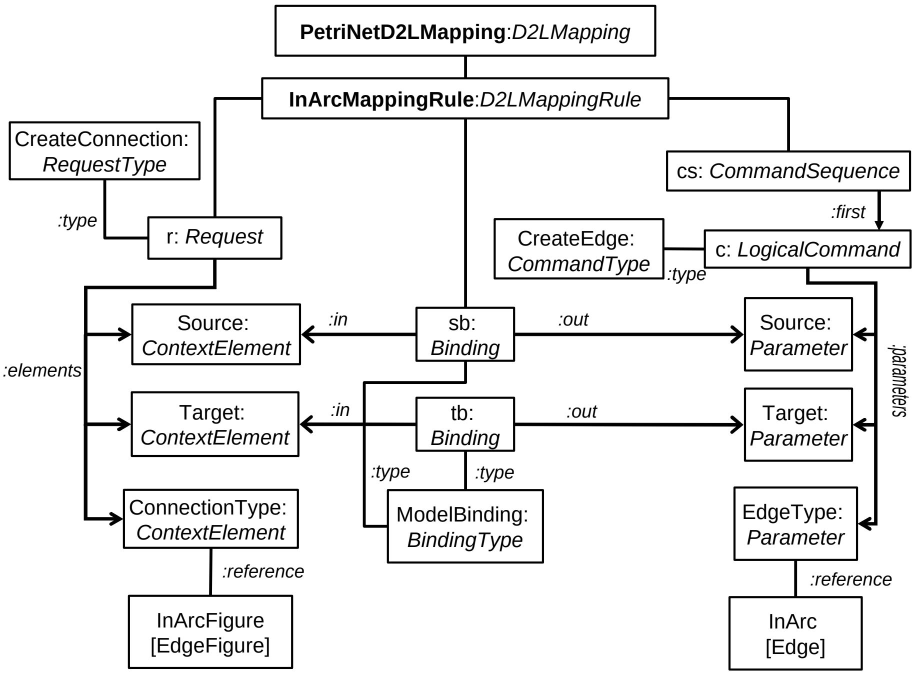


Figure 6.9: Mapping model example: Petri net edgefigures and edges

meant to be designed by the tool author directly, but rather generated by a (future) automated mapping generator. Therefore, extending the mapping metamodel to include all possible combinations is unnecessary; tool authors should rather use the model transformation-based approach.

6.3.5 Interpreting mapping models

The mapping models are read by an interpreter, which implements the `IDSMMapper` interface. The interpreter creates an internal hashmap-based representation (the request mapper and notification mapper registries), which is used at run-time whenever the mapping interface is queried by the EditPolicies.

The generation of a command sequence for a given request is based on the following process:

1. A key is generated for the request. The key is based on the request type, the names of the context elements, and the names of any possible referenced model elements (more precisely, the typenames are used, as metamodel elements are referenced).

2. The request mapper registry is queried for an entry using the generated key. If no result is returned, the generation process ends.

3. The result of the query is a binding and commandsequence descriptor, which can be used to instantiate command and binding classes and call their appropriate `performBinding()` and `set()` methods.

4. The binding classes are instantiated and used with the request's context elements to obtain command parameters.

5. The command classes are instantiated, their parameters are set and the result is returned.

The generation of a command sequence based on a notification is identical to the process described above, with a slight difference: in that case, the notification mapper registry and triggers are used.

In the following sections, currently supported request, notification, binding, and trigger types are briefly described.

Request types

- CreateNewNodeFigure. The user created a node figure, by using a creation tool (i.e. the node figure has no pre-existing counterpart in the logical model).
- CreateRepresentingNodeFigure. The user dragged a node from the logical model (tree) onto the diagram.
- CreateNewConnection. The user created a new edge figure by using a creation tool.
- CreateRepresentingConnection. The user dragged an edge from the logical model onto the diagram.
- RetargetConnection. The user retargeted a connection by moving its target endpoint onto a different nodefigure.
- ResourceConnection. The user resourced a connection by moving its source endpoint onto a different nodefigure.
- DeleteNodeFigure. The user deleted a nodefigure.
- DeleteConnection. The user deleted a connection figure.
- MoveNodeFigure. The user moved a node figure into a different container.
- SetConstraint. The user applied a new graphical constraint on the diagram element (e.g. resize, move around, set radius, etc).
- AddBendpoint. The user added a new bendpoint to an edgefigure.
- DeleteBendpoint. The user deleted a bendpoint from an edgefigure.

Notification types

- NewNode. A new child node has been created.
- NewEdge. A new source/target edge has been created.
- NewProperty. A new property has been created.
- DeleteNode. An existing child node has been deleted.
- DeleteEdge. An existing source/target edge has been deleted.
- DeleteProperty. An existing property has been deleted.
- ChangeName. The name of the model element has changed.
- ChangePropertyValue. The value of the property has changed.

Predefined Commands

- Logical Commands
  - ChangeEdgeSource. Sets the source of an edge.
  - ChangeEdgeTarget. Sets the target of an edge.
  - ChangeProperty. Sets a new property value.
  - CreateEdge. Creates an edge (precondition: source and target must exist).
  - CreateNode. Creates a node.
  - CreateProperty. Creates an optional property.
  - DeleteEdge. Deletes an edge.
  - DeleteNode. Deletes a node (precondition: all properties and source/target edges must be deleted).
  - DeleteProperty. Deletes a property.
  - MoveNode. Moves the node to a different container.
  - RenameElement. Renames a model element (edge, node).
- Diagram Commands
  - ChangeEdgeFigureSource. Sets the source of an edgefigure.
  - ChangeEdgeFigureTarget. Sets the target of an edgefigure.
  - ChangeDiagramProperty. Sets a new diagram property value.
  - CreateEdgeFigure. Creates an edge figure.
  - CreateNodeFigure. Creates a node figure.
  - DeleteEdgeFigure. Deletes an edge figure.
  - DeleteNodeFigure. Deletes a node figure (precondition: all source/target edgefigures must be deleted).
  - MoveNodeFigure. Moves a node figure to a different container.
  - RenameDiagramElement. Renames a diagram element.

Predefined bindings

This can be extended by plugins by implementing the `IDSMBinding` interface. The collection of binding implementations can be queried on the `IDSMCommandMapper` interface.

The following bindings are provided by the framework:

- ModelBinding. This binding is used for finding the appropriate model elements in the logical model for their representating elements in the diagram model, using the **model** relation that is pointing from diagram elements to logical model elements (see Fig. 4.5). The binding simply uses the thin wrapper adaptor layer provided by the ViatraDSM framework to navigate through the model relation and retrieves the corresponding logical model element.
- PropertyValueBinding. This binding can be used to extract a String object from a property (logical or diagram both work).

Predefined triggers

This can be extended by plugins by implementing the `IDSMTrigger` interface. They can be queried using the `IDSMNotificationMapper` instance.

- ModelTrigger. Analogously to the ModelBinding, this trigger is used for finding a possible diagram representation for a given logical model element. Currently, the implementation is slow because the thin wrapper layer does not support "inverse" queries along the **model** relation.
- PropertyTrigger. This is essentially the same as PropertyValueBinding.

6.3.6 GTASM transformations

The third technique introduced in Sec. 6.3 is based on the capabilities of the VIATRA2 model transformation engine. The basic idea is the following: instead of constructing commands that alter the modelspace directly, the interpreter invokes a native GTASM transformation (with optionally passing arguments obtained through the bindings/triggers), which scans the diagram models (or the logical model, in the case of L2D mapping), and applies graph transformation rules (or pure ASM rules, depending on the author's preference).

This approach can be more intuitive for the toolsmith than using the mapping metamodel, because graph patterns are better comprehensible for humans. However, this is a more general approach and thus somewhat slower to interpret than the mapping models.

For example, recall the mapping introduced on Fig. 6.9, in which the creation of an InArcFigure between a TransitionFigure and a PlaceFigure was mapped to the creation of an Edge between the Transition and Place instances corresponding to the TransitionFigure and PlaceFigure.

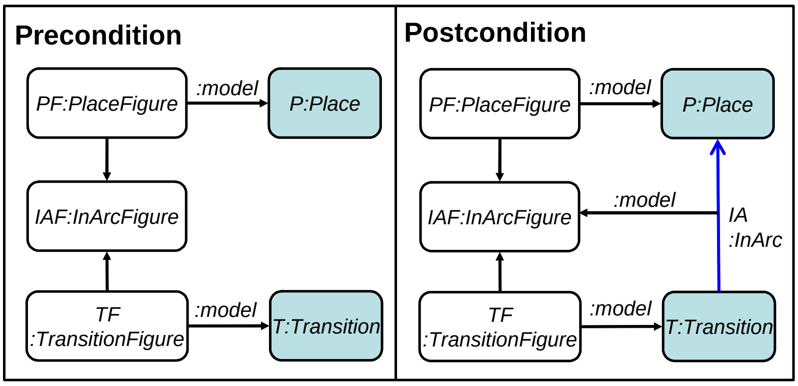


Figure 6.10: Mapping transformation example: Petri net edgefigures and edges

That scenario can be visualised using a graph pattern as shown on Fig. 6.10. Elements highlighted in blue are part of the diagram model, while elements in white are logical model elements. It is important to note that the transformation approach takes advantage of the fact that the transformation program is executed *after* the diagram model has been modified, thus the precondition pattern can be successfully matched.

This simple graph transformation rule, and an ASM machine capable of invoking it using arguments provided by the framework, can be programmed in VTCL, the native transformation specification language of VIATRA2, in the following way:

```
namespace DSM.machines.PetriNet;

import DSM.metamodel.PetriNet.PetriNetDiagram.PetriNetFigure;
import DSM.metamodel.PetriNet.PetriNetEditor.PetriNet;

machine inArcMapper
{
  gtrule inArcMapping(in PF, in TF, out InArc) =
  {
        precondition pattern diagram(PF,TF,InArcF,P,T) =
        {
          // first, the diagram model is described
          PlaceFigure(PF);
          TransitionFigure(TF);
          InArcFigure(InArcF);
          PlaceFigure.pnd_inarcs(PFin,PF,InArcF);
          TransitionFigure.pnd_t_inarcs(TFin,TF,InArcF);
          // second, the logical model is described
          Place(P);
          Transition(T);
          // third, the association between them is established
          // by domain-specific subclasses of the model relation
          PlaceFigure.pnd_place_model(PFm,PF,P);
          TransitionFigure.pnd_transition_model(TFm,TF,T);
        }
        postcondition pattern editor(P,T,InArcF,InArc) =
        {
          // the InArcFigure needs to be declared because
          // the pnd_inarc_model relation will use it as target
          InArcFigure(InArcF);
          // the postcondition describes how
          // the logical model will look like
          Place(P);
          Transition(T);
          Transition.InArc(InArc,T,P);
          // and how the newly created InArc will be connected
          // to the logical model
          InArcFigure.pnd_inarc_model(PFInArcM,InArcF,InArc);
        }
  }
  rule main(in PlaceFFQN, in TransitionFFQN) = seq
  {
        let PlaceF = ref(PlaceFFQN) in
          let TransitionF = ref(TransitionFFQN) in
          // the input parameters are the fully qualified names
          // of the PlaceFigure and TransitionFigure instances
          // the ref() rule binds them to model elements in the modelspace
          seq
          {
            choose X with apply inArcMapping(PlaceF,TransitionF,InArc) do skip;
            // in this simple example, the machine only applies the rule
          }
  }
}
```

The GTASMCommand class This transformation program is invoked using the special GTASMCommand class. That command class is instantiated in a special way by the framework: using a reference to the GTASM machine, the command class can support an arbitrary number of parameters; however the bindings / triggers are still determined by an appropriate mapping model.

The transformation-based mapping approach is especially useful in logical model to diagram model mappings. As already discussed in 6.3.2, truly arbitrary mapping would require a stateful notification listener, and transaction support, however even that would not be entirely sufficient. If the user is editing the logical model using the tree-based abstract syntax editor, the transactional nature of complex editing action sequences cannot be guaranteed.

For instance, the language engineer wishes to map a Source Node-Edge-Association Node-Edge-Target Node pattern to a Source NodeFigure-EdgeFigure-Target NodeFigure triplet. The pattern in the logical model can be constructed in a number of ways, and thus a stateful notification follower implementation would require all of these ways to be specified so that the creation of the pattern can be detected.

However, if the user decides to edit a different portion of the modelspace, and return to the incomplete pattern later, the notification tracker would *not* be able to detect the construction of the complex pattern. Therefore, in this case, a GTASM transformation, which is invoked on the creation of all the elements in the pattern, is much simpler, and in fact the only solution. Although it is resource consuming to execute a transformation program each time the user performs an action on the GUI, the user is typically much slower than todays computers so this is not much of a problem anymore.

6.4 Summary and future improvements

In this chapter, the capabilities of the ViatraDSM framework regarding the mapping between abstract and concrete syntaxes of visual languages were discussed. The importance of the conceptual and metamodel-level separation of abstract and concrete syntax representation is supported the following arguments:

- the conceptual (model-level) separation of diagrams and models enables the *user* to select what to visualize and what to hide;
- the metamodel-level separation enables the *tool designer* to tailor the graphical appearance of the language precisely to the needs of the application domain while the abstract syntax representation can still be optimized for code generation or other types of model transformation.

The ViatraDSM framework achieves these goals by providing an array of imperative (Java code) and declarative (models, transformations) approaches which can be used to implement mapping in both directions (logical-diagram, diagram-logical).

The **diagram-to-logical model** mapping is based on a stateless translator which receives *Requests* from the user interface and provides *Commands* which modify the logical model. As the user can only perform a limited set of possible modification actions (requests), this approach is sufficient for a practically arbitrary mapping. However, the language engineer can opt for a transformation-based approach, where instead of generating commands directly, a model transformation is applied.

The **logical model-to-diagram** mapping can be implemented in two ways: either by a stateless command generating notification tracker, or using a trigger-oriented model transformation approach. That way, the logical modelspace is scanned every time a (pre-defined) modification occurs and a model transformation program is run which constructs the appropriate structures in the diagram model.

The main components of the implementation are: (i) Java interfaces and framework classes which facilitate the administration of mapping rules provided by domain plugins; (ii) the mapping metamodel, which is a complex set of metamodels designed to enable the language engineer to define mapping using purely declarative techniques; (iii) an interpreter and implementation classes which can read mapping rules from the model space and execute them by invoking the VIATRA2 transformation engine where appropriate.

The current implementation could be improved and extended in the following ways:

- The CommandMapper currently does not have access to the previous state of the diagram model, however in some scenarios that would be useful. This can be overcome currently by creating a Java implementation which accesses the Command Stack.
- As mentioned earlier, a stateful implementation of logical-to-diagram mapping would require *transaction support*. However, that would only be useful for automated logical model modifications (simulation).
- An improvement which will be soon implemented is extendable RequestTypes. That is required for custom tools, which can be defined and implemented by the language engineer to perform complex operations at a single mouse click, for instance (e.g. increate the number of tokens in a place).
- Another improvement could be allowing reverse queries along the *model* relation and its subtypes. The current implementation of the thin wrapper interfaces does not allow this, however it would speed up notification mapping by a factor of two at least.

Chapter 7

Case study: Petri net editor

In the previous chapters, theoretical foundations, design considerations, and implementation details were discussed. While plenty of simple examples were provided to ease understanding, I felt necessary to include a chapter which gives a complete overview of the process of constructing a domain-specific modeling language using the ViatraDSM framework. The example domain will be, as introduced in Sec. 2.8, the domain of *Petri nets*.

Step 1: Initial planning

The first step is to get a few Petri net experts together to work out the basic domain structure and domain rules. During the design of a domain specific editor, it is always important to have an expert of the given domain working on the project. His knowledge and expertise is a key factor in successfully adopting the DSM method within a company. Most probably the editor is created to increase the productivity of your developers. Therefore the usability and correctness of a domain specific editor and code generator is an essential point. If the editor is difficult to use, its model structure is incoherent or the code generated contains faults, the productivity would rather drop than increase. Only with a deep understanding of the domain can we assure that our editor gets a positive approval. Therefore an expert with a good understanding and deep knowledge of the domain is indispensable.

Step 2: Planning features

The ViatraDSM framework offers a wide range of possibilities from a simple editor using the built-in implementation for model representation and diagrams up to fully customizable editors with custom model representation, custom graphical elements and so on.

For instance, if the editor is to be used only for modeling and graphical display (e.g. only for documentation and not for code generation), the default implementation classes are sufficient. However if we need to access models outside the VIATRA2 framework and our graphical representation requires 3D rendering, we need to provide our own implementation of the domain metamodel and diagram interfaces.

Two model transformation features are implemented in this example: the simulation of Petri nets, and the generation of PNML[36] code. Since transformations are only available for VIATRA2-based models, the default VIATRA2-based implementation for model representation will be used.

The editor is required to look familiar for users who have experience with other Petri net tools, so the tool will employ custom graphical elements (using a circle for Petri net places instead of built-in rectangular boxes); the underlying graphical technology will be the default presentation layer of the ViatraDSM framework.

The mapping betweeen concrete and abstract syntax will be the most simplistic: places will correspond to node-like place figures, transitions to transition nodes, with inarc and outarcs shown as straight connections (bendpoints will be possible, too).

Step 3: Creating the domain metamodel

The first step in the implementation is the construction of the domain metamodel, that is, the structure and rules of the domain specific language.

This task should be carried out by the domain experts, since they fully understand the structure of the given domain. When designing the domain metamodel, the only thing the user needs to take care of is to use only the concepts defined by the core domain metamodel (Fig. 4.3). The Petri net domain metamodel is presented on Figure 7.1.

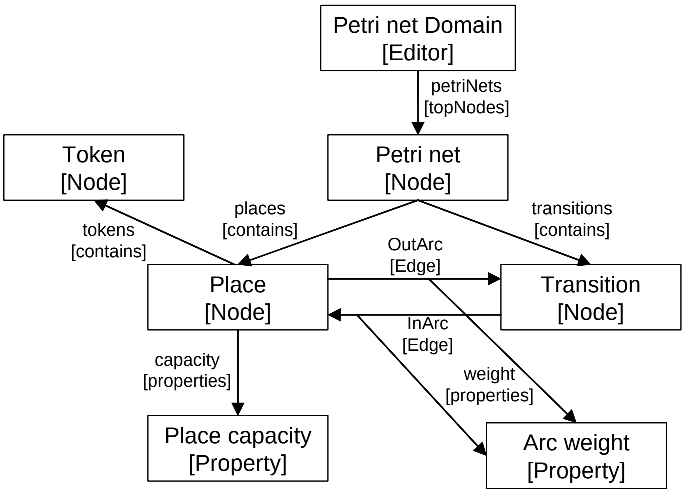


Figure 7.1: Step 3: Petri net domain metamodel

The domain expert has multiple ways of inputting this metamodel into the VIATRA2 model space. First, he may design it using the VIATRA2-specific graphical editor. If the expert is more familiar with textual languages, he may create a VTML file which contains the metamodel. A sample VTML code excerpt is provided below:

```
entity(petriNet)
{
    entity(place);
    supertypeOf(place,DSM.coremetamodel.Editor.Node);
    entity(transition);
    supertypeOf(transition,DSM.coremetamodel.Editor.Node);
    entity(token);
    supertypeOf(token,DSM.coremetamodel.Editor.Node);
    relation(inArc,transition,place);
    supertypeOf(inArc,DSM.coremetamodel.Editor.Node.Edge);
    relation(outArc,place,transition);
    supertypeOf(outArc,DSM.coremetamodel.Editor.Node.Edge);
    relation(tokens,place,token);
    relation(places,petriNet,place);
    relation(transitions,petriNet,transition);
    supertypeOf(tokens,DSM.coremetamodel.Editor.Node.contains);
    supertypeOf(places,DSM.coremetamodel.Editor.Node.contains);
    supertypeOf(transitions,DSM.coremetamodel.Editor.Node.contains);
    // define the properties
    entity(placeCapacity);
    relation(capacity, place, capacity);
    supertypeOf(capacity,DSM.coremetamodel.Editor.Node.properties);
    entity(arcWeight);
    relation(weight, inArc, arcWeight);
    relation(weight, outArc, arcWeight);
    supertypeOf(inArc.weight,DSM.coremetamodel.Editor.Node.Edge.properties);
    supertypeOf(outArc.weight,DSM.coremetamodel.Editor.Node.Edge.properties);
}
```

Finally, if a metamodel of the required domain exists already in some other modeling tool, he can use the various importers supplied with the VIATRA2 framework. In this latter case, after the import, the expert must ”tag” the elements of his metamodel using our core metamodel concepts. That means, for every element of his metamodel, he must tell whether that should be a node, an edge or a property, assigning an appropriate supertype from the core metamodel.

Step 4: Designing diagrams

Now that the structure of the domain is defined, the tree-structured editor provided in the Outline view of Eclipse is ready to use. For diagrams, some other metamodels have to be designed.

Each domain has exactly one domain metamodel (designed in step 3), but it may have numerous diagram metamodels. Each diagram metamodel describes one kind of diagram, which may be constructed using the element of the given domain. For Petri nets only one diagram will be used, which may contain places, transitions, inarcs and outarcs.

To design the Petri net diagram, the language engineer needs to use concepts included in the core diagram metamodel (Fig. 4.5).

Figures 7.2 and 7.3 show the Petri net diagram metamodel. By comparing it to figure 7.1 one may notice the similarities. In this case, the diagram metamodel has a very similar structure to the domain metamodel itself, simply because of the fact that on diagrams we would like to see the same hierarchy than in the model.

Step 5: Designing the mapping between diagrams and models

In this step, the mapping between concrete syntax (diagram models) and abstract syntax (editor or logical models) is designed and implemented. In the case of the Petri net editor, this mapping will be very simple: Places and Transitions will be displayed as Node figures, and inarcs and outarcs will be rendered as EdgeFigures. Basically, this means a one-to-one correspondence, however *Tokens* are not represented on the diagram directly. The number of tokens associated to a place

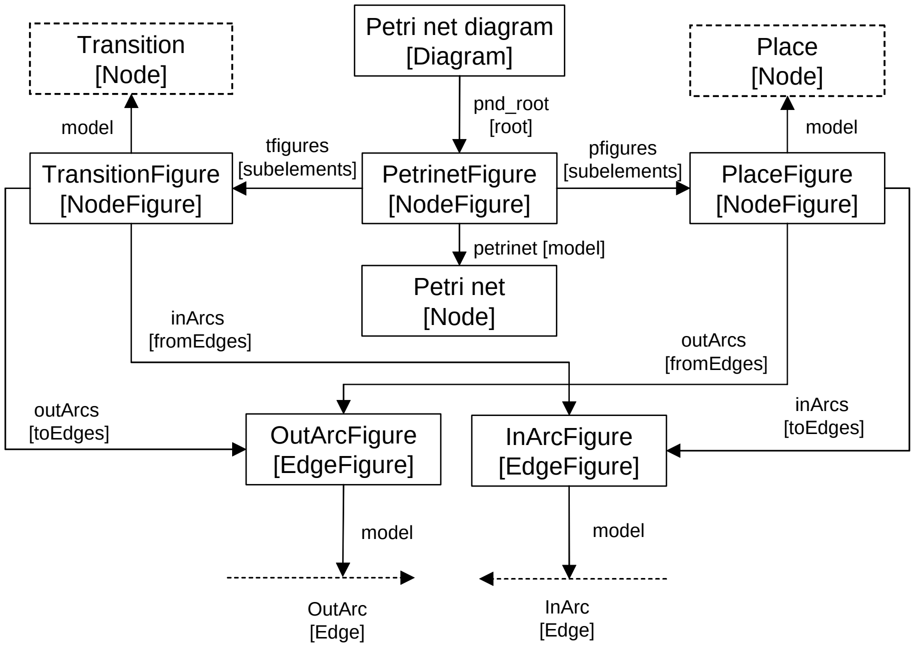


Figure 7.2: Petri net diagram metamodel

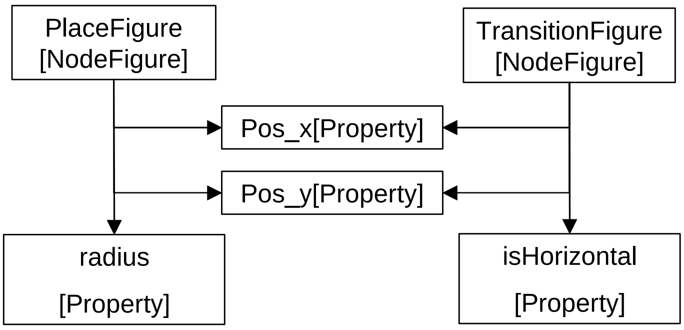


Figure 7.3: Petri net diagram metamodel - properties

will be shown as a digit inside a label, contained by the appropriate PlaceFigure. This means, that direct editing of that label will be mapped to creation/deletion of tokens from the given place.

Since the whole mapping model is fairly large, only the relevant portion of it is shown on Figure 7.4. In that case, the number of tokens property, edited by the user (using direct edit or the property sheet), is bound to the number of Tokens assigned to the Placem, using a command sequence consisting of creation and deletion commands (deletion commands are not shown on Fig. 7.4 for the sake of simplicity). The rest of the mapping can be implemented using similar rules or GTASM transformation programs (for another example, refer to 6.3.6).

Step 6: Implementing custom graphical elements

Arriving at step 6, the domain metamodel and diagram metamodel is already in place, describing both the structure of the domain and the various graphical representations. Up to this point, no manual coding was necessary (except if mapping was implemented using Java code).

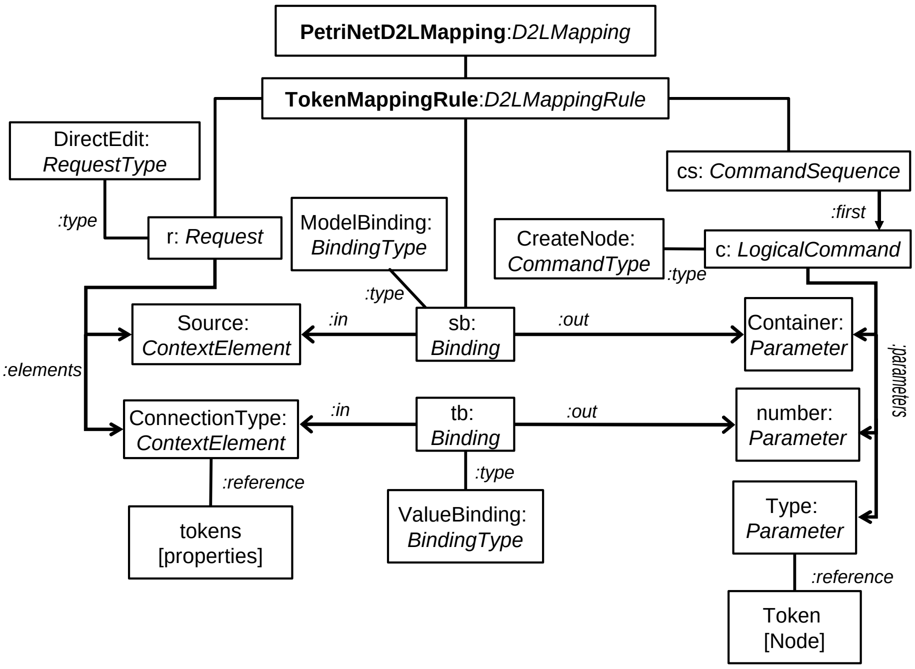


Figure 7.4: Mapping model example: Petri net tokens

If the language engineer decides to stop here, and he does not want to do any Java coding, he has the option to use the default implementation for the views and the graphical editor. That would result in a user interface shown on figure 7.5. All nodes are represented as rectangular boxes, and neither the tools on the palette nor the entries in the tree editor have custom icons.

To use custom graphics, first, the language engineer designs icons, that will be used in the tree and on the palette. That is not a difficult job to do, any application may be used, which can save images in a format understood by Eclipse SWT (that can be PNG, GIF, BMP and may others).

The second thing to do is creating a custom graphics editor, where model elements are not represented as boxes, rather something more domain specific. In case of Petri nets, little circles for places, where the number of tokens in that place is written inside the circle, will be used. Transitions will be represented by small filled black rectangles. Using a GEF-based editor, that requires the engineer to implement these custom figures as Draw2D classes.

Besides providing a large set of pre-defined symbols, Draw2D allows its figures to be completely user-drawn, so every pixel of the graphical editor may be under the control of the language designer. This is not the place to discuss detailed Draw2D coding, but just to have an idea of how much coding is required, the constructor of the PlaceView class is given as an example:

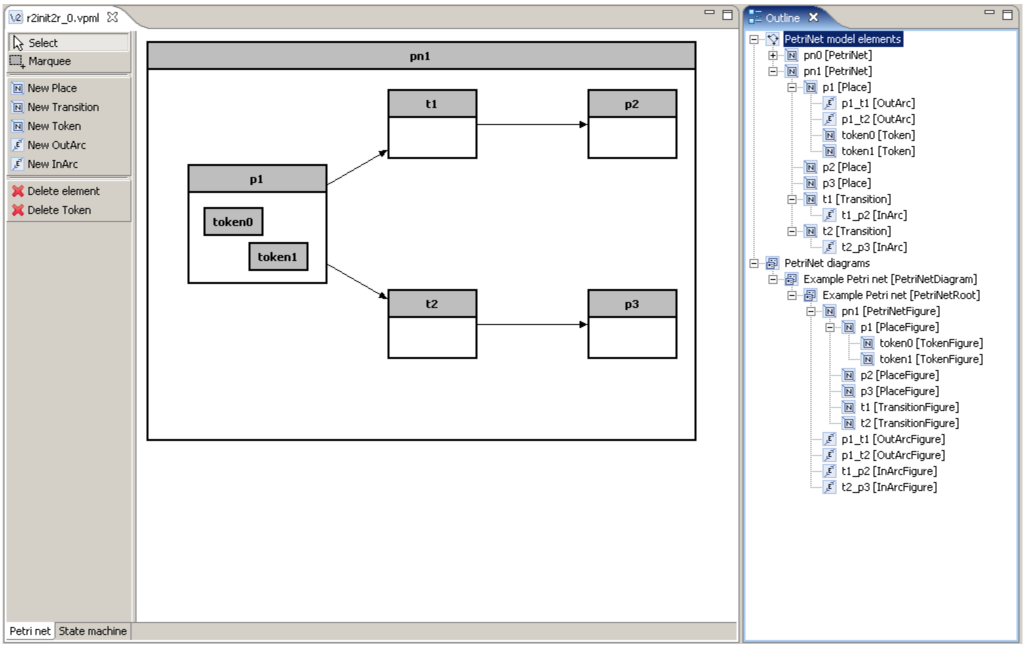


Figure 7.5: Step 5: Petri net editor without custom graphical elements

```
public PlaceView(IDSMMetaDiagramElement meta)
{
  propertyHelper = new DSMViewPropertyHelper(this, meta);

  this.setLayoutManager(new BorderLayout());
  iLabel = new Label();
  add(iLabel, BorderLayout.TOP);

  iPlace = new Ellipse();
  iPlace.setPreferredSize(new Dimension(40, 40));
  add(iPlace, BorderLayout.CENTER);

  iTokenLabel = new Label("0");
  iPlace.setLayoutManager(new BorderLayout());
  iPlace.add(iTokenLabel, BorderLayout.CENTER);
}
```

The whole PlaceView class is 85 lines of Java code long.

Step 7: Bringing it all together

This step requires the construction of the domain editor plugin class, which connects all the different components created so far. In addition to the editor class, a view factory is provided which handles the instantiation of the view classes.

The domain editor plugin class extends the DSMSimpleEditor class provided by the ViatraDSM framework, and overrides a couple of methods allowing the editor to use custom icons and diagram elements. The API documentation is quite extensive, so an engineer with minimal experience in Java should have no problems creating the plugin class. However, the automatic generation of plugin and supporting classes is planned.

Now the DSM editor is ready to use (see figure 7.6. Comparing it to figure 7.5, it can be seen that the icons in the tree and on the palette are replaced by the custom ones, and nodes and transitions have nicer looking figures.

Starting from scratch, the engineer has arrived at a working domain specific editor only by drawing a domain and a diagram metamodel, constructing mapping rules, and writing some code (alltoghether less than 500 lines). But a working tree-only editor is possible even without Java coding.

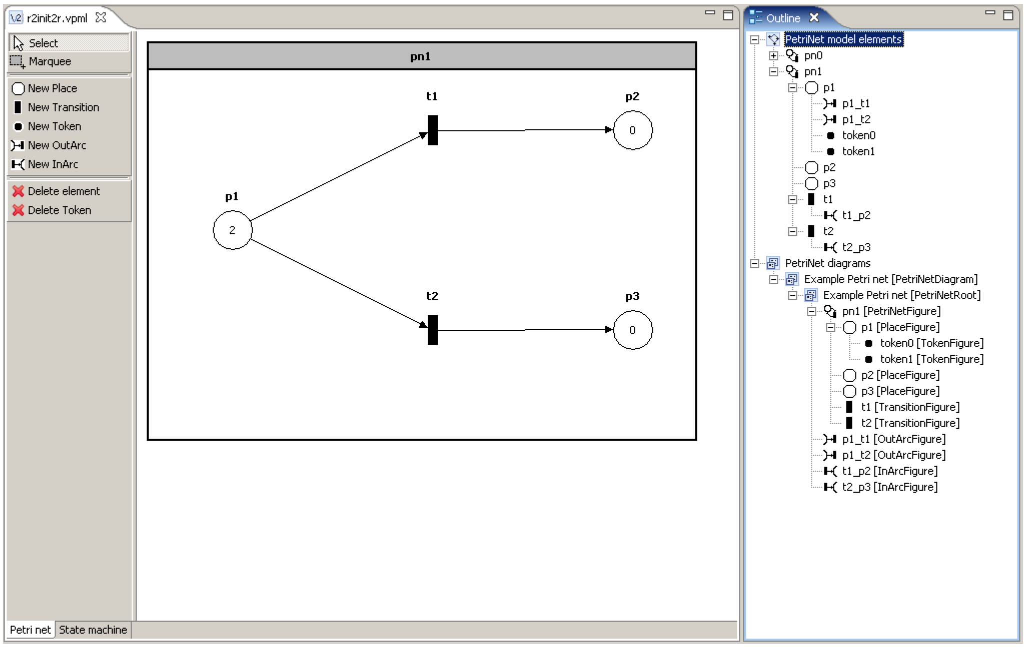


Figure 7.6: Step 6: Petri net editor with custom graphical elements

Step 8: Simulation

In order to enable the design-time simulation for the Petri net editor, the language engineer must describe the dynamic behaviour of Petri nets. These rules must be given as a set of graph transformation rules and abstract state machine programs, in VIATRA2's native transformation language. The syntax and semantics of these rules have been detailed in chapter 3.2.

The following example is a simple Petri net simulator. It simulates one atomic transition of a Petri net. The graph transformation rule that finds an enabled transition has a negative condition as well.

The first code snippet is the frame of the simulation program. The program finds one transition that can be fired at the state that is represented by the model. The second step is to delete tokens from the *input places*. Then tokens on the *output places* are created, and last the mark from the transition to be fired is deleted. The whole procedure is an atomic step of the Petri net.

```
machine psim
{
    rule main()=seq
    {
        // Find a transition and mark it
        choose T below Model do apply mark_fireable(T);
        // Delete tokens from in places
        forall P do apply delete_in_tokens(P);
        // Create tokens at out places
        forall P do apply create_out_tokens(P);
        // Delete the mark from fireable token
        forall T do apply delete_fireable(T);
    }
    ...
}
```

A transition can be fired if it has no input places (the first negative pattern means this). Input places do not block the transition if they are marked. This rule is represented by the second negative pattern (double negation). The transition that can be fired at the state is marked with relation that connects the transition with itself. This new relation is generated by the RHS of the following transformation rule.

```
{
    ...
    gtrule mark_fireable(inout T) =
    {
        precondition pattern lhs(T) =
        {
            petriNet.transition(T);
            // Transition is allowed if it has no in places
            neg pattern t_neg(T)=
            {
                petriNet.transition(T);
                petriNet.place(P);
                petriNet.place.outArc(Out,P,T);
                // Or if all in places has a token
                neg pattern t_negneg(P)=
                {
                    petriNet.place(P);
                    petriNet.token(Tok);
                    petriNet.place.mark(Mark,P,Tok);
                }
            }
        }
        // Mark the transition we found with an edge from and to it
        postcondition pattern rhs(T) =
        {
            petriNet.transition(T);
            petriNet.transition.markTransable(M,T,T);
        }
    }
    ...
}
```

The following rule deletes one token from each in place of the firing transition. This rule is called with *forall* semantics on variable P, so all in places will match this rule once. The variable *Tok* will match for each place one token that is on that place. This variable has no *forall* semantics, so only one token will be selected for each place.

```
    gtrule delete_in_tokens(inout P) =
    {
        precondition pattern lhs(P, Tok) =
        {
            petriNet.transition(T);
            petriNet.transition.markTransable(M,T,T);
            petriNet.place(P);
            petriNet.place.outArc(Out,P,T);
            petriNet.token(Tok);
            petriNet.place.mark(Mark,P,Tok);
        }
        // Tok is not part of RHS pattern, so it will be deleted.
        postcondition pattern rhs(P, Tok) =
        {
            petriNet.place(P);
        }
    }
```

An additional rule creates one token on each output place of the transition.

```
    gtrule create_out_tokens(inout P) =
    {
        precondition pattern lhs(P) =
        {
            petriNet.transition(T);
            petriNet.transition.markTransable(M,T,T);
            petriNet.place(P);
            petriNet.transition.inArc(In,T,P);
        }
        // Tok is not part of LHS pattern, so it will be created.
        postcondition pattern rhs(P) =
        {
            petriNet.token(Tok);
            petriNet.place(P);
            petriNet.place.mark(Mark,P,Tok);
        }
    }
}
```

The last rule deletes the marking relation from the marked transition. After deleting the mark from the firing transition the petri net is in consistent state again and the atomic step of simulation is ready.

```
{
    ...
    gtrule delete_fireable(inout T) =
    {
        precondition pattern lhs(M,T) =
        {
            petriNet.transition(T);
            petriNet.transition.markTransable(M,T,T);
        }
        // The mark is not part of RHS so it will be deleted.
        postcondition pattern rhs(M,T) =
        {
            petriNet.transition(T);
        }
    }
    ...
}
```

The transformation programs have to be connected to the plugin by appropriate mapping models and code segments. The simulation engine was developed by Dávid Vágó, and its detailed description can be found in his masters thesis. After the appropriate mappings are in place, petri nets can be fired in the editor at the user's will. On Figure 7.7, two transitions are marked fireable (the graphical representation of marking, red highlight, has to be coded manually, by modifying the view classes). The user can decide which one to fire.

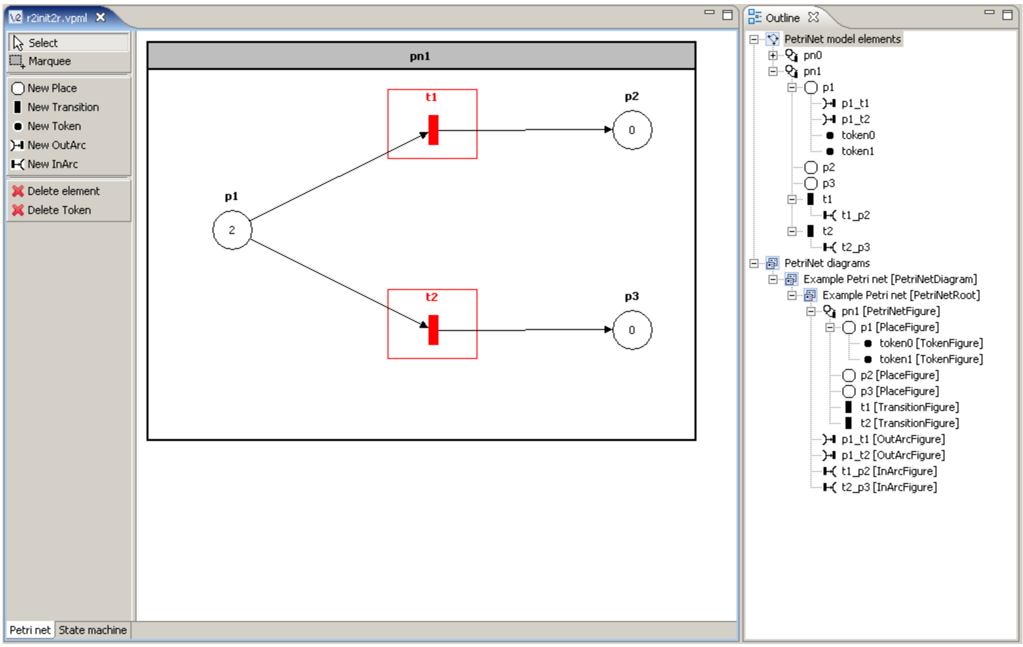


Figure 7.7: Guided simulation of Petri nets

Step 9: Code generation

In the ViatraDSM framework, code generation is also based on VIATRA2 transformations. Code generator transformations were also mentioned in chapter 3.2.

In the case of the current example, code generation is implemented as a PNML generator. Such a code generator transformation iterates over the entire Petri net structure, and at each node (place or transition) it emits snippets of PNML code.

The following example generates a (domain-specific) Petri Net Markup Language file from the Petri net in the VPM model.

```
machine petri2pnml
{
  // This ASM function is a mapping from the VPM entity
  // that represents a place
  // to the count of tokens at that place
  asmfunction tokenCount/1;
  rule main(in ModelName) = seq
  {
    let Model=ref(ModelName) in seq
    {
      // Count tokens at all places
      iterate choose P below Model with find place(P) do seq
      {
        forall Tok do apply count_token(P,Tok);
      }
      // print the heading of the PNML file (open tags)
      call print_head();
      // print all places
      forall P below Model do apply print_place(P);
      // print all transitions
      forall T below Model do apply print_transition(T);
      // pring all edges
      forall I below Model do apply print_in_arc(I);
      forall O below Model do apply print_out_arc(O);
      // print tail of the PNML file (close tags)
      call print_tail();
    }
  }
}
```

The following simple print rules matches on the *place* model element and prints XML tags with appropriate attributes. Similar rules can be constructed for all model elements.

```
gtrule print_place(P) =
{
 // print a place
 precondition lhs(P) =
 {
  petriNet.place(P);
 }
 action
 {
   print("<place id=\"");
   print(name(P));
   print("\">");
   print("<name><text>");
   print(name(P));
   print("</text></name>");
   print("<initialMarking><text>");
   print(tokenCount(P));
   print("</text></initialMarking>");
   print("</place>");
  }
}
```

The model stores each token as a separate entity, but PNML requires the number of them, so they must be counted. The following rule matches on a place with a token, and updates the appropriate entry in the *tokenCount/1* ASMFunction.

```
gtrule count_token(in P, inout Tok) =
{
 precondition lhs(P, Tok) =
 {
   petriNet.place(P);
   petriNet.token(Tok);
   petriNet.place.tokens(Mark,P,Tok);
 }
 action
 {
   if(tokenCount(P)==undef) tokenCount(P)=0;
   update tokenCount(P)=tokenCount(P)+1;
```

```
  }
}
```

The code generator transformation is integrated into the ViatraDSM framework through some additional coding (overriding a method) in the plugin editor class.

Step 10: Constraint checker

The Petri net metamodel also supports place capacity, which is a positive integer, assigned to places, it is logical to enforce the validity of this language-specific constraint using the functionality provided by the modeling environment. The constraint on the model is to have a less or equal number tokens at each place than the capacity of the given place.

The following code can be used to check the constraint:

```
[...]
rule main(in ModelName) = seq
{
  let Model=ref(ModelName) in seq
  {
    // Count tokens at all places
    forall P below Model with find place(P) do seq
    {
      // the count_token rule from the previous
      // example is called at this point.
      forall Tok do apply count_token(P,Tok);
        call checkPlace(P);
    }
  }
}
```

The following rule checks whether the condition specified by the constraint holds on the matched place-capacity subgraph:

```
rule checkPlace(in P) = seq
{
  choose C with find capacity(P, C) do seq
  {
     // this machine uses the same tokenCount/1 ASMFunction
     // as the previous example
     if(value(C)<tokenCount(P)) do
       log("Error: too many tokens on place: "+name(P)+"\n");
  }
}
```

This pattern selects the appropriate place-capacity pairs from the modelspace:

```
 pattern capacity(P, C) =
 {
    petriNet.place(P);
    petriNet.capacity(C);
    petriNet.place.capacity(CO,P,C);
 }
}
```

Using the editor

The domain specific editor is now ready. The end user (domain expert) can easily create and edit Petri nets, using domain specific graphical elements. By default, the Petri nets and diagrams are saved along with the entire model space in VPML, the native XML format of VIATRA2 models. The editor can simply generate standard PNML description with a single click, interactive simulation can be started from the context menu, and models can be checked against a simple constraint to check the capacities of places.

Chapter 8

Conclusions

Today, domain specific modeling gathers ground in the area of software development. Unlike UML, which tries to be a single modeling language as general as possible, DSM follows the idea of creating separate modeling languages for every specialized application domain.

Since domain specific models are on a higher abstraction level than general UML models, application source code is easier to generate and many times results in a more comprehensive generation process, consequently a leap in developer productivity. However, whilst for UML modeling there are a number of industrial tools available, domain specific modeling languages are harder and more expensive to construct, simply because of the complexity of the task. Thus, language engineering frameworks are being developed, at software houses ranging from small open source projects to industry giants like Microsoft.

At the beginning of our research, I studied many existing and upcoming language engineering tools to get a good and thorough view of what others were trying to achieve, and what could be possible with this technology. The result of this initial research is presented in this thesis in Chapter 2. The most important conclusions were the following:

- *Weak domain integration.* Most of existing DSM tools separate the models of the distinct domains (by generating standalone editors), therefore modeling the interaction between heterogenous components of a complex system is difficult.
- *No integrated support for simulation and model checking.* While some vendors experiment with enabling some simulation abilities based on pre-defined dynamic patterns such as finite state machines, this area of language engineering is widely ignored in current tools. Additionally, whilst support for the checking of more complex constraints based on the Object Constraint Language (OCL) is being developed for GMF, for instance, this is not based on a model transformation approach.
- *Simplistic approach to diagrams.* With the exception of the most modern approach, GMF, language engineering frameworks are stuck at the level of "class → node, association → edge" type of diagram mapping. This is acceptable for simple domains, however, particularly if abstract syntax is optimised for code generation, or model transformation purposes, a separate visualisation modeling layer that can be tailored to the needs of the human user, is hardly omittable.

Interestingly, the initial conceptualisation of the ViatraDSM framework began at approximately the same time as GMF was drafted. Many of the ideas that the designers of GMF have come up with are very straightforward, and since we missed those features from existing modeling environments as well, it was natural to add them to the ViatraDSM framework. Therefore, many aspects of ViatraDSM and GMF are very similar, as both use GEF and Eclipse.

The key difference, however, which separates our design from all other language engineering products is the fundamental idea of encompassing all important engineering aspects in the model transformation domain. This concept arose from the fact that most of the implementation behind the VIATRA2 framework is the work of András Schmidt, Dávid Vágó and the author of this thesis. The first motivation to develop the ViatraDSM framework, therefore, was that while VIATRA2 was a promising transformation tool, it lacked an intuitive modeling interface.

Based on our implementation, the VIATRA2 framework has become an official subproject of the Eclipse Generative Model Transformers project [49] in September, 2005. This open source contribution is dominantly based on the results of our Scientific Students' Association report, and the masters theses of András Schmidt, Dávid Vágó and myself.

The ViatraDSM framework was designed and developed in strong co-operation with Dávid, and we also got valuable feedback from colleagues, especially Dániel Tóth, who used it for his own master's thesis. The results emphasized in this thesis, *multi domain modeling* and the *diagram modeling layer* are my own work, carried out during the last semester.

The ViatraDSM framework Our achievement presented in this thesis, the ViatraDSM framework, is a domain specific language engineering environment built on top of the model transformation capabilities of the VIATRA2 engine. The design process, as documented in this thesis, produced the following results:

- A formal method for the precise specification of domain specific visual languages has been developed (the DSM Core metamodel, Sec. 4.1.3).
- This specification was projected into the VIATRA2 modelspace, along with a prototype implementation.
- Additionally, model transformation-based technologies were developed to uniformly specify constraints, simulation rules, and code generation, using VIATRA2's native transformation languages (Chapters 4, 3, and 7).
- The implementation of the ViatraDSM framework prototype was refined, resulting in the code base that is described in detail in Chapter 4. The main advantages of our implementation are:
  - it can be adapted to arbitrary modeling frameworks;
  - it is a memory efficient implementation;
  - it supports modeling and mode transformation in multiple domains;
  - it enables the model-based development of domain-specific visual languages (without manual coding) but also provides a flexible solution for language engineers by manual coding.

Multi domain modeling As an individual project, I developed the fundamentals for multi-domain modeling based on multiple aspects (Chapter 5). I described (in Sec. 5.4) light-weight techniques for integrating multiple domain-specific modeling languages into a consistent system model in addition to traditional model transformation based domain integration. Using these novel techniques, integration of DSMLs can be specified very easily compared with designing a complex model transformation for the same problem in many cases. Then I presented (in Sec. 5.5) how these concepts can be used to create multi-domain models in the ViatraDSM framework.

Abstract - concrete syntax separation I designed multiple techniques to facilitate the mapping between the abstract syntax and concrete syntax model layers of visual languages (Chapter 6). The features implemented are very important, both from the perspective of the language engineer and the end user. The *language engineer* benefits from greater freedom in designing the concrete and abstract syntax representations of the language, while still retaining the possibility to create default mappings between the two layers in a simple fashion. *End users* experience greater freedom in constructing models visually.

The most important features of the implementation are:

- The mapping technique allows for arbitrary bidirectional mapping between abstract and concrete syntax models (logical and diagram models), and provides declarative support based on a mapping metamodel and VIATRA2 transformations.
- Diagram and logical models share the same modelspace, making VIATRA2-native GTASM transformations between abstract and concrete syntax models possible.
- The mapping interfaces follow the plugin architecture of the ViatraDSM framework by allowing the plugin author to design and implement mapping in both directions in a flexible way (e.g. by using pure Java code, or mostly declarative techniques).
- The mapping metamodel allows for future automatic generation based on a concrete syntax metamodel, further minimizing the amount of manual coding required to build a domain-specific visual editor.

Case study Finally, I demonstrated the feasibility of our approach by discussing a descriptive language engineering example, the creation of a Petri net editor with simulation, code generation, and model checking capabilities. In ten steps, such a modeling tool can be constructed with minimal Java coding (mostly writing simple view classes for custom graphical representation and some "glue code"). However, as the range of possibilities offered by the ViatraDSM framework is wide, a simple, tree-based domain specific editor can be created with an absolute minimum of manual coding.

Future work

From an architectural viewpoint, the most missing feature of the ViatraDSM framework is the lack of an automated mechanism of *enforcing complex language-specific constraints*. While contraint-checking transformations can be run, there is no way of guaranteeing the "correctness" of models, apart from simple static constraints supported natively by the underlying VPM modeling infrastructure (containment, multiplicity, type correctness). It is important to note that this feature is missing from *all* domain-specific frameworks, because enforcing constraints in an on-line fashion is not always possible, because there may be scenarios where a temporarily invalid model state must be accepted so that the user can finish her editing actions which may lead to a correct model. Most tools check OCL constraints at user request, and highlight invalid model parts. This is one important feature which will be added to the ViatraDSM framework in the near future.

Concerning graphical representation, there are two major aspects of the language engineering process which are currently time consuming and complicated for the designer. One of these aspects is the construction of visualisation classes, where Java coding is required. A straightforward idea is to implement a simple visual editor for Draw2D classes, similarly to the one found in many DSM tools, such as Microsoft's DSL Tools suite, and generate source code which can be used by the framework. The other aspect is the mapping mechanism. While considerable efforts were made to simplify a generally complicated problem of model synchronisation to an acceptable level, a mechanism which would enable the (at least partial) automatic generation of mapping rules would significantly reduce the time required to construct a visual language.

On the implementation side, the ViatraDSM framework does not currently offer the rich rendering capabilities found in professional software like GMF or MetaEdit+. In fact, I have attempted utilizing GMF's presentation layer over the ViatraDSM infrastructure, however GMF proved to be too intergated with EMF, and the EMF philosophy (generated editors) for this to be possible. In this area, much could be improved, however ViatraDSM was primarily conceived as a prototype to test and develop new ideas, we did not consider the development of fancy graphics a priority.

The most important goal to reach is the total elimination of Java coding. Currently, visualisation classes, as well as "glue code", logic connecting the various compoments that make up a domain specific editor (including the plugin class and helper classes) need to be implemented manually. However, as the code generation capabilities of VIATRA2 are powerful enough, we plan to enable the automatic generation of these code fragments using VIATRA2-based models.

Bibliography

[1] AGG Project website
http://tfs.cs.tu-berlin.de/agg/

[2] Apache Velocity Project
http://jakarta.apache.org/velocity

[3] b+w GmbH website
http://www.architectureware.de

[4] András Balogh, Dániel Varró: Advanced Model Transformation Language Constructs in the VIATRA2 Framework, September 2005, Accepted to SAC 2006, Model Transformation Track

[5] E. Börger and R. Särk: Abstract State Machines. A method for High-Level System Design and Analysis, Springer-Verlag, 2003.

[6] Compuware: Landmark Study Reveals 35 per cent Productivity Gains for Businesses Using Model-Driven Architecture, 21 July, 2003
http://www.compuware.co.uk/pressroom/news/21072003_02.htm

[7] Eclipse Project website
http://www.eclipse.org

[8] Eclipse Modeling Framework website
http://www.eclipse.org/emf

[9] Eclipse Corner Articles: Introduction to Java Emitter Templates (JET)
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html

[10] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.): Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, in: World Scientific, 1999.

[11] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Gabriele Taentzer, Daniel Varro, and Szilvia Varro-Gyapay: Model Transformation by Graph Transformation: A Comparative Study, Model Transformations in Practice Workshop, 2005. In press.

[12] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, Gabriele Taentzer: Towards Graph Transformation Based Generation of Visual Editors using Eclipse
http://tfs.cs.tu-berlin.de/~karstene/public/gEEHT04.pdf

[13] C. Ermel and M. Rudolf and G. Taentzer: In [10], chapter The AGG-Approach: Language and Tool Environment, World Scientific, 1999, pages 551–603.

[14] David S. Frankel - Steve Cook: Domain-Specific Modeling and Model Driven Architecture, in: MDA Journal, January 2004
http://www.bptrends.com/publicationfiles/01-04COLDomSpecModelingFrankel-Cook.pdf

[15] General Modeling Framework website
http://www.eclipse.org/gmf

[16] GMF Project Requirements
http://www.eclipse.org/gmf/requirements.html

[17] Jack Greenfield: The Case for Software Factories, in: ITArchitect Magazine, August 24, 2004
http://www.itarchitect.co.uk/articles/display.asp?id=96

[18] Jack Greenfield: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, in: Microsoft Developer Network Technical Articles, November 2004
http://msdn.microsoft.com/library/en-us/dnbda/html/softfact3.asp

[19] Martijn Iseger: Domain-specific modeling for generative software development, April 17, 2005
http://www.itarchitect.co.uk/articles/display.asp?id=161

[20] Steven Kelly: Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM, in: OOPSLA: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004
http://www.softmetaware.com/oopsla2004/kelly.pdf

[21] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J: Aspect-oriented Programming, in: Proceedings of ECOOP, 1997
http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[22] Paul Klint and Ralf Lämmel and Chris Verhoef: Toward an engineering discipline for grammarware. In ACM Transactions on Software Engineering Methodology, Vol.14, No.3, 2005. pp.331-380.
http://doi.acm.org/10.1145/1073000

[23] Thomas Klein and Ulrich Nickel and Jörg Niere and Albert Zündorf: From UML to Java And Back Again, University of Paderborn, 2000, tr-ri-00-216

[24] Kobryn, C.: UML 2001: A standardization Odyssey. Communications of the ACM, 42(10), 1999

[25] Bernd Kolb, Markus Völter: openArchitectureWare and Eclipse, 2004
http://architecturware.sourceforge.net/data/oawEclipse.pdf

[26] J. Larrosa and G. Valiente: Constraint Satisfaction Algorithms for Graph Pattern Matching, in: Mathematical Structures in Computer Science, Vol. 12, No. 4, 2002, pages 403–422.

[27] MetaCase website
http://www.metacase.com

[28] MetaEdit+: Technical Summary
http://www.metacase.com/papers/MetaEditPlus.pdf

[29] Meta Object Facility Specification
http://www.omg.org/docs/formal/02-04-03.pdf

[30] Microsoft DSL Tools website
http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx

[31] Microsoft DSL Tools Walkthroughs
http://go.microsoft.com/fwlink/?LinkId=43636

[32] William Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, in: IBM RedBooks, 17 February 2004
http://www.redbooks.ibm.com/abstracts/sg246302.html

[33] U. Nickel, J. Niere, and A. Zündorf: Tool demonstration: The FUJABA environment, in: The 22nd Int. Conf. on Software Engineering (ICSE), ACM Press, Limerick, Ireland, 2000.

[34] The Object Management Group: UML Profile for Schedulability, Performance, and Time Specification,
http://www.omg.org/cgi-bin/doc?formal/2005-01-02

[35] openArchitectureWare website
http://www.openarchitectureware.org

[36] The Petri Net Markup Language Specification
http://www.informatik.hu-berlin.de/top/pnml/pnml.html

[37] Dorin Bogdan Petriu and C. Murray Woodside: A Metamodel for Generating Performance Models from UML Designs, UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings, 2004. editors: = Thomas Baar and Alfred Strohmeier and Ana M. D. Moreira and Stephen J. Mellor, Vol. 3273, Ser. Lecture Notes in Computer Science, Springer.
ftp://ftp.sce.carleton.ca/pub/cmw/csm-uml04.pdf

[38] Risto Pohjonen: Boosting Embedded Systems Development with Domain-Specific Modeling, 2004
http://www.metacase.com/papers/RTC04_Pohjonen.pdf

[39] A. Rensink: Representing first-order logic using graphs, in: H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), Proc. 2nd Int. Conf. on Graph Transformation (ICGT 2004), Rome, Italy, vol. 3256 of LNCS, pp. 319335. Springer, 2004.

[40] Rich Seeley: ADT at Gartner ITxpo: Gates sees more modeling, less coding, 30 March, 2004
http://www.adtmag.com/article.asp?id=9166

[41] Charles Simonyi: The Death of Computer Languages, The Birth of Intentional Programming, 1995
http://research.microsoft.com/research/pubs/view.aspx?type=TechnicalReport&id=4

[42] Dave D. Straube and M. Tamer Özsu: Query Optimization and Execution Plan Generation in Object-Oriented Data Management Systems, Knowledge and Data Engineering, 1995, Vol. 7, No. 2, pages 210–227.

[43] Tiger Project website
http://tfs.cs.tu-berlin.de/tigerprj/

[44] Tiger User Documentation
http://tfs.cs.tu-berlin.de/tigerprj/papers/userdoc.pdf

[45] Dániel Varró: Automated Model Transformations for the Analysis of IT Systems, PhD Thesis, 2004
http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2004/phd_thesis.zip

[46] Dániel Varró, András Pataricza: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics), in: Journal of Software and Systems Modeling, October, 2003

[47] Dániel Varró and Gergely Varró and András Pataricza: Designing the Automatic Transformation of Visual Languages, Vol. 44, No. 2, Elsevier, pages 205–227, in: Science of Computer Programming, August 2002

[48] Gergely Varró, Katalin Friedl, Dániel Varró: Graph Transformation in Relational Databases, In Proc. GraBaTs 2004: International Workshop on Graph Based Tools. In press.

[49] *VIATRA2 Framework*. An Eclipse GMT Subproject.
http://www.eclipse.org/gmt/

[50] Attila Vizhanyo and Aditya Agrawal and Feng Shi: Towards Generation of Efficient Transformations, Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004), pages 298–316, October 2004, editors: Gabor Karsai and Eelco Visser, Vol. 3286, Ser. LNCS, Vancouver, Canada, Springer-Verlag

[51] VMTS Website
http://avalon.aut.bme.hu/~tihamer/research/vmts

[52] David S. Wile: Abstract Syntax from Concrete Syntax. In Proceedings of the 19th International Conference on Software Engineering. Boston, MA. May, 1997. 472-480.

[53] A. Zündorf: Graph pattern-matching in PROGRES, Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, pages 454–468, 1996, Vol. 1073, Ser. LNCS, Springer-Verlag