



M Ű E G Y E T E M 1 7 8 2

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

# **Live Model Transformations in Domain-Specific Modeling**

Master's Thesis

**András Ökrös**

Supervisor:

**István Ráth**

PhD student

Budapest, 16 May 2008

*The author would like to thank the supervisor István Ráth for his continued support, friendly advice, and enthusiasm. I would also like to thank Dr. Dániel Varró, Gergely Varró and Dániel Tóth for their numerous valuable observations and related suggestions.*

## Nyilatkozat

Alulírott Ökrös András, 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.

.....  
Ökrös András

## Összefoglaló

Napjainkban a modellbázisú rendszerfejlesztési paradigma térhódítása figyelhető meg. E módszertan a fejlesztési folyamatot egy precíz modellezési lépéssel kezdi, legtöbbször a szabványos UML modellezési nyelv felhasználásával. Ezekből a rendszermodellekből az alkalmazások végrehajtható kódját automatikus kódgenerálás segítségével származtathatjuk.

Az ipari modellvezérelt fejlesztési folyamatok tapasztalatai azonban azt mutatják, hogy az UML általános fogalmai nem elégitik ki minden esetben az adott alkalmazási terület (domain) specifikus igényeit. Továbbá a rendszerünket tipikusan több nézőpontból szükséges terveznünk, ahol az egyes nézőpontok modellezésére külön domain-specifikus modellezési nyelvet használunk. Az UML 2.0 szabvány is ezt a filozófiát követi: az egyes UML diagramok felfoghatók egy-egy (grafikus) domain-specifikus nyelvnek (DSL), de az adott alkalmazási terület igényeinek megfelelően további modellezési nyelvek is felhasználhatók a tervezés során. A DSL technológia széleskörű alkalmazhatóságát jelzi, hogy az ipari megoldások (pl. Eclipse EMF és GMF keretrendszerek) az elmúlt két év során széles körben elterjedtek. Ezek az eszközök magasszintű támogatást nyújtanak komplex, grafikus domain-specifikus nyelvek fejlesztéséhez, azonban továbbra is igaz az, hogy az ilyen nyelvek kifejlesztése még mindig drága és a létező megoldások nehezen újrahaznosíthatók.

A domain-specifikus nyelvek fejlesztésénél és testreszabásánál a jelenlegi technológiák csak kezdetleges támogatást nyújtanak több fontos nyelv-tervezési aspektusra, mint például a nyelv különböző megjelenési formáinak (azaz a logikai és a grafikus reprezentációk) szinkronizálása. Hasonlóan, az egyes nyelvek és eszközök integrációja, ahol nyelvközi leképezésekkel állítjuk elő a globális és koherens rendszermodellt is egy hátralévő nehéz kihívás. Továbbá, ahogy a domain-specifikus modellezési környezetek fejlődnek, az igény olyan fejlett szolgáltatásokra, mint például az összetett nyelv-specifikus kényszerek azonnali kiértékelése, is egyre nagyobb. Tekintve, hogy ezek a problémák megfogalmazhatók modell transzformációs feladatként, egy lehetséges megoldás lehet, ha a domain-specifikus modellezési környezetet integráljuk egy modell transzformációs rendszerrel.

Egy ilyen rendszer alapjait fektettük le korábbi munkánk [4] során. Az ott ismertetett eredmények folytatásaként és kiterjesztéseként bemutatom a következőket:

- Kiegészítettem a VIATRA2 eseményvezérelt transzformációs keretrendszerét oly módon, hogy az támogassa a soros és a párhuzamos végrehajtási szemantikát, amely megadja az alapját az eszközbe integrált inkrementális transzformációknak.
- A kiegészített rendszerre építve megterveztem és megvalósítottam egy új magasszintű támogatást a domain-specifikus vizuális nyelvek logikai és grafikus reprezentációinak (absztrakt-konkrét szintaxisának) automatikus szinkronizálásához.
- Végül megterveztem és megvalósítottam egy csatoló felületet a VIATRA2 DSL rendszere és az iparban széles körben használt Eclipse Graphical Modeling Framework között, hogy ezzel is segítsen a VIATRA2 alapú domain-specifikus grafikus nyelvek fejlesztését.

## Abstract

Nowadays, the model-based software development paradigm has gained significant popularity. This paradigm begins the development process with a precise modeling step, usually based on UML. Following that, the application's source code is generated from these models using automated code generation technology.

Industrial experience with model driven development has shown 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 as a (graphical) domain specific language (DSL), but it can also be extended to support new languages for modeling in a special domain. The widespread applicability of the DSL technology can be supported with the fact that industrial solutions (e.g. Eclipse EMF and GMF frameworks) have gained widespread use over the last two years. While these tools have high-level support for developing complex and visual domain specific languages, the development process is still expensive, and existing solutions cannot be reused without difficulties.

For the development and customization of domain-specific languages, current technologies have only basic support for a number of crucially important language engineering aspects, such as the synchronization between the various representations of the language (i.e. logical and graphical representations). Similarly, the integration of different languages and tools where a global and coherent system model is generated from domain specific submodels, remains a difficult challenge. Moreover, as domain-specific modeling environments evolve, demand for advanced features such as the on-line evaluation of complex language specific constraints is emerging. Since these problems can be formulated as model transformation tasks, a possible solution for these challenges can be a domain-specific modeling environment integrated with a model transformation system.

In [4], we have established the foundations of such an integrated system. As a continuation and extension to our results, I present the following contributions in the current report:

- I extended the event-driven model transformation engine of VIATRA2 to support serial and parallel execution semantics, which provides the basis for tool-integrated incremental transformations.
- Based on the improved engine, I developed a new high-level support for the automatic synchronization of logical and graphical representations (abstract-concrete syntax) in domain-specific visual languages.
- Finally, I designed and implemented an automated software bridge between VIATRA2's DSM environment and the industry standard Eclipse Graphical Modeling Framework, to aid the rapid development of VIATRA2-based domain-specific visual languages.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Model based development in software engineering . . . . .	7
1.1.1	Model Driven Architecture . . . . .	7
1.1.2	MDA Development steps . . . . .	8
1.2	Transformations in MDA . . . . .	8
1.3	Domain-Specific Modeling . . . . .	9
1.4	Problems and challenges in domain-specific modeling . . . . .	9
1.5	Objectives . . . . .	11
1.6	Structure of the Report . . . . .	11
<b>2</b>	<b>Background technologies and concepts</b>	<b>12</b>
2.1	Model transformation in VIATRA2 . . . . .	12
2.1.1	Metamodeling: Definition of Abstract Syntax . . . . .	12
2.1.2	The VTCL language . . . . .	13
2.1.3	VIATRA2 Architectural overview . . . . .	19
2.2	The ViatraDSM framework . . . . .	21
2.2.1	Domain-specific modeling in depth . . . . .	21
2.2.2	Architecture . . . . .	22
2.2.3	Modeling . . . . .	23
2.2.4	Diagrams . . . . .	25
2.2.5	Required steps for a complete domain-specific editor . . . . .	27
<b>3</b>	<b>Event-driven model transformations with triggers</b>	<b>29</b>
3.1	Goal of this Chapter . . . . .	29
3.2	Incremental pattern matching . . . . .	30
3.2.1	Pattern matching . . . . .	30
3.2.2	Tuples and Nodes . . . . .	30
3.2.3	Joining . . . . .	31
3.2.4	Updates after model changes . . . . .	31
3.3	Event-driven model transformations with triggers . . . . .	32
3.3.1	The definition of the trigger concept . . . . .	32
3.3.2	Adapting triggers into the VIATRA2 model transformation framework . . . . .	33
3.3.3	Trigger example . . . . .	33
3.4	Execution semantics . . . . .	34
3.4.1	Events and the delta set . . . . .	34
3.4.2	Operational semantics . . . . .	35
3.5	Handling synchronizational problems . . . . .	38
3.5.1	Description of the problem . . . . .	38

3.5.2	Our approach . . . . .	38
3.6	Architecture of the implementation . . . . .	39
3.6.1	Delta Monitor . . . . .	39
3.6.2	Transaction handling . . . . .	40
3.6.3	VIATRA2 user interface extension . . . . .	41
3.7	Summary . . . . .	41
<b>4</b>	<b>Live model transformations in domain-specific modeling</b>	<b>42</b>
4.1	Goal of this Chapter . . . . .	42
4.2	Challenges in the domain-specific modeling world . . . . .	42
4.2.1	Separation of abstract and concrete syntax representations . . . . .	42
4.2.2	Evaluation of complex constraints . . . . .	43
4.3	Generic abstract-concrete syntax synchronization . . . . .	44
4.3.1	Architectural changes in ViatraDSM . . . . .	44
4.3.2	Generic triggers . . . . .	45
4.3.3	Conclusion . . . . .	47
4.4	Incremental constraint checking . . . . .	48
4.4.1	Architectural changes in ViatraDSM . . . . .	48
4.4.2	Triggers . . . . .	49
4.4.3	Conclusion . . . . .	51
4.5	Summary . . . . .	52
<b>5</b>	<b>Graphical Modeling Framework integration</b>	<b>53</b>
5.1	Goal of this Chapter . . . . .	53
5.2	Presentation of the GMF technology . . . . .	53
5.3	Demonstrating example . . . . .	55
5.4	Proposed usage workflow . . . . .	56
5.5	Importing the GMF definition models . . . . .	57
5.6	Processing the GMF definition models . . . . .	57
5.6.1	Processing the <i>domain model</i> . . . . .	57
5.6.2	Processing the <i>tooling definition</i> . . . . .	58
5.6.3	Processing the <i>graphical definition</i> . . . . .	58
5.6.4	Processing the <i>mapping model</i> . . . . .	59
5.7	Summary . . . . .	60
<b>6</b>	<b>Summary</b>	<b>62</b>
6.1	Overview . . . . .	62
6.2	Scientific contributions . . . . .	62
6.3	Practical accomplishments . . . . .	62
6.4	Future work . . . . .	63
<b>A</b>	<b>Generic DSM diagram triggers</b>	<b>64</b>

# Chapter 1

## Introduction

### 1.1 Model based development in software engineering

#### 1.1.1 Model Driven Architecture

In 2001, the Object Management Group started an initiative called *Model Driven Architecture* [19] to provide a visionary paradigm of software development. MDA relies on OMG's other flagship technologies: Unified Modeling Language (UML) [22], the Meta Object Facility (MOF) [21], XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM).

On the temporal scale, MDA encompasses the complete life cycle of designing, deploying, integrating, and managing applications. On the horizontal scale, MDA supports evolving standards in application domains as diverse as enterprise resource planning, air traffic control and human genome research. These MDA-based standards are tailored to the needs of diverse application domains, with the promise of surviving changes in technology and enabling the organizations to integrate their past achievements and readily available resources with present and future developments.

MDA was designed with various goals in mind: portability and reusability, cross-platform interoperability, platform independence, domain specificity, productivity.

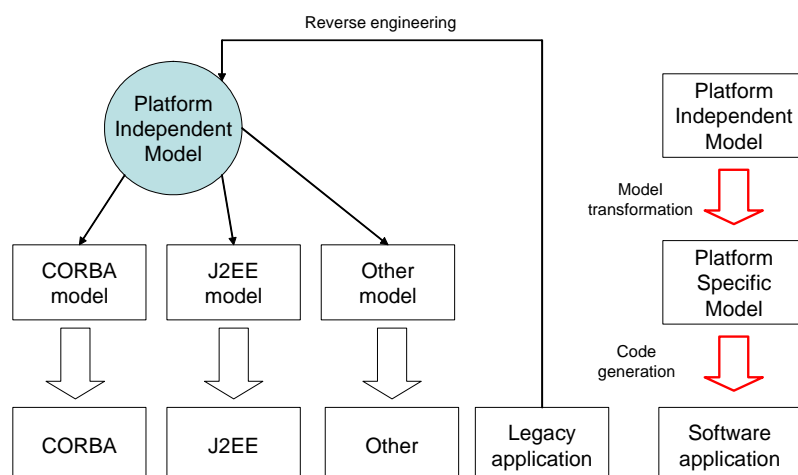


Figure 1.1: Model Driven Architecture

### 1.1.2 MDA Development steps

As it can be seen in 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 Platform 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) [20], and behavioral semantics described in Action Semantics (AS) [18] language.

The second step is to generate a Platform 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 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.

## 1.2 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 [34]:

- 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 released the QVT [17] standard.

### Graph transformation and metamodeling

For many years, the abstract syntax of UML (and related profiles) has been defined visually by means of metamodeling. Metamodeling is a term for capturing the design of user models and modeling languages uniformly, in a single modeling framework. A straightforward representation

of such models and languages can rely on the use of directed, typed, and attributed graphs as the underlying semantic domain. In this sense, graph transformation [27] has recently become very popular as being a general, rule-based visual specification paradigm to formally capture (i) requirements, constraints and behavior of UML-based system models, and (ii) the operational semantics of modeling languages based on metamodeling techniques. Similar ideas are applied directly on formalizing transformations from UML into various semantic domains (Petri nets, dataflow nets, etc.). [32]

A number of graph transformation tools have emerged, including VIATRA2, which provides the foundation of this paper.

### 1.3 Domain-Specific Modeling

*Domain-Specific Modeling* (DSM) is a new approach to model-based software development. 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. pharmaceuticals, business processes, civil engineering). 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.

DSM gives the designer the freedom to use structures and logic that is specific to the target application domain, thus it is completely independent of programming language concepts and syntax. This is a more flexible solution, in contrast with MDA, which emphasizes the importance of a single and universal modeling language at the center of the development process. Moreover, these domain models (with appropriate visualization) are more easily readable and usable for the domain experts, than UML diagrams. This is one of the biggest advantages of DSM, because developers usually have limited knowledge about the application domain, which can result in malfunctioning programs.

In most commercial DSM frameworks, source code is generated directly from domain models, which makes portability difficult, because separate code generators have to be written for various platforms. Although MDA and DSM are usually considered as rival technologies, they can be combined to achieve better results (e.g. the idea of separation PIM and PSM can be used in DSM as well). This requires model-to-model transformation abilities from the DSM framework.

Nowadays, DSM is one of the most important trends in model based development along with UML and MDA. Therefore easily usable, flexible, widely applicable DSM frameworks are needed in the industry.

### 1.4 Problems and challenges in domain-specific modeling

Nowadays, many experts consider tool integration to be the most important challenge, mainly because industry tools are usually not compatible with each other. DSM tools are used by developers, so in real world applications the usability (e.g. reusability of metamodels) is a very important factor. The integration of these modeling tools to the current development process should be as seamless as possible. So it is not a surprise, that framework integrated tools, like the Eclipse Modeling Framework (EMF) [29], or the Eclipse Graphical Modeling Framework (GMF) [31] are gaining popularity over the standalone model transformation- / domain-specific modeling tools.

The integration between modeling tools is equally important. In this way, for example, domain-specific editors created in one of the tools could be reused in the other, which evidently leads to

faster development, if this tool is better suited for the actual problem domain. This tool-to-tool mapping inevitably leads us to model transformations, and more specifically graph transformations, which are well-suited for handling such problems.

Additionally, there is increasing demand from developers for high-level support of important features such as the synchronization of the logical (abstract syntax) and graphical (concrete syntax) representations of the domain models. Currently, modeling environments are not capable of a conceptual separation between these modeling layers, which means that concrete syntax models are structurally bound to abstract syntax and thus the environment lacks the necessary abstraction power. This is a crucial problem, because one of the main goals of domain specific modeling was to provide an easily readable and usable visualization for domain experts. Although the speed of development is always a key issue, the support for automatic synchronization for most of the metamodel elements is equally important.

Another key issue is the on-the-fly evaluation of complex language specific constraints (e.g. design pattern conformity checking), which is important, because the sooner an error is found, the less it costs to repair. Generally, this is computationally expensive problem. A possible solution in state-of-the-art tools is to validate models by running batch evaluators on request, which is still time consuming, especially on large models, and does not provide instantaneous and easy-to-understand feedback about errors. For certain constraints, there is a possibility to be evaluated and enforced as the user is editing the model. Unfortunately, support for this in current tools is only emerging, in fact, it is usually limited to checking simple static constraints (e.g. attributes, multiplicities). The challenge here is to identify these constraint classes and provide high-level specification and efficient evaluation support for on-live evaluation, integrated in the modeling environment.

A straightforward approach to improve the state-of-the-art of the DSML technology is to combine the metamodeling environment with integrated model transformations. The ViatraDSM framework [24] is built on this idea: with integrated transformations, based on the VIATRA2 engine, the language engineer has a powerful toolset to build dynamic modeling environments supporting the inter-domain transformations for tool-integration and event-driven transformations for model synchronization and constraint evaluation. In our paper [4], we have established the event-driven model transformation infrastructure on which ViatraDSM is built. In the current paper, I report my contributions in improving this engine and I also present how it can be used for model synchronization and constraint evaluation.

Additionally, to improve the integration of the VIATRA2 tool-set into the Eclipse infrastructure, I designed and implemented a bridge to import and process GMF editor definitions to enable the rapid development of domain-specific visual languages. In this report, I present this technology.

## 1.5 Objectives

In this paper, I present an effective incremental (event-driven) model transformation technology. I apply it to solve major domain-specific modeling challenges.

My detailed objectives for scientific contribution:

- I design an extended methodology for **event-driven model transformations**.
- I propose a generic method for the **incremental synchronization** of logical models and their graphical representations in a domain specific modeling environment.
- I propose a method for incremental, **on-the-fly checking** of complex well-formedness constraints during modeling.
- I design a **bridge** between the ViatraDSM and the Eclipse Graphical Modeling Framework.

In addition to conceptual contributions, my goal is also to implement:

- I **implement** the proposed changes on the incremental model transformation technology in the VIATRA2 model transformation framework.
- I **apply** the proposed domain specific modeling improvements in the ViatraDSM domain specific modeling framework to address the outlined issues with domain specific modeling.
- I **implement** the required functions for the ViatraDSM and the Eclipse GMF integration.

## 1.6 Structure of the Report

In Chapter 2, I introduce the technical and theoretical background of my work. The first section is dedicated to VIATRA2, the model transformation framework I use. The next Section introduces the domain specific modeling framework that Chapter 4 relies on: ViatraDSM.

In Chapter 3, an event-driven transformation methodology is elaborated, which relies on the incremental pattern matcher, and enables transformation rules to be triggered by conditions defined on the model. The new options and features of this engine are also described.

In Chapter 4, the event-driven transformation engine of Chapter 3 is put to use, to enhance the capabilities of the ViatraDSM framework with interactive constraint checking and incremental automatic abstract-concrete syntax synchronization.

In Chapter 5, the bridge between Eclipse GMF and ViatraDSM is elaborated. In this process, some newly created features (shown in Chapter 4) are applied as well.

Finally, Chapter 6 gives an overview of my work and accomplishments, and marks important directions of future improvements.

## Chapter 2

# Background technologies and concepts

### 2.1 Model transformation in VIATRA2

*VIATRA* (VI-sual Automated TRAn-sformations) is a model transformation framework developed at Department of Measurement and Information Systems. This section introduces the basic concepts and features of this system, including its modeling paradigm, input languages, semantics and architecture. In this Section, I conceptually follow [2].

#### 2.1.1 Metamodeling: Definition of Abstract Syntax

##### Visual and Precise Metamodeling

Currently, most widely used metamodeling languages (e.g. ECore) are derived (with slight variations) from the Meta Object Facility (MOF) [21] metamodeling standard issued by the OMG. However, as stated in [34], 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) [34] metamodeling approach was chosen in the VIATRA2 framework, which can support different metamodeling paradigms by supporting multi-level metamodeling with explicit and generalized *instanceOf* relations.

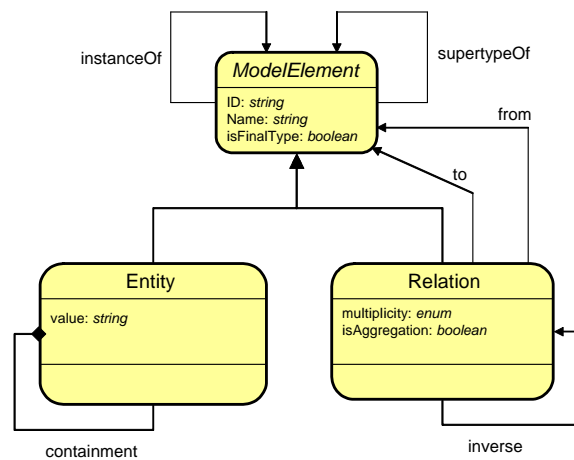


Figure 2.1: The VPM Metamodel

The VPM language consists of two basic elements: the entity (a generalization of MOF pack-

age, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). An *entity* represents a basic concept of a (modeling) domain, while a *relation* represents the relationships between other model elements<sup>1</sup>. 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 model space: 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 cannot 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:

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

Relations have *multiplicity* constraints, which impose 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.

### 2.1.2 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) [8] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [6] 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.

<sup>1</sup>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.

## 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) satisfies a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique presented in [33]. Basically, this means a subgraph isomorphism problem; each occurrence of the pattern is a mapping of pattern variables on model elements in such a way that they satisfy all the constraints that constitute the pattern.

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 [26].

VTCL also supports *generic patterns* and meta-transformations by providing matchable *supertypeOf* and *instanceOf* relationships.

**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 (Parent);
    UML.Class.parent (X, Child, Parent);
    UML.Class (Child);
} or
{
    UML.Class (Parent);
    UML.Class.parent (X, C, Parent);
    UML.Class (C);
    find ancestorOf (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. A variable is free 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, the matching will be done (in parallel) for all possible values of the given variable.

### Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [8], 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.

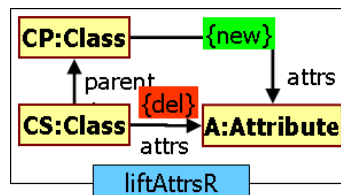


Figure 2.2: Sample graph transformation rule

The sample graph transformation rule in Figure 2.2 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 [16]) 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. 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 VIATRA2 framework 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 VIATRA2 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);
```

## 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 [6] 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 functions*, 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 C1 with find isClass(C1) do seq
  {
    print("Found a class : "+name(C1));
  }
  //Write to log
  log(info, "transformation done ");
}
rule printFormatted( in C) =
{
  //Print out the value
  print("Value is : "+C);
}

```

### 2.1.3 VIATRA2 Architectural overview

#### 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 model space 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.

As it can be seen on Fig. 2.3, the internal structure of the VIATRA2 framework can be split up into four major components:

1. VPM model space container, GTASM model store and VPM core interfaces
2. Pattern matchers
3. GTASM interpreter
4. Import/export facilities

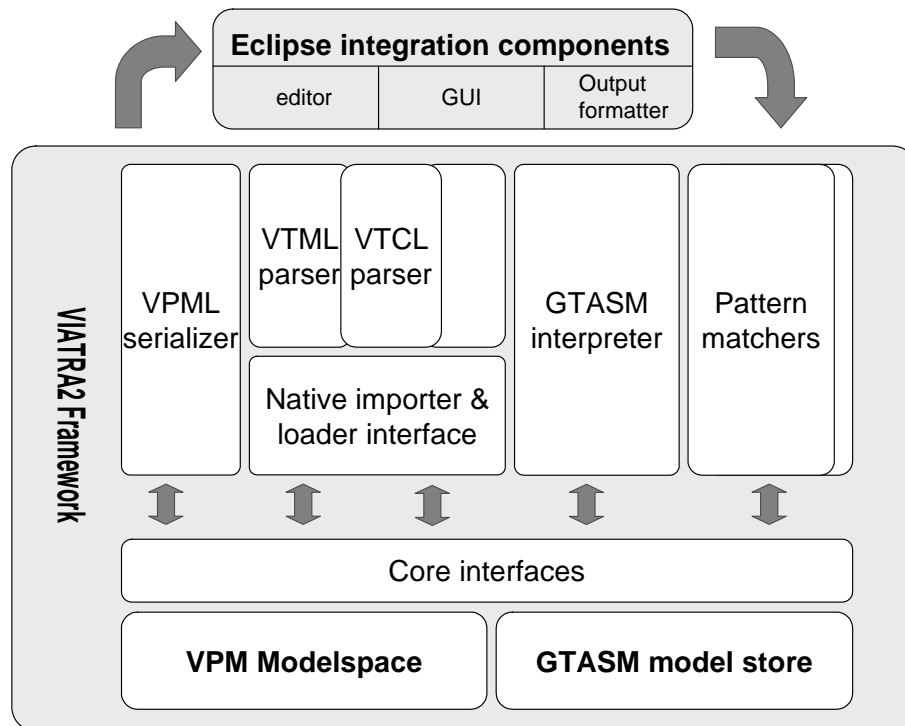


Figure 2.3: The architecture of the VIATRA2 framework

**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.

**Pattern matcher** Pattern matching is a key subject. The efficiency of the whole model transformation system highly depends on the efficiency of model storage and pattern matching. The VIATRA2 R3 framework can handle different pattern matchers.

**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 model space. The *VTML* parser is implemented as a native importer. The *loader* interface provides support for loading GTASM machines into the model store. The *VTCL* parser is implemented as a loader.

## 2.2 The ViatraDSM framework

In this section, 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 István Ráth; an early version was presented as a Scientific Students' Association report in 2005 [25], since then, it has matured greatly. During the introduction of this framework, I conceptually follow [24].

### 2.2.1 Domain-specific modeling in depth

With domain-specific visual editors, one can create high abstraction level models in a variety of application domains, ranging from enterprise applications to embedded systems. These models should ideally constitute an integral part of the development process, not only in documentation but in design-time analysis as well as runtime code generation. In order to support these requirements, a domain-specific language engineering tool should provide support for the following language engineering aspects:

- **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.

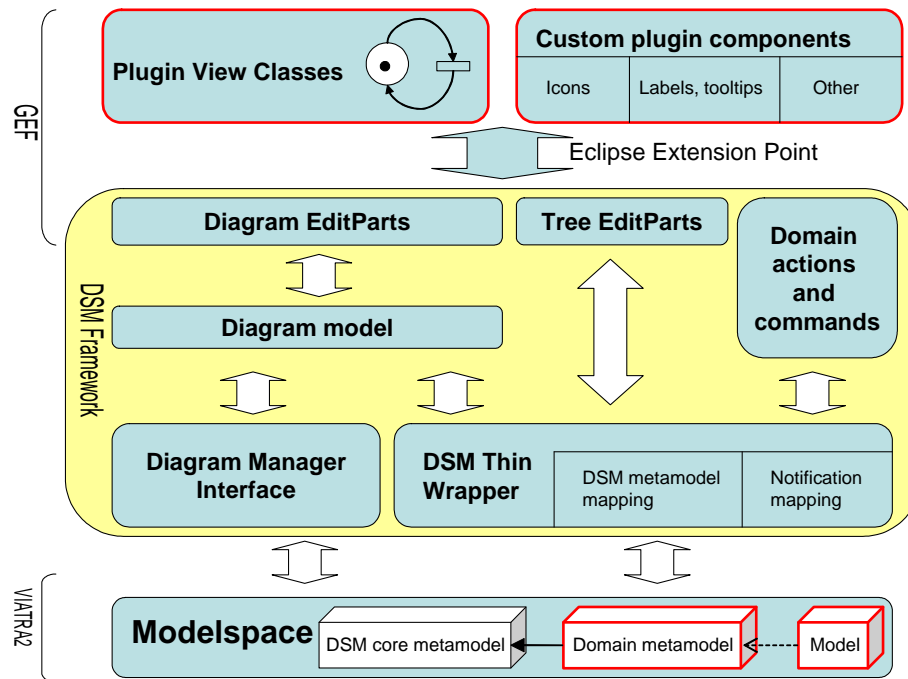


Figure 2.4: Detailed architecture of the DSM framework

While most state-of-the-art implementations, such as the Eclipse project’s Graphical Modeling Framework [31], or Microsoft’s DSL Tools suite [15], provide support for the first three aspects, the execution of dynamic semantics and model transformations in general is lacking in current tools. (As of now, usually academic tools provide support for model transformations, e.g. VMTS [13], ViatraDSM.)

ViatraDSM was built on the concept that by making use of the underlying VIATRA2-based model persistence transformation infrastructure, mathematically precise and high abstraction level support can be provided for all language engineering aspects. Most advanced features, such as the evaluation of complex constraints as well as the design-time execution of dynamic semantics can be efficiently formulated as model transformation problems, while VIATRA2 can also provide easy-to-use support for “traditional” model transformation tasks such as model-to-model mappings and code generation.

ViatraDSM follows the technical foundations of the VIATRA2 framework with choosing the popular and open source Eclipse [1] development platform as its environment. Although other Eclipse-based projects, such as GMF and openArchitectureWare [5], as well as numerous research projects such as TIGER [9] have implemented domain-specific modeling at various levels, ViatraDSM offers a different, model transformation-based perspective on this technology.

### 2.2.2 Architecture

Figure 2.4 shows the overall architecture of the ViatraDSM framework. In the next few sections we will elaborate many concepts of that figure, giving a short descriptions about components of the framework. The description of many internal components and implementation details are omitted, as they are unimportant in the context of this paper.

### Runtime framework

ViatraDSM 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. Another example of the runtime framework approach could be the Meta-Case MetaEdit+ tool. As an alternative to this principle, some other DSM frameworks (e.g. Eclipse Modeling Framework) perform *editor generation*.

On the architectural diagram of Figure 2.4) the 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.

### Domain specific graphical representation

In the ViatraDSM framework the choice about graphical model representation is reflected by the component *Plugin View Classes* in Figure 2.4. 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.

### 2.2.3 Modeling

**Metamodel structure** The core domain metamodel defines the set of elements that the language engineer may use to build up the metamodel of his own domain. Figure 2.5 shows the structure of the core domain metamodel; in the following paragraphs, the elements of this metamodel are described.

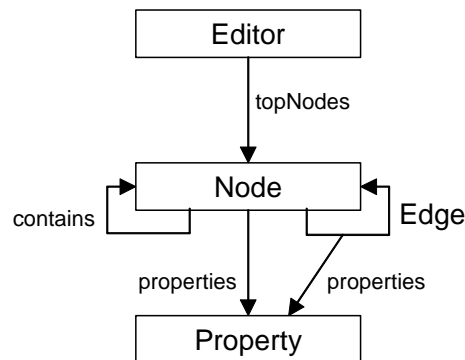


Figure 2.5: The core domain metamodel of the DSM framework

**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. In Figure 2.5, only one *Property* entity and two *properties* relations are present for simplicity, but in the actual implementation both have various subtypes.

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.

**Petri net example** During the demonstration of domain and diagram metamodels a simple but descriptive domain-specific visual language example will be used.

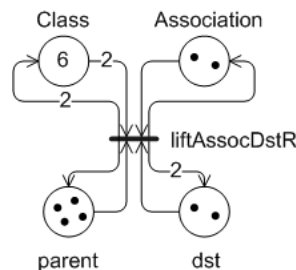


Figure 2.6: A sample Petri net with place capacities and arc weights

Petri nets (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 and add a token to all output places.

In Figure 2.7, the simplified domain metamodel for Petri nets can be seen. 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 Transition, whereas *InArc* is the opposite, and edge from a Transition to a Place. Both kind of edges may have a property entitled *weight*.

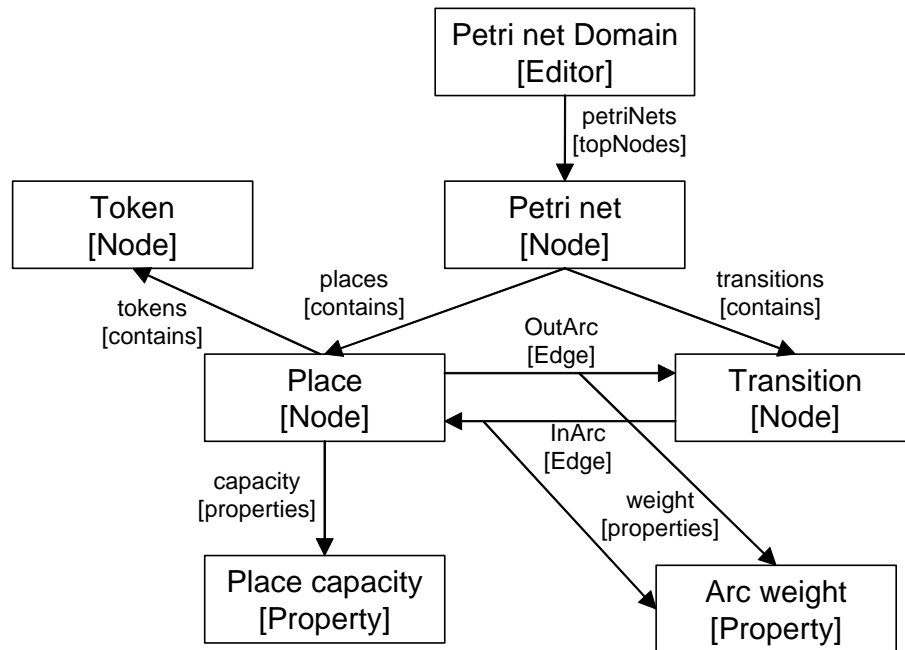


Figure 2.7: Example of a domain metamodel (Petri net domain)

**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.

## 2.2.4 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.

**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 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 Figure 2.8).

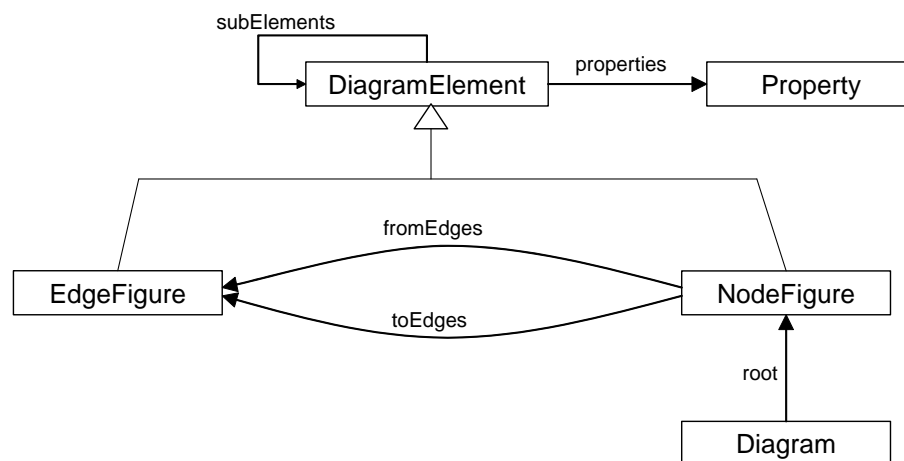


Figure 2.8: The core diagram metamodel of the DSM framework.

**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 be 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* through two relations (*fromEdges* and *toEdges*).

In Figure 2.9, an example diagram metamodel can be seen. (The names inside the square brackets give the core metamodel supertypes of each element.)

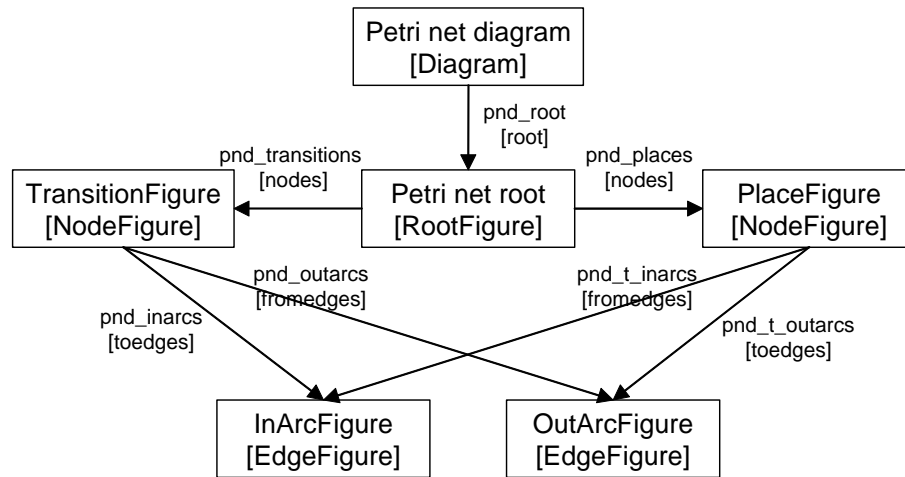


Figure 2.9: Example of a diagram metamodel (Petri net domain)

### 2.2.5 Required steps for a complete domain-specific editor

A language engineer would have to perform the following tasks to create a domain-specific editor:

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 and a basic UML-like syntax for diagrams.
4. Using the API of the DSM framework, and Draw2D primitives, create a customized concrete syntax representation.

A finished Petri net editor with custom graphical elements can be seen in Figure 2.10. Note that the language engineer only has to learn *one* expressive domain-specific language for *all* aspects of language engineering.

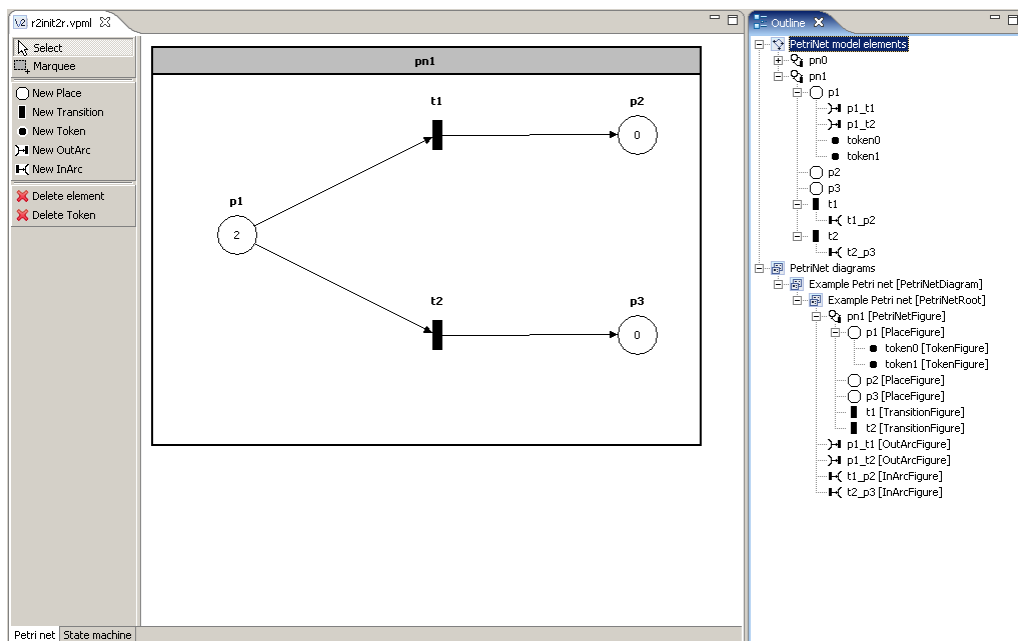


Figure 2.10: Petri net editor with custom graphical elements

## Chapter 3

# Event-driven model transformations with triggers

### 3.1 Goal of this Chapter

In this chapter, I present the new live model transformation engine, which was first introduced in [4]; since then it has improved in many ways. This is built on top of the incremental pattern matcher, which is briefly presented in Section 3.2. This live model transformation engine was conceptionally designed to work in a graph transformation environment; in this case it was implemented for the VIATRA2 framework.

In graphical modeling environments, a straightforward extension to simple editing operations is to execute a model transformation step based on the user's action (i.e. creation, deletion - as proposed in [11]). Such functionality is necessary to support various important features such as:

- automatic synchronization between an abstract and concrete syntax representation of domain-specific models.
- complex editing actions which feature a domain-specific complex operation (e.g. a visual tool for a refactoring operation in UML).
- in model simulators, the user's action may be mapped to the execution of a simulation step (e.g. firing a transition in a Petri net).
- incremental code generation, where a textual representation is kept in *live* accordance with a high-level model.

An important limitation of this approach is that the set of *events* is limited to actual editing operations accessible from the interface, which is not trivially extensible. Moreover, it is hard to provide high level, declarative support for the specification of these operations, since most implementations only provide a programming interface into which user action event handlers can be hooked.

The original implementation of the ViatraDSM framework, as described in [24], utilised this approach to map user actions to transformation sequences, which would be used to maintain a proper correspondence between abstract and concrete representations of domain models. As it was outlined in [25, 24], this approach could be extended to the execution of constraint-checking transformations as well as general model-to-code transformations.

My approach embraces this idea, however I have chosen a different strategy to detect changes in the model. The nature of this problem calls for an incremental approach for the pattern matching

strategy, because of the *locality principle*: such an atomic editing operation usually causes a small, incremental change in the model. Since underlying model transformations depend on information stored in the whole model, it is a straightforward idea to enable the transformation engine to “see” what has changed and only perform the necessary operations needed (e.g. for synchronization or the emission of a small textual output), instead of executing the whole model transformation program again.

Hence, the concept of *triggers* has been borrowed from relational database management systems. A *trigger* in general is a *condition* combined with a set of *commands*. Each time the condition is fulfilled, the underlying system executes the preset commands. In database systems, the condition for triggers is usually a modification of a relation (table) (e.g. INSERT, UPDATE, DELETE). Analogously, our triggers are fired by modifications of the VIATRA2 model space.

As it is shown in this chapter, this is a reasonable solution for the above problem types. Moreover, it constitutes the foundations of a new kind of model transformation execution methodology, where the control flow of transformations is not specified by a separate description (such as ASM structures in the case of VIATRA2), but it is encoded in the transformation rules themselves (much like the original as-long-as-possible execution semantics defined for graph transformation systems). This novel approach paves the way for a new kind of model transformation programming, where some aspects of a mapping may be formulated as event-driven transformations, and others using a more traditional, control flow oriented approach.

In Chapter 4, I demonstrate the usage of this technology in the context of domain-specific languages.

## 3.2 Incremental pattern matching

The incremental pattern matching engine was first introduced in [4], since then it became an integral part of the VIATRA2 framework. In this Section, I conceptually follow [12].

### 3.2.1 Pattern matching

Pattern matching plays a key role in the execution of VIATRA2 transformations. The goal is to find the occurrences of a graph pattern, which contains structural as well as type constraints on model elements. In the case of incremental pattern matching, the occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time – excluding the linear cost induced by the size of the result set itself –, making pattern matching a very efficient process. Generally speaking, besides memory consumption, the drawback is that these stored result sets have to be continuously maintained, imposing an overhead on update operations.

Our approach is based on the RETE algorithm [10], which is a well-known technique in the field of rule-based systems. This section is dedicated to giving a brief overview on how we adapted the concepts of RETE networks to implement the rich language features of the VIATRA2 graph transformation framework.

### 3.2.2 Tuples and Nodes

The main ideas behind the incremental pattern matcher are conceptually similar to relational algebra. Information is represented by a tuple consisting of model elements. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples that conform to the pattern. This set of tuples is in analogy with the relation concept of relational algebra.

The *input nodes* are a special class of nodes that serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing unary tuples representing the instances that conform to the type. Similarly, there is an input node for each relation type, containing ternary tuples with source and target in addition to the identifier of the edge instance. Miscellaneous input nodes represent containment, generic type information, and other relationship between model elements.

*Intermediate nodes* store partial matches of patterns, or in other terms, matches of partial patterns. Finally, *production nodes* represent the complete pattern itself. Production nodes also perform supplementary tasks such as filtering those elements of the tuples that do not correspond to symbolic parameters of the pattern (in analogy with the projection operation of relational algebra) in order to provide a more efficient storage of models.

### 3.2.3 Joining

The key intermediate component of a RETE is the join node, created as the child of two parent nodes, that each have an outgoing RETE edge leading to the join node.

The role of the join node can be best explained with the relational algebra analogy: it performs a natural join on the relations represented by its parent nodes.

Figure 3.1(a) shows a simple pattern matcher built for the *sourcePlace* pattern, which describes a Place-Transition pair connected by an out-arc, illustrating the use of join nodes. By joining three input nodes, this sample RETE net enforces two entity type constraints and an edge (connectivity) constraint, to find pairs of Place and Transitions instances which fulfill the constraints described in the pattern.

### 3.2.4 Updates after model changes

The primary goal of the RETE net is to provide incremental pattern matching. To achieve this, input nodes receive notifications about changes on the model, regardless whether the model was changed programmatically (i.e. by executing a transformation) or by user interface events.

Whenever a new entity or relation is created or deleted, the input node of the appropriate type will release an update token on each of its outgoing edges. To reflect type hierarchy, input nodes also notify the input nodes corresponding to the supertype(s). Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Each RETE node is prepared to receive updates on incoming edges, assess the new situation, determine whether and how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

Figure 3.1(b) shows how the network in Fig. 3.1(a) reacts on a newly inserted out-arc. The input node for the relation type representing the arc releases an update token. The join node receives this token, and uses an effective index structure to check whether matching tuples (in this case: places) from the other parent node exist. If they do then a new token is propagated on the outgoing edge for each of them, representing a new instance of the partial pattern “place with outgoing arc”. Figure 3.1(c) shows the update reaching the second update node, which matches the new tuple against those contained by the other parent (in this case: transitions). If matches are found, they are propagated further to the production node.

More details of this incremental pattern matching approach can be found in [3], where initial investigations concerning the run-time performance of our implementation also have been presented. Our results indicate a significant efficiency increase over the conventional (local search-based) pattern matcher; in certain applications, the difference is two orders of magnitude.

```

pattern sourcePlace(T, P) =
{
  transition(T);
  place(P);
  outArc(A, P, T);
}

```

Listing 3.1: VIATRA source code for the sourcePlace pattern

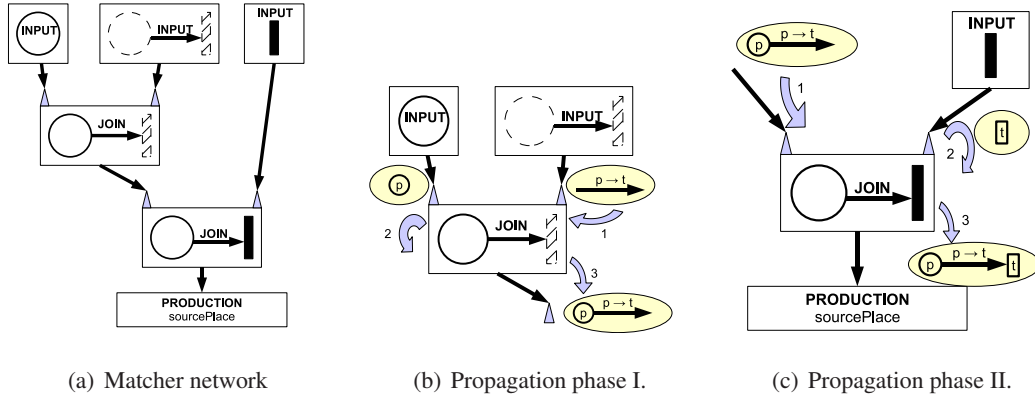


Figure 3.1: RETE matcher for the sourcePlace pattern

### 3.3 Event-driven model transformations with triggers

#### 3.3.1 The definition of the trigger concept

The condition of a trigger is a graph pattern. Thus an *event* in my case is not just a kind of user-action, it is either a new occurrence or a lost occurrence of a graph pattern.

Secondly, the trigger's predefined commands are a collection of graph manipulation rules, also referred to as the *action sequence*.

Finally, to ensure the proper functioning of the triggers I defined *control options*.

*Priority* (integer): If two or more triggers are ready to run at the same time, the trigger with the highest priority will run first. Of the triggers with the same priority one is selected non-deterministically.

*Mode* (once | always): The default setting is the *always mode*, where the trigger runs in the background until it is stopped by the user. The *once mode* is different, in this setting the trigger will only run once.

*Sensitivity* (rise | fall): If the trigger is set to *rise sensitivity* then the action sequence is executed with every new match of the trigger's condition pattern, on the other hand if it is set to *fall sensitivity* then the action sequence is executed with every loss of a previously existing pattern match. The default setting of a trigger is *rise sensitivity*.

*Startup* (active | passive): *Active startup* means that the action sequence of the trigger will be executed with all pattern substitutions that are present in the model space at the trigger's startup. Whereas with *passive startup* all current matches that are present in the model space when the trigger is added to the framework, are discarded. The default setting of a trigger is *active startup*.

*Execution* (forall | iterate): *Forall execution* means that the action sequence of the trigger will be executed with all pattern substitutions in a pseudo-parallel way. With *iterate execution* the pattern substitutions are executed in separate steps. Iterate mode is needed when a trigger can modify its own pattern matchings (e.g. generates new matches). However this is not the general case, so the default setting of a trigger is *forall execution*, since it is the faster method.

*AutoStart* (true | false): If the *autoStart* annotation is set to *true*, then the trigger will start automatically after loading.

In parallel with the execution control option, the trigger execution engine has an *execution mode* as well, which can be set to *serial mode* or *parallel mode*. In *serial mode* the action sequences of *different triggers* are executed separately, whereas in *parallel mode* these are executed in a pseudo-parallel way. The precise execution semantics are described in Section 3.4.

### 3.3.2 Adapting triggers into the VIATRA2 model transformation framework

I adapted the concept of triggers into the VIATRA2 model transformation framework. Thus, the *condition* of a trigger is a VIATRA2 graph pattern and the *action sequence* is a VIATRA2 ASM Rule. A trigger can be defined in a VTCL (Viatra Textual Command Language) code as a *graph transformation rule* consisting of a precondition pattern and an action part, and additionally specifying the *control options* with “Java5-like” annotations. A trigger is started with the *startTrigger*(“*gtrule name*”) function.

Note that currently I do not use the left hand side (LHS) - right hand side (RHS) GT representation to define triggers, instead a format which is more analog to the procedural programming languages is used (the FUJABA notation [16]). In this format, the trigger GT rule contains a precondition pattern (LHS), like the previous one, but instead of defining the postcondition (RHS) pattern the actions to be executed are defined. There is no limitation due to this style of coding, because any creation or deletion that can be described with a postcondition (RHS) pattern, can also be done in the action sequence. The current version of the trigger execution engine only supports this kind of trigger definition, but in possible future work it could be extended to handle the LHS-RHS format as well.

### 3.3.3 Trigger example

To better understand the nature of triggers, let’s consider the following example problem: vegetarians, whose boss is the father of a boy scout, commit suicide and leave a suicide note with the boy’s name, before they receive their paycheck. How do we formulate this behaviour as a trigger?

The action sequence is the easiest to form: it prints the boy’s name and removes the vegetarian subordinate from the model. When should this sequence commence? Whenever a boy and a subordinate is found, who satisfy the pattern formulated in Section 3.2.4. With the precondition and the action sequence, we have our transformation rule ready.

```
@Trigger(priority='350')
gtrule suicide( inout Subordinate, inout Boy) =
{
    //This pattern triggers the action sequence.
    precondition pattern boyScoutSonOfBoss(Subordinate, Boy) =
    {
        metamodel.boyscout(Boy);
        metamodel.person.child(R, Father, Boy);
        metamodel.male(Father);
    }
}
```

```

        find superior(Subordinate, Father);
       .metamodel.vegetarian(Subordinate);
    }

    //The action sequence.
    action
    {
        print(name(Boy) + " made me kill myself ");
        delete(Subordinate);
    }
}

```

The action part should execute whenever a situation with a boy scout and a vegetarian subordinate is found, and not when the situation is resolved; so the trigger should have a rise sensitivity. The specification states that all such subordinates commit a suicide, not just one: the trigger is in always mode. Since the rule applies also to those who are in these circumstances from the very beginning, and not just subordinates who fall into this situation at some time in the future, the trigger should have an active startup. Finally, let's assume that a trigger with priority 300 has already been defined to issue paychecks; under no circumstances should a paycheck be received by such a vegetarian, since the specification explicitly states that they die before getting paid; so the new trigger has to have a priority above 300. The trigger of the above transformation rule can be started the following way:

```
startTrigger( "suicide ");
```

A RETE network similar to that shown in Section 3.2.4 will be built; updates regarding to new occurrences of the pattern will be received by the trigger engine; the engine will execute the action sequence of the rule on these occasions.

## 3.4 Execution semantics

In this section, the conceptual background of the new event-based transformation engine written for the VIATRA2 model transformation framework will be described.

### 3.4.1 Events and the delta set

The *event* is the core concept of event-driven behaviour. In our context, events indicate a new occurrence or a lost occurrence of the precondition pattern of the trigger. Events consist of the tuple describing the pattern occurrence and a sign indicating whether it is a rising edge (positive update, new occurrence) or a falling edge (negative update, lost occurrence); furthermore, each event belongs to a trigger.

For each registered trigger, an event container called *delta monitor* receives events from the incremental pattern matching engine. It always contains the set of events that have occurred but have not yet been handled by the trigger executor. The *delta set* (denoted by  $\Delta$ ) consists of all events received on behalf of any trigger that have not yet been handled; in other words, it consists of the individual delta monitors.

The trigger engine may select and remove (concisely: pop) an event from  $\Delta$ ; the removal is the sign that the event is taken care of. There is one other way that an event can disappear from its delta monitor: *cancellation*, where an event with the same tuple but opposite sign is received; the result is that both events are eliminated, since the net unhandled change is zero.

The trigger engine removes an event when it handles it. Consequently, a subsequent update for that tuple with the opposite sign will generate a new event, that has to be handled separately. If the older event had been unhandled, it would have been cancelled out. This cancellation behaviour is necessary to eliminate logical network hazards, subsequent positive and negative updates for the same tuple within the same transaction, that amount to no change, but could induce wrong and/or unneeded trigger executions. Hazards can be caused by either subsequent model space changes belonging to the same transaction, or updates initiated by the same change but traveling different paths in the RETE network.

### 3.4.2 Operational semantics

#### Lifecycle of a trigger

Figure 3.2 shows the lifecycle of a trigger. Whenever a machine is loaded into the VIATRA2 framework (i.e. a VTCL is parsed), the trigger execution engine loads the triggers. The triggers with *autoStart='true'* control options are immediately started after the loading, the others can be started from the VTCL code (with the *startTrigger* function), or from the GUI as well. After starting, the corresponding delta monitor is constructed for every trigger, so the trigger becomes ready for execution. For passive triggers, pre-existing pattern occurrences are ignored and the delta monitor initializes empty; for active triggers, the delta monitor will contain all occurrences of the precondition pattern as cumulative rising edge events since the start of the framework. Figure 3.2 also shows how 'once' mode triggers will get unregistered and discarded after their first execution, while 'always' triggers can be activated again and again indefinitely. A trigger in this stage can be temporarily stopped from the GUI, or from VTCL code (with the *stopTrigger* function), during this suspended state its delta monitor is still running in the background.

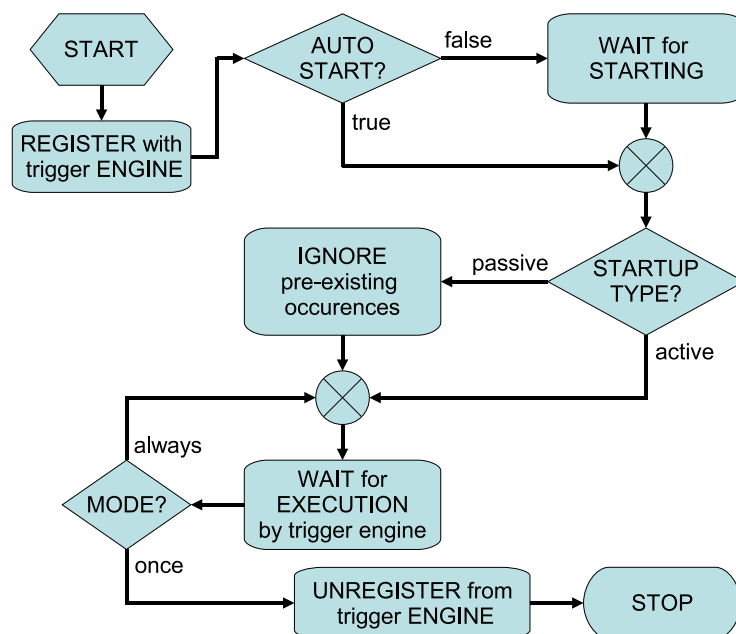


Figure 3.2: The lifecycle of a trigger

**Operational semantics of the trigger engine**

Figure 3.3 shows the operational semantics of the trigger engine, if the engine is set to serial mode. After a transaction is performed on the model space, the delta set is checked for new events (note: the suspended triggers' events are neglected). Triggers will be processed in the order of decreasing priority. After the highest priority trigger is popped (which has a new event in the delta set as well), its sensitivity is examined. If the trigger is sensitive for the sign of the event ('rise' triggers for positive updates, 'fall' triggers for negative updates), then the trigger is executed. At this point, the event is considered to be handled by the engine and the delta set is inspected again for events (some of which may have appeared as a consequence to the action part). When there are no more events in the delta set, the trigger engine suspends its operation until a new transaction is performed or a new trigger is registered, since these are the only occasions when the delta set is potentially filled with new events.

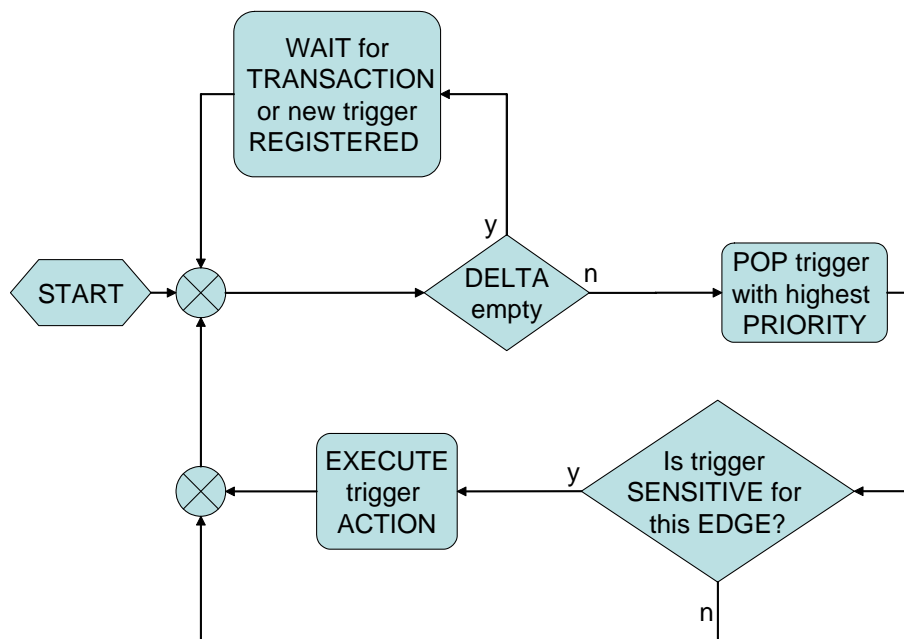


Figure 3.3: The operational semantics of the trigger engine (serial mode)

If the trigger engine is set to parallel mode, then it executes every (currently present) pattern matching in a pseudo-parallel way. The delta set is checked for new matches, only after this execution round. In Figure 3.4 the difference between the serial and parallel execution mode can be seen.

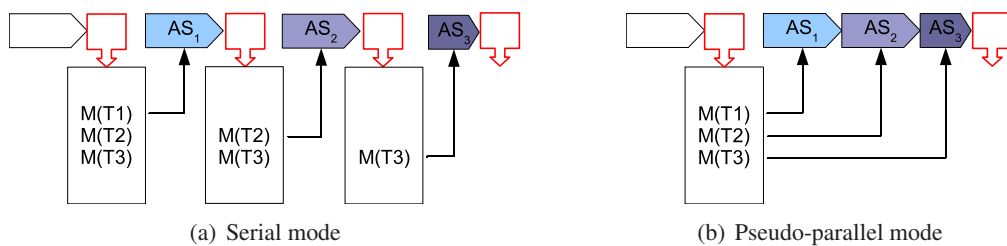


Figure 3.4: Execution modes for multiple trigger activation

### Execution of a trigger

The execution of a trigger depends on its *execution control option*. If it is set to *forall execution*, then the trigger engine will execute its action sequence with every possible matching in a pseudo-parallel way (see Figure 3.5(b)). If it is set to *iterate execution*, then the action sequence will be run with only one matching in this ‘iteration’ (see Figure 3.5(a)). The difference is important in two ways: firstly the *once mode* triggers are always discarded after the first iteration; so with forall mode they will run with every matching generated during the last transaction, but with iterate mode they will run with only one (non-deterministically selected) pattern matching. Secondly, although forall execution is faster, it can lead to certain problems, which are discussed in the forthcoming paragraphs.

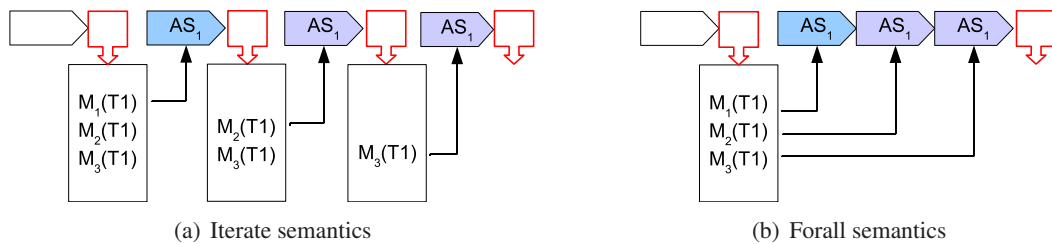


Figure 3.5: Execution semantics for multiple match set changes

### Notes regarding dynamic behavior

The action part of the trigger obviously influences the model space when executed. Consequently, it can generate events, both of higher and lower (or equal) priority; they will be handled in the same way as if they were the results of a user interaction. While this feature may be confusing at first, the trigger engineer can use it to simplify complex problems. Note, that if the action sequence makes a new match for its own trigger’s precondition pattern, then the action sequence will run with this pattern substitution as well. This means that recursive triggers can be created, which can be hazardous, it is the designer’s responsibility to keep this in mind.

Other possibilities include that an action part performs a change that cancels out an unhandled event. If the trigger engine is set to *serial execution* this is not a problem, since the other event has not been handled yet, it cannot have a higher priority than the trigger that was just executed. If the other event was of lower priority, the cancellation could be a desired effect; the trigger designer should pay attention to this, and specify the priorities in a way that suits his needs. However, the trigger engineer should be wary of the fact that cancelling out events of the same priority could lead to nondeterministic behaviour.

Moreover, if the trigger engine is set to *parallel mode*, there should not be running triggers in the system which perform a change that cancels out an unhandled event. This is due to the fact that when the trigger engine has started execution it stores all the current matchings, and will execute all of them; so if a match is cancelled in the model space before its execution, it will throw an exception.

The same applies to triggers with *forall execution control option* as well, but it is not that critical. A forall trigger can modify other triggers’ matches, although it should not cancel out its own matchings, as they are stored before execution (and executed in a pseudo-parallel way). However, based on my experience in trigger-writing, this is highly unlikely.

## 3.5 Handling synchronizational problems

In this section, our approach for handling model synchronizations in graph modeling environments are demonstrated.

### 3.5.1 Description of the problem

In a synchronizational problem, a source and a target model (or an entire source graph and target graph) are present, which we want to keep synchronised at all times. Note that usually this is not just a model transformation from a source model to a new target model, both models evolve concurrently. Every modification on the source model has to be followed on-the-fly with the relevant modification on the target model. Furthermore, the consistency of the models has to be kept, so every change in the target model which is ‘relevant’ to the source model, has to be handled as well. Usually these two models are fully separated from each other, this means that a model element in the source model does not have a ‘reference’ property to the relevant model element in the target model. Moreover this ‘reference’ may not even exist: multiple elements from the source model can be related to a single target element. So it can be seen that another model to record the relations between the source and target models is needed.

This third model is usually called the reference, or correspondence model. QVT [17] defines automatically generated reference models (or ‘trace’ models, as they call them), so the designer does not have to deal with this. In TGG [28] on the other side, these additional correspondence model elements and rules are included in a TGG rule definition.

### 3.5.2 Our approach

In Section 3.3 the description of the triggers was given. Thanks to the versatility of the VIATRA2 graph patterns and ASM rules, triggers alone can solve numerous problems. But as I discussed in 3.5.1, the use of a reference model is inevitable to handle synchronizational problems. As a result of the versatility of triggers, in a general problem the trigger designer can use any kind of reference (correspondence) model he wants.

On one hand this gives much more freedom to the designer, as a result many special problems can be handled in addition to general ones. On the other hand this requires the proper use of reference models by the designer. But, for the ViatraDSM framework (which I think is one of the most important domains where the trigger engine can be used) reference metamodels were designed, which can be applied in the most common situations. This helps the reusability and the development of triggers. These metamodels along with some detailed implementation of triggers can be found in Chapter 4. Here a very simple example is described, just to show how easily the triggers with reference models can be used for synchronization.

#### A simple example

In this example we have only three model elements: a *SourceElement*, a *TargetElement* and a *ReferenceElement*. What we want to achieve is that, if a *SourceElement* is created then a *TargetElement* has to be created too. The *ReferenceElement* itself is very simple, it has two relations (named *R-S* and *R-T*) which connect to the other two model elements. So after the creation of a *SourceElement* our model space looks like that shown in Figure 3.6.

**Creating a new model element** In Figure 3.7 we can see that a new *SourceElement* can be easily determined, because no *ReferenceElement* exists with the appropriate *R-S* relation in the

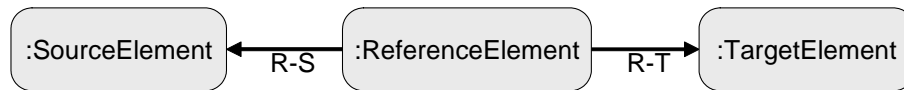


Figure 3.6: A consistent state of the model space

model space. A trigger handling this is easily written, because the VIATRA2 graph pattern supports negative pattern calls. In the action sequence of the trigger the *ReferenceElement* and the *TargetElement* and both relations belonging to the *ReferenceElement* need to be created.



Figure 3.7: SourceElement created

**Deleting a model element** As long as every *SourceElement* is created properly we should have a consistent model space, which means that every *SourceElement* has a *TargetElement* pair, and an appropriate *ReferenceElement* with its relations. As a consequence deleting can be handled easily. Taking a look at Figure 3.8 it can be seen, that after deleting a *SourceElement*, a *ReferenceElement* remains, which lacks the *R-S* relation. This can be easily defined with a negative graph pattern call, thus a trigger handling it can be written. The action sequence in this case will be to delete this *ReferenceElement* and the trigger's designer can decide whether to delete the *TargetElement* as well.

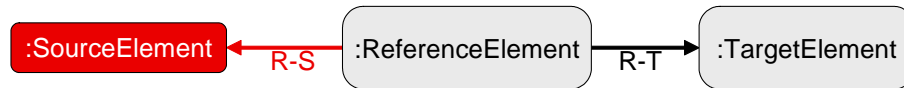


Figure 3.8: SourceElement deleted

## 3.6 Architecture of the implementation

Having settled the architectural concepts of the triggers in Section 3.4, we may now focus on the implementation of these concepts. To handle all aspects of the triggers, an orchestrator-like plugin, the Trigger Execution Engine was created (see Figure 3.9).

Every VIATRA2 VPML model space has a single Trigger Execution Engine. This engine stores the triggers, and provides the user with feedback about the ongoing triggers in the system. Furthermore, the Trigger Execution Engine registers all of the triggers' precondition graph patterns in our incremental pattern matcher. A special receiver object is implemented to capture the new or the lost matches of the triggers' precondition graph pattern (further details are given in Section 3.6.1). Another implementation task was to define when the Trigger Execution Engine should check the receivers for new matches. The practical results concerning this problem are discussed in Section 3.6.2.

### 3.6.1 Delta Monitor

The *delta monitor* is an object responsible for monitoring unhandled events belonging to the trigger. It contains events; rising edge events are the results of positive updates received from the

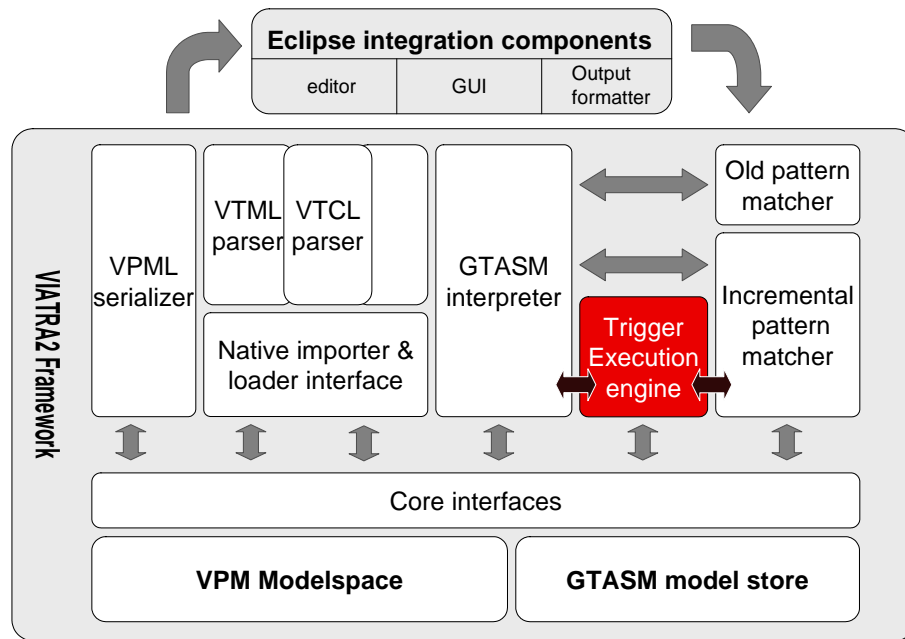


Figure 3.9: VIATRA2 R3 Framework extended with the Trigger Execution Engine

incremental pattern matcher (see Section 3.2); falling edge events reflect negative updates.

Upon initialisation, the delta monitor accesses the *production node* in the RETE network responsible for the precondition pattern of the trigger, and attaches itself to it as a child node. Depending on the startup property of the trigger, the delta monitor is either initialised empty (‘passive’), or with receiving the entire contents of the production node as positive updates (‘active’). Note that this is the only part of the Trigger Execution Engine that actually depends on the RETE-based incremental pattern matcher.

The operational semantics of the delta node is the following: when a RETE update token is received, the delta monitor checks whether it already contains an event with the same tuple as the token (and, consequently, with an opposite sign). If so, then the newly received token cancels out the previously received event, and neither of them is stored. If not, then a new event is created from the updated tuple and with the sign of the update. In accordance with the semantics of events and the *delta set*, when the engine handles an event, it is removed from the delta monitor.

### 3.6.2 Transaction handling

The VIATRA2 framework has its own *transaction handling* mechanism, every modification to the model space is done by transactions. Above all there is an inner notification system which notifies the listeners about various operations proceeding in the system. This enables us to ‘wake up’ the Trigger Execution Engine after the completion of each transaction, to check the *delta set* for new matches. Thus there is no need for regular, ‘timed’ polling of the delta set, because nothing can be changed without a transaction. Another advantage is, that the delta set are never checked during a model transformation, when improper matchings could be found.

When a transaction is started in the system, the execution engine clears the delta sets and when it is completed, it processes the results as discussed in Section 3.4.

### 3.6.3 VIATRA2 user interface extension

The VIATRA2 framework has an Eclipse view, the *VIATRA2 Model spaces* view. This view provides an interface to the import/export/parser facilities, and the loaded GTASM program models (machines) can also be seen. I extended this view, so that the imported triggers can be viewed, started, stopped and deleted. The Eclipse native properties view has also been extended to show the control options of the selected triggers. The extended view can be seen in Figure 3.10.

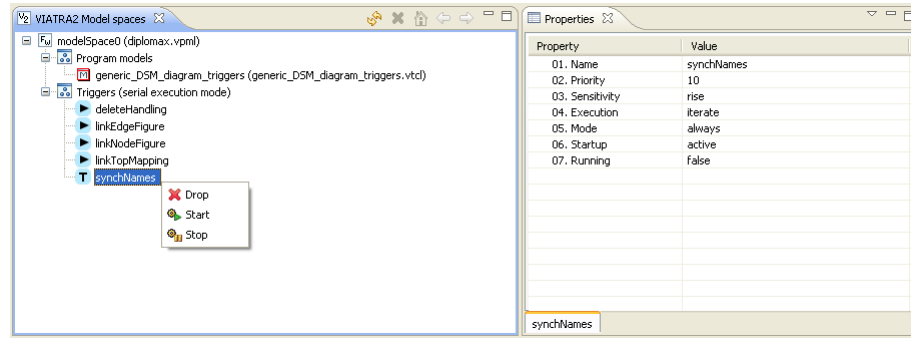


Figure 3.10: The trigger execution engine’s user interface

Upon parsing a VTCL code into the VIATRA2 framework, the triggers are imported automatically into the trigger execution engine as well. Previously imported triggers, which have the the same name as one of the new ones, are discarded from the engine. Triggers can be started and stopped from the VTCL code as well, with the *startTrigger(“grule name”)* and *stopTrigger(“grule name”)* functions.

## 3.7 Summary

In this Chapter my new event-driven transformation engine was presented. Although it can be used for any general incremental synchronization problem, the main reason motivating its design was to solve some specific problems in the ViatraDSM framework. The detailed description of these problems along with how I used triggers to solve them are shown in Chapter 4.

In this chapter, I presented the live model transformation engine, which was first introduced in [4]. I designed and implemented this trigger execution engine; the incremental pattern matcher (shown in Section 3.2) which it is based on was developed by Gábor Bergmann.

I would like to summarize the work I have done on the trigger execution engine while working on master’s thesis. The *control options* were extended with the *execution* and *autoStart* options. The execution engine itself was extended with an *execution mode* option. I extended the engine to handle the import of the triggers automatically, and to handle the annotations for setting the control options. The operation semantics shown in Section 3.4.2 were conceptually extended and refined with these new execution options/modes. All in all, the import- /execution handling parts of the source code were reimplemented. The user interface was extended with the start/stop possibility and a new property-view extension was created.

## Chapter 4

# Live model transformations in domain-specific modeling

### 4.1 Goal of this Chapter

In this Chapter, the use of the event-driven model transformation engine will be presented. This engine, along with the definition of the triggers, was discussed in Chapter 3. The ViatraDSM framework, which is a tool supporting the construction of dynamic and visual modeling languages based on the transformation facilities of the VIATRA2 framework, was presented in Chapter 2.2.

The extension of the VIATRA2 framework with these new live transformations enables the handling of some aspects of domain specific modeling (e.g. logical and graphical representation synchronization) to be more efficient than before, inside the ViatraDSM framework. Furthermore, it opens the possibility to deal with some problems, which we were unable to handle without this (e.g. evaluation of complex language-specific constraints). Some of these aspects are common problems, which usually cannot be solved in the commercial domain specific modeling tools without writing program code.

In Section 4.2 a detailed presentation about the current challenges in the domain-specific modeling world is presented. In Section 4.3 an automatic abstract-syntax synchronization is presented, based on generic triggers. In Section 4.4 the incremental validation of a complex dynamic modeling constraint is demonstrated.

### 4.2 Challenges in the domain-specific modeling world

As it has been established in 2.2.1, a DSM tool typically provides support for the following language engineering aspects:

- **Concrete syntax**, specifying the visual appearance of domain models;
- **Abstract syntax**, defining their low-level representation;
- **Well-formedness rules**, describing constraints which must be satisfied.

#### 4.2.1 Separation of abstract and concrete syntax representations

In most implementations, the concrete and abstract syntax representations are structurally bound, either meaning that

- (i) they are practically identical (as in the case of MetaCase MetaEdit+ [14]), or
- (ii) the concrete syntax is a subgraph of the abstract syntax (i.e. a partition of the abstract model space is visualised – some elements may be omitted).

In other words, a node in the abstract syntax is necessarily a node in the concrete syntax and an edges need to correspond accordingly. This is acceptable for basic languages, but as the complexity of a domain metamodel approaches practical applications, this becomes a serious limitation. After all, DSMs should be designed to provide a simple and easy-to-use interface so that non-technical *domain experts* can be involved in the development process. This has been recognized in recent developments such as Graphical Modeling Framework [31]: GMF introduced the notion of the “mapping metamodel” and specifying a generative bridge between conceptually separated diagram (concrete syntax) and logical model (abstract syntax) layers. However, this only provides limited abstraction facilities, for instance, a common technique, such as the representation of *derived values* (e.g. the number of tokens assigned to a petri net place) from abstract syntax models cannot be mapped to concrete syntax elements (such as the label in the petri net place’s graphical notation).

In [4] we outlined how this idea can be extended using the live model transformation technology to give the toolsmith maximum flexibility in completely separating abstract and concrete syntax representations (and keeping it synchronised all times). However, writing unique triggers for every problem domain is very inefficient, so in this thesis I take a further step, and in Section 4.3 propose a generic method for the incremental synchronization of logical models and their graphical representations along with the possibility of maintaining unique triggers for the complex abstraction facilities. This new method applies the concept of the “mapping metamodel” from the Eclipse GMF.

#### 4.2.2 Evaluation of complex constraints

In typical DSM tools, constraints are limited to simple static checks expressing conformance to the domain metamodel. For instance, such a constraint may state that a Petri net place may only contain Tokens, and an OutArc may only start at a Place and only end at a Transition instance. Such constraints are easy to be enforced in an editor, since based on the metamodel, a syntax-driven editor (such as described first in TIGER [9] and implemented in basically every environment) can be derived either by code generation (GMF, Microsoft DSL Tools), or runtime orchestration (MetaEdit+, ViatraDSM). Such an editor only allows operations which preserve the syntactic correctness of the model.

In practical applications, constrains are typically more complex than this. For example, even a simple Petri net editor can contain support for capacity-constrained places, which may not hold more tokens than a given pre-determined value. In a user interface design applications, constraints can be even more complex, involving conditional expressions and more complicated structures. The Object Constraint Language [20] is intended to provide high-level support for the specifications of such constraints; DSM tools only recently started to implement some support for the evaluation of more complex constrains (GMF’s support is based on the EMF-OCL project [23]). Such a facility can support the validation against complex constraints using two approaches:

1. an *off-line* execution schema checks constraints on user request, producing some cumulative output which can be reviewed later (which constraint is violated where);
2. an *on-line* approach checks the validity of constraints in the context of a given editing action, potentially preventing actions which would result in a violation of a constraint (note:

typically, violations should be allowed since a constraint’s validity may be restored by a later operation).

As outlined in [24], both cases can be formulated as model transformation problems, involving the execution of checking transformations. In section 4.4, I demonstrate the feasibility of this approach using *incremental* transformations in Section 4.4, which allow for a high-performance implementation and also yield an intuitive way of specification.

### 4.3 Generic abstract-concrete syntax synchronization

The importance of arbitrary abstract-to-concrete syntax mapping in visual languages is best supported by the simplicity argument: abstract syntax representations tend to be too complicated and difficult to interpret 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 way, while concrete syntax is usually more compact.

In this Section I present a generic method for the automatic handling of the abstract-concrete syntax snychronisation. The language engineer does not have to write any sort of program or model code in most of the cases, only the domain metamodels (described in Section 2.2) have to be extended with a new one.

#### 4.3.1 Architectural changes in ViatraDSM

In [4] we introduced a new core diagram metamodel (see Figure 4.1(a)) for the ViatraDSM framework. Instead of the previous modelbindings, every DiagramElement has a MappingElement. The core domain metamodel (see Figure 4.1(b)) of the ViatraDSM framework introduced in 2.2 was not changed.

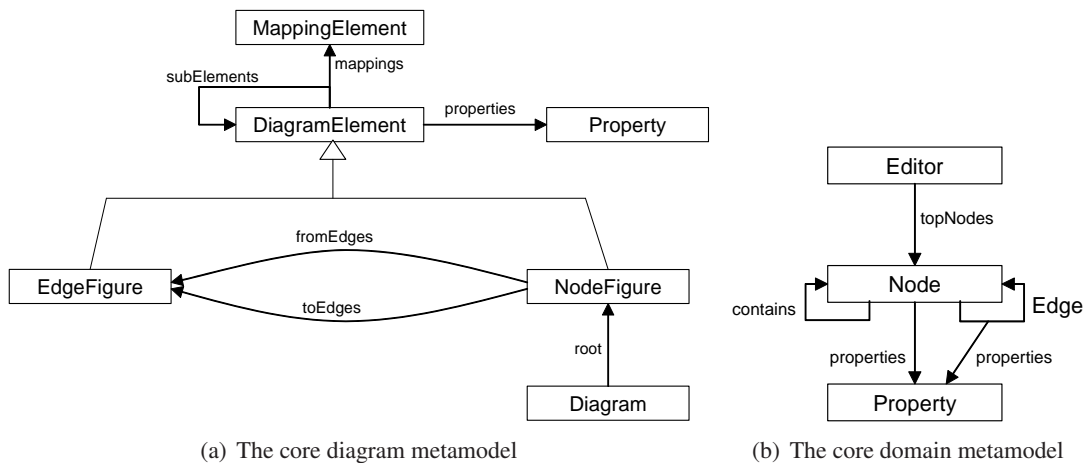


Figure 4.1: VIATRA metamodels

The core mapping metamodel is shown in Figure 4.2(a). A MappingElement can be an EdgeMappingElement with an edgeMapping relation, or a NodeMappingElement with a nodeMapping relation. In general the mapping model is actually a reference model between the graphical and the logical representations of the model. This metamodel has been extended with a *TopMapping*, which is the “reference metamodel element” between the root elements of the logical and graphical metamodels of the domain; it has a diagram and an editor relation.

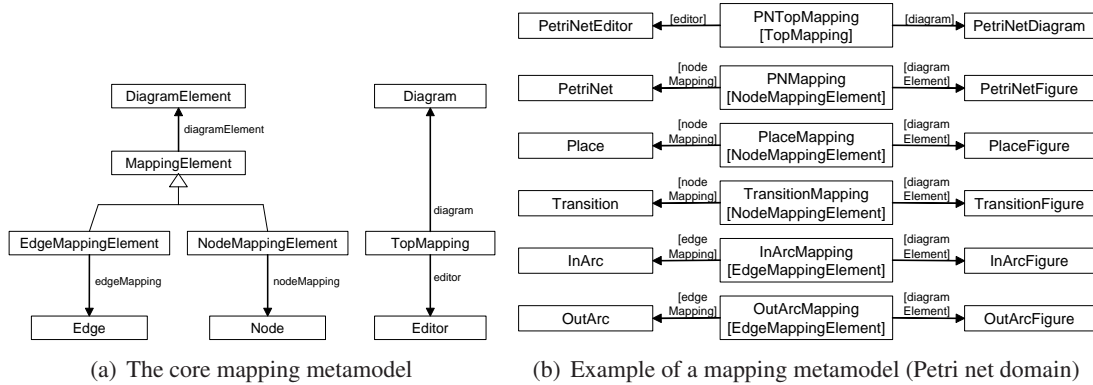


Figure 4.2: The mapping metamodel with an example

In [4] we have shown that this mapping metamodel can be used as a reference model between the abstract and concrete syntax, and based on that, the incremental synchronization of the two separate models can be done using triggers. However as it turned out there are many cases, when a mapping between a logical and the graphical element is simple; writing unique triggers in these cases is time consuming.

So I decided to adapt the Eclipse GMF’s mapping metamodel to the ViatraDSM. This means that along with the domain and diagram metamodels a *mapping metamodel* should also be created for every domain-specific language in the ViatraDSM. This metamodel is not the instance of the core mapping metamodel, rather its subtype. Every possible one-to-one mapping should be formulated in this metamodel. An example mapping metamodel for the Petri net domain is shown in Figure 4.2(b) (note: the domain and the diagram metamodel of the Petri net can be found on page 25 and 27). Some abstraction (like the *Token* element in the Petri net, which is a *Node* in the domain metamodel, and an attribute property in the diagram metamodel) cannot be formulated with this new mapping metamodel, so these have to be handled with unique triggers.

### 4.3.2 Generic triggers

In this Section, I demonstrate the generic triggers which can handle the synchronization of abstract and concrete syntax for every domain-specific language, where the mapping metamodel (described in Section 4.3.1) is present along with the domain and diagram metamodels. Only one-to-one mappings can be handled by this trigger, although they can be easily changed or extended to handle more complex abstractions.

To present the triggers a “compact” notation (e.g. see Figure 4.3) is used in this section, because the original VTCL code (see Appendix A) is a bit hard to understand, if someone is not familiar with the VIATRA2 modeling language. In these Figures the abstract-syntax model elements are coloured yellow, the concrete-syntax elements are orange. The elements of the metamodels, are represented with dotted lines.

In the Figure, the precondition pattern and the action sequence are shown. The keyword NEG marks a subpattern that is embedded into the current one to represent a negative condition for the original pattern. The NEW and DEL keywords mark the respective operations of the trigger’s action sequence.

### Creation handling

In Figure 4.3 the *linkNodeFigure* trigger is presented. This trigger creates *Nodes* for every *NodeFigure* which is created by the ViatraDSM during model editing. Two more triggers has been written for creation handling: one for synchronizing the *Diagram-Editor* elements, and one for synchronizing the *EdgeFigure-Edge* elements. The basic operational schema behind the three triggers is the same, so I will only demonstrate the NodeFigure-Node synchronization in depth.

The precondition pattern of the *linkNodeFigure* trigger is actually an *OR* pattern, it handles two cases (which require the same action sequence) at once. The first is where a NodeFigure is created under the Diagram element, and the second is where a NodeFigure is created inside another NodeFigure.

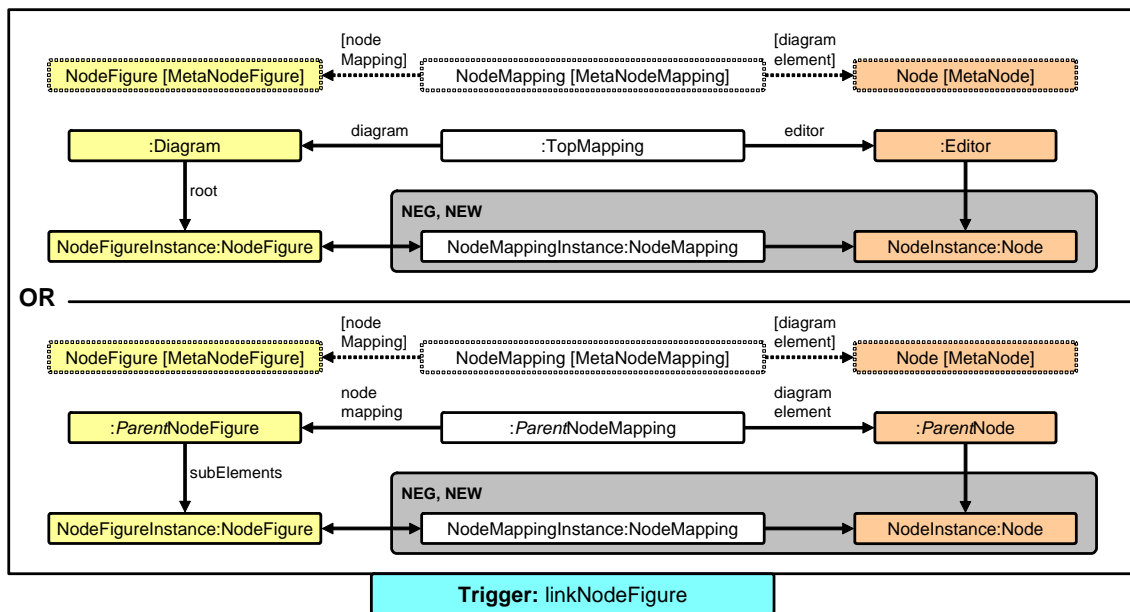


Figure 4.3: Node creation

Both patterns have the same structure. The most important aspect is that the pattern has a negative condition (marked with NEG keyword in the figure), so it is matched to NodeFigure instances, which do not have mapping elements. This part of the pattern is actually marked with NEW keyword as well, so in the action sequence, this part will be created. This is where the new mapping metamodel is needed. The top of the pattern describes that the selected NodeMapping metamodel element has a relation to the actual NodeFigure type. This mapping metamodel element, with the two relations “contains” all the needed information: it has a relation to the Node type, which will be instantiated in the action sequence, and “mapped” to the NodeFigure instance. To determine the “place” of the Node in the model space, the “parent” element of the NodeFigure is also matched; it could be a *Diagram* or a *NodeFigure*, with its mapping-, and the respective *Editor* or *Node* element.

### Deletion handling

In Figure 4.4 a part of the *deleteHandling* trigger (which handles the deletion of the *Nodes* and *NodeFigures*) is presented.

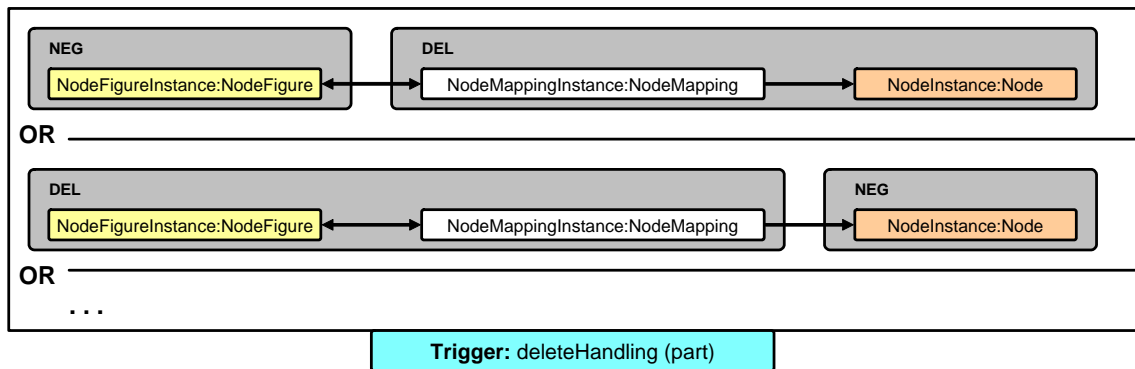


Figure 4.4: Node, NodeFigure deletion

The first pattern can be matched to a Node with its related MappingElement, which has no related NodeFigure. This occurs when a graphical element is deleted from the diagram. The MappingElement has to be deleted. In my realization the related Node element is deleted as well, but that is optional.

The second pattern is the opposite of the previous one. It can be matched to a NodeFigure, with its related MappingElement, which has no related Node. This occurs when an abstract syntax element is deleted, thus its related Figure should be deleted from the graphical representation too. Naturally, the MappingElement used also has to be deleted.

Note, that these *deleteHandling* triggers only use ViatraDSM coremetamodel elements, so they are domain metamodel independent (i.e. they can be used even with ViatraDSM domain languages, which do not have mapping metamodel).

### 4.3.3 Conclusion

In this example I proposed a way to use triggers and mapping metamodels to achieve automatic incremental abstract-concrete syntax synchronization for one-to-one mappings. Due to the versatility of triggers, it is possible to maintain unique triggers for the complex abstraction facilities along with the generic ones.

The most important aspect of this method is that, thanks to the versatility of triggers, the domain-specific language designers do not have to write any sort of Java source code to handle even arbitrary abstract-concrete syntax synchronization. Above all this, thanks to the efficiency of the incremental pattern matching, all these synchronizations operate on-the-fly.

## 4.4 Incremental constraint checking

As I described in Section 4.2, the on-the-fly evaluation of complex language specific constraints (e.g. design pattern conformity checking) is crucially important, because the sooner we find an error, the less it will cost to repair. Previously, the ViatraDSM framework had no mechanism for evaluating complex constraints, so this new efficient solution is a completely new addition to the framework.

In this section, I give another motivating example for live transformations with the incremental validation of a complex dynamic modeling constraint. In this Section, I conceptually follow our conference paper [12].

### 4.4.1 Architectural changes in ViatraDSM

Generally, the user is editing models using a domain-specific editor which is capable of enforcing static type constraints so that only syntactically correct Petri net graphs can be produced. However, an advanced framework may go beyond this and provide immediate feedback if more dynamic constraints, such as a *capacity constraint* (in Petri nets) is violated (e.g. the user tries to assign too many tokens to a place). Figure 4.5(a) shows a simplified metamodel for Petri nets, which will be used throughout this Section.

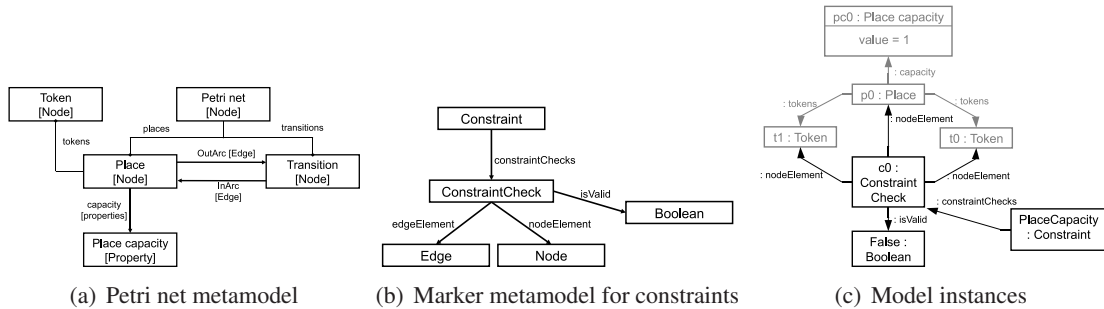


Figure 4.5: VIATRA metamodels and model instances

In order to provide support for the editor, the modeling environment makes use of a *marker metamodel* which is a special type of trace model depicted in Figure 4.5(b). A *Constraint* denotes a particular run-time constraint being enforced within the editor, e.g. “PlaceCapacity”. For each constraint, we explicitly mark all the (Petri net) elements, which are required to evaluate the constraint within a given context by a *ConstraintCheck* element. Each evaluation context of a *Constraint* is explicitly marked by a *ConstraintCheck* instance (i.e. separately for each Petri net place and its respective tokens in our case).

The *isValid* relation indicates whether the constraint is valid *currently* for the context defined by the *ConstraintCheck* instance; the runtime environment makes use of this relationship to indicate graphical feedback to the user. In Figure 4.5(c), place *p0* contains two tokens but has a capacity of 1, thus, the associated *ConstraintCheck* instance indicates that the *PlaceCapacity* constraint is violated in this context. In our demonstrating example used throughout the paper, we aim at providing an incremental evaluation of the capacity constraint in all contexts in response to elementary changes or complex transactions initiated by the user or another transformation.

#### 4.4.2 Triggers

In Figure 4.6, an “initializing” trigger is shown. It is automatically fired after the user creates a new Place and the modeling environment creates (as a complex model change involving multiple elements) an additional Capacity and a ConstraintCheck marker element for the new Place-Place.Capacity pair. As a common technique in graph transformation based approaches, we use a negative application condition to indicate that the action sequence should only be fired for new pairs without a marker element.

```
@Trigger(priority='10', mode='always', sensitivity='rise')
gtrule initPlace() = {
  precondition pattern pre(P) = {
    Place(P);
    Place.Place_Capacity(PC);
    Place.capacity(Cap,P,PC);
    neg pattern placeSet(P) = {
      Constraint.ConstraintCheck(CC);
      Constraint.ConstraintCheck.nodeElement(NE,CC,P);
    }
  }
  action {
    new(Constraint.ConstraintCheck(CC));
    new(Constraint.ConstraintCheck.nodeElement(NE, CC, P));
  }
}
```

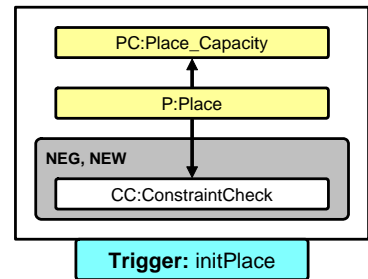


Figure 4.6: Place instance initialisation

As described in Chapter 3, the system tracks changes in the match sets of patterns and executes the action sequences in a persistently maintained *execution context*. This context consists *persistent variables* (called *ASM functions* in VIATRA2; essentially global associative arrays) along with the matched pattern variables.

```
// An array to cache token numbers
asmfunction numberOfTokens / 1;

@Trigger(priority='10', mode='always', sensitivity='rise')
gtrule placeAdded() = {
  precondition pattern pre(CC) = {
    Constraint.ConstraintCheck(CC);
    Place(P);
    Constraint.ConstraintCheck.nodeElement(NE_P,CC,P);
  }
  action {
    // Initialize the 'numberOfTokens' array
    update numberOfTokens(P) = 0;
    // calculate the initial number of tokens
    forall T with find placeToken(P,T) do
      update numberOfTokens(P) = numberOfTokens(P)+1;
    // check the constraint's validity
    call constraintCheck(P,CC);
  }
}
```

Listing 4.1: Invoking constraint checking in the transformation context

In Listing 4.1, the *numberOfTokens* array is used to cache the amount of tokens assigned to a given place (the array is indexed by the Place reference). This trigger is fired after the ConstraintCheck marker element has been created by the trigger described in Figure 4.6, and performs

the necessary steps to set up the cache with the appropriate value (Listing 4.2; note that some pattern definitions have been omitted for space considerations).

```
rule constraintCheck( in P, in CC) = seq {
  // match the PlaceCapacity element storing the value of P's capacity.
  choose PC with find placeCapacity(P,PC) do seq {
    if (numberOfTokens(P) <= value(PC)) seq {
      // delete a possible previous 'False' marking
      choose R find constraintFalse(CC,R) do delete(R);
      // create a new 'True' marking
      new ConstraintCheck.isValid(R,CC, Boolean.True);
    }
    else seq {
      choose R with find constraintTrue(CC,R) do delete(R);
      new ConstraintCheck.isValid(R,CC, Boolean.False);
    }
  }
}
```

Listing 4.2: Command sequence to check the validity of the capacity constraint

## Creation

In practical applications, a chain of triggers may be used to execute multiple incremental updates. For instance, after a Token instance has been added by the user, the system may execute a trigger similar to Figure 4.6 to connect the new Token to the CapacityConstraint marker element. In reaction to that, after `initPlace` has reached the commit point, the `tokenAdded()` trigger (Listing 4.3) is activated.

```
@Trigger(sensitivity='rise')
gtrule tokenAdded() = {
  precondition find connectedToken(P,CC,T) = {
    find placeToken(P,T);
    Constraint.ConstraintCheck(CC);
    Token(T);
    Constraint.ConstraintCheck.nodeElement(NE_Tok,CC,T);
  }
  action {
    update numberOfTokens(P) = numberOfTokens(P) + 1;
    call checkConstraint(P,CC);
  }
}
```

Listing 4.3: Trigger to handle the addition of Tokens

The `tokenAdded()` trigger updates the `numberOfTokens` array stored in the execution context, and initiates a constraint update which provides feedback to the user.

## Deletions

To detect deletions, a trigger for the same precondition pattern as used in Listing 4.3 can be used in *fall* mode. In this case, the `undef` constant is assigned to the corresponding pattern variables to indicate that the model element identified by the pattern variable is no longer existent (Listing 4.4). However, other pattern variables (pointing to existing model elements) can be used in the action part in the usual way.

```
@Trigger(sensitivity='fall')
gtrule tokenRemoved() = {
  precondition find tokenAdded.connectedToken(P,CC,T)
  action {
    // only act if token T has been lost (deleted)
  }
}
```

```

if (T == undef) seq {
  update numberOfTokens (P) = numberOfTokens (P) - 1;
  call checkConstraint (P,CC);
}}

```

Listing 4.4: Handling token deletion

### Attribute updates

The system also provides support for the incremental detection of attribute changes. VIATRA2 provides a *value* field for all node types; in this example, this value field of the *PlaceCapacity* property node is used to store the actual value of the capacity of the connected *Place*.

```

// associative array to cache place capacity values
asmfunction capacities / 1;

@Trigger(sensitivity='fall')
gtrule capacityChanged() = {
  precondition pattern pre (P,PC) = {
    find placeCapacity (P,PC);
    // check condition to define a value constraint
    check (value (PC) == capacities (PC))
  }
  action {
    // check whether the attribute update caused the activation
    if (PC!= undef && P!= undef && value (PC) != capacities (PC)) seq {
      // update constraint validity
      choose CC with find placeConstraint (P,CC) do call checkConstraint (P,CC);
      // store new value
      update capacities (PC) = value (PC);
    }
  }
}}

```

Listing 4.5: Handling attribute updates

In Listing 4.5, a *fall* trigger is defined for changes in the capacity value (the user may change that any time during modeling). The trigger is activated for changes in the match set of a complex pattern involving a *check condition*, which is a special feature of the VIATRA2 transformation language to define additional attribute constraints which cannot be expressed using structural graph patterns. The global array *capacities* is used to cache known capacity values; the trigger checks whether the cause of activation was a change in the attribute value and proceeds to update the constraint validity.

### 4.4.3 Conclusion

Apart from this simple constraint, some basic ‘trigger design patterns’ should be formulated here. First, the triggers which create the ConstraintChecks and its relations usually should have negative pattern calls in their precondition, to ensure that Node elements (e.g. Place, Token,...) are only connected to one ConstrainCheck. These ‘builder’ triggers usually should not use ASMFunctions as the information they store are discarded when closing the model space.

As this example showed, the objective of the on-the-fly incremental constraint evaluation is achieved and the constraint designer does not have to write any sort of Java source code to create such a constraint. However, to speed-up constraint creation, a good future improvement of the ViatraDSM could be the automatic generation of triggers for simple constraints like this one.

## 4.5 Summary

In Section 4.2, a detailed presentation about the current challenges in the domain-specific modeling world is presented. In the following two sections, I presented two detailed examples. Alongside the examples, I proposed important conceptual results.

In the first example, in Section 4.3, I proposed a new diagram mapping mechanism, along with generic triggers for the incremental synchronization of logical models and their graphical representations. Maintaining unique triggers for the complex abstraction facilities is not detained by this new technique.

In Section 4.4 I proposed a new Constraint metamodel, which can be used with the event-driven transformation engine for evaluating complex language specific constraints on-the-fly. The currently available DSM tools usually have limited constraint checking abilities. The on-the-fly incremental checking of complex constraint is not present in any of them.

## Chapter 5

# Graphical Modeling Framework integration

### 5.1 Goal of this Chapter

The goal of this chapter is to present a method for better integrating the VIATRA2 toolset with Eclipse. This question will be addressed in the domain-specific modeling perspective: to integrate the Eclipse Graphical Modeling Framework (GMF) with ViatraDSM. The tool integration is a very important factor in helping the reusability of various solutions, thus speeding up, and reducing the cost of the development process.

The goal is to present a method for importing and processing the GMF's domain-specific languages, in a way, that their functionality and behaviour remain exactly the same as in ViatraDSM. The import of the actual graphical-look is not part of this thesis, as the GMF uses a theoretically different approach for the graphical visualization than ViatraDSM. Overall, this method should be as complete as possible, extensible and flexible.

The techniques described and demonstrated in Chapter 3 and 4 proved that ViatraDSM, with its rich model transformation based features, can be in some ways a conceptually better alternative than the GMF for model-driven software development. As the GMF is a widely used technique in the industry, the ViatraDSM can benefit from this integration in many ways.

In Section 5.2 a brief introduction to the Graphical Modeling Framework is presented. In Section 5.3 an example is presented which will be used throughout this Chapter. In Section 5.4 the overall workflow of the integration is described. In Section 5.5 the method of importing of the GMF definition models is described, and in Section 5.6 the processing of these models is discussed.

### 5.2 Presentation of the GMF technology

In this Section I present an overview of the Graphical Modeling Framework (GMF [31]), based on [7].

The use of both Eclipse Modeling Framework (EMF [29]) and Graphical Editing Framework (GEF [30]) for building Eclipse-based functionality is quite common. Although developing a domain-specific language editor with them requires a great amount of coding, which is an easy, but time consuming task for anyone who is familiar with these technologies. This is one reason, which inspired the GMF project; it uses EMF and GEF in a generative manner to create domain-

specific editors without writing source code.

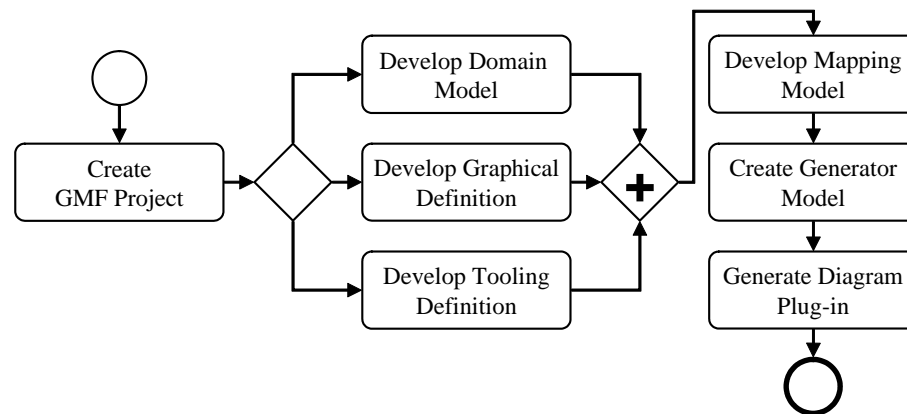


Figure 5.1: GMF overview

Figure 5.1 illustrates the main components and models used during GMF-based development. In the following, the steps for creating a domain-specific visual language in GMF is shown:

1.
  - **Domain model:** Although it may seem necessary to begin with the domain model (the abstract syntax), it is not the case with GMF, as the diagram definition is maintained separate from the domain. However, it is worthwhile to start with this, as the graphical- and tooling definition can be derived from it using a built-in wizard. This metamodel should define the language concepts and relationships.
  - **Graphical definition:** A graphical definition model is used to define the figures, nodes, links, etc. that will be displayed on the diagram, which is a GEF-based runtime. It has no direct connection to the domain models, for which they will provide representation and editing. With the help of the built-in wizards the graphical definition can be easily derived from the domain model.
  - **Tooling definition:** This is an optional component, which is used to design the palette and other periphery (menus, toolbars, etc.). This definition can be derived from the domain model as well.
2. **Mapping model:** This is a key model in GMF development and will be used as input to a transformation step which will produce our final model, the *generation model*. It is expected that a graphical or tooling definition may work equally well for several domains. For example, the UML class diagram has many counterparts, all of which are strikingly similar in their basic appearance and structure. A goal of GMF is to allow the graphical definition to be reused for several domains. This is achieved by using a separate mapping model to link the graphical and tooling definitions to the selected domain model(s).
3. **Generator model:** Once the appropriate mappings are defined, GMF provides a generator model to allow implementation details to be defined for the generation phase. From this model the final diagram code, or even a standalone Rich Client Platform (RCP) application can be generated.
4. **Diagram plug-in:** The production of an editor plug-in based on the generator model will target a final model; that is, the diagram runtime (or “notation”) model. The runtime will bridge the notation and domain model(s) when a user is working with a diagram, and also

provides for the persistence and synchronization of both. An important aspect of this runtime is that it provides a services-based approach to EMF and GEF functionality and is able to be leveraged by non-generated applications.

### 5.3 Demonstrating example

In this Section I present an example, which will be used throughout this Chapter for demonstrating purposes.

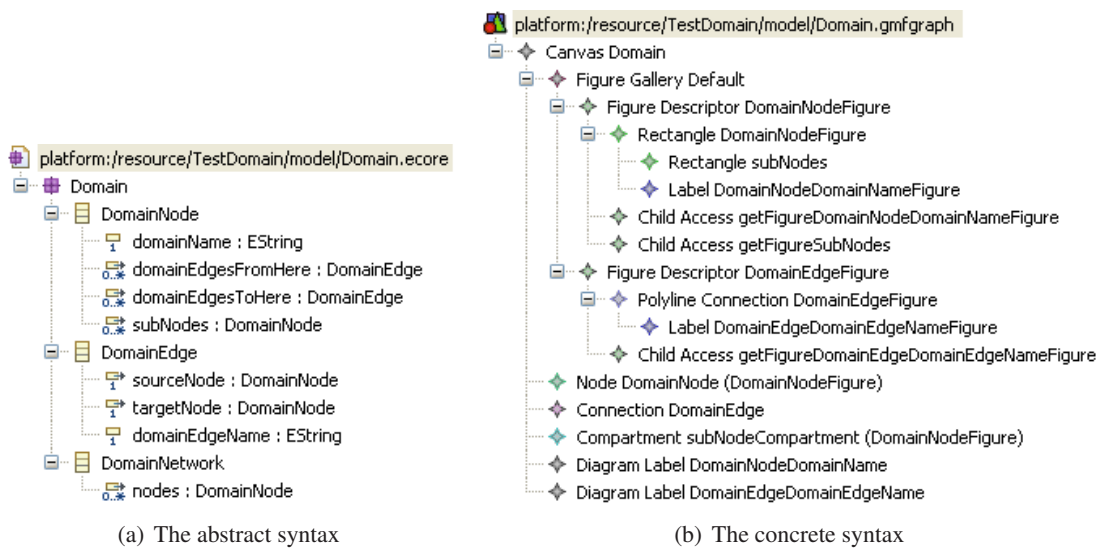


Figure 5.2: Demonstrating example

In Figure 5.2(a) the language metamodel (abstract syntax) can be seen. It is a very basic domain-specific language, it has three elements only. The root (container) element is the *DomainNetwork* which can store multiple *DomainNodes*. The *DomainNodes* can be embedded into each other, and can be connected with *DomainEdges*. A *DomainEdge*'s *sourceNode* and *targetNode* can even be the same *DomainNode*.

In Figure 5.2(b) the graphical definition (concrete syntax) is shown. It can be seen, that it mostly contains the actual design of the elements, although it also contains the graphical representations of the abstract syntax elements (Node *DomainNode*, Connection *DomainEdge*, etc.). This definition file was created with the built-in wizard from the domain model.

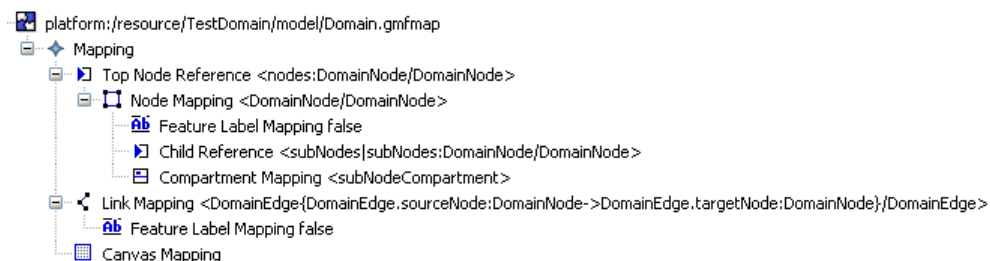


Figure 5.3: The GMF mapping model of the example

The tooling definition and the mapping model (which can be seen in Figure 5.3) was created with the built-in wizard as well. From these models the generator model was derived, and the

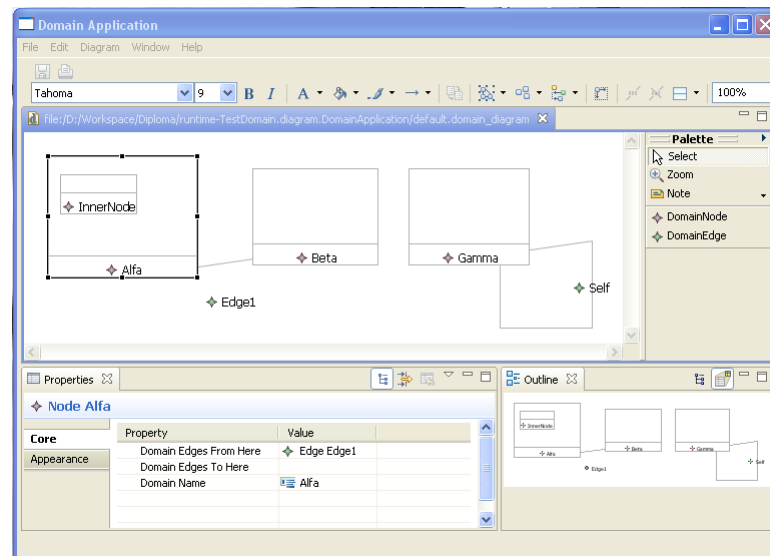


Figure 5.4: The generated GMF diagram editor

actual standalone editor was generated, which can be seen on Figure 5.4.

## 5.4 Proposed usage workflow

My main goal is to create a ViatraDSM domain-specific editor which is functionally equivalent to the GMF generated editor.

At this point it is necessary to point out some major conceptual differences between the GMF and the ViatraDSM. GMF uses an *editor generation* (as described in Section 5.2) approach. The initial four definition models (domain model, graphical definition, tooling definition, mapping model) are only processed when the generator model is created, from which the actual editor code is generated. In contrast the ViatraDSM is a *runtime framework*, which means it has a general (domain independent) editor, which firmly relies on the metamodels (domain and diagram), and uses them to validate model editing actions. So trivially, the GMF's generator model and the generated code is irrelevant for us, only the initial four definition models will be needed to create a ViatraDSM domain-specific editor.

The other main conceptual difference is the visualization. In the GMF it is generated from the graphical description as well, but ViatraDSM does not contain any design information in the diagram metamodel. In ViatraDSM there is a default implementation for the views and the graphical editor, which will be used with the imported GMF visual languages. (However a custom graphics editor can be written in Java for any ViatraDSM domain-specific language. These custom figures have to be implemented as Draw2D classes.)

The workflow of the usage is the following:

1. Import the four fundamental definition models to the VIATRA2 model space.
2. Process the metamodels: create ViatraDSM domain-, diagram- and mapping metamodels.
3. Implement custom graphics for the editor.

The details about importing the metamodels is described in 5.5, their processing of them is shown in 5.6.

## 5.5 Importing the GMF definition models

The first step of the workflow is importing the GMF definition models: the *domain model* (\*.ecore), the *graphical definition* (\*.gmfgraph), the *tooling definition* (\*.gmftool) and the *mapping model* (\*.gmfmap) files into the VIATRA2 model space. Throughout this process the newest release of the GMF (2.0), along with the newest EMF (2.3) was used.

For this step the VIATRA2's EMF Support plugin can be used, which was developed by Dániel Tóth. With this tool the three basic models (domain model, graphical definition and tooling definition) can be imported into the model space. However, the mapping model cannot be imported. To overcome the limitations of this generic VIATRA2 EMF importer, I implemented an importer plugin for the mapping models.

The VIATRA2's native importer interface was used during the implementation of this importer plug-in. With this interface arbitrary entities and relations can be created in the existing model space from Java program modules.

At this point, everything needed for the ViatraDSM domain-specific language generation is now present in the model space. In the next Section the processing of these models is described.

## 5.6 Processing the GMF definition models

In this Section, the second step of the workflow is presented, the processing of the GMF definition files. The aim is to create a valid ViatraDSM domain-specific language, which can be divided into the creation of three metamodels; a *domain metamodel* (see page 44), a *diagram metamodel* (see page 44) and a *mapping metamodel* (see page 45). Throughout this Section I concentrate on the conceptual differences between the GMF definition files and the ViatraDSM metamodels.

### 5.6.1 Processing the *domain model*

The VIATRA2 EMF Support has a built-in extra support for ViatraDSM, so the domain models imported from ECore files can be compactized, which means, that they are tagged with ViatraDSM metamodel types. These imported and compactized GMF domain models are automatically valid ViatraDSM domain metamodels, so no further changes have to be done to them. The ViatraDSM domain metamodel can be seen in Figure 5.5 (the original GMF domain model can be seen in Figure 5.2(a)).

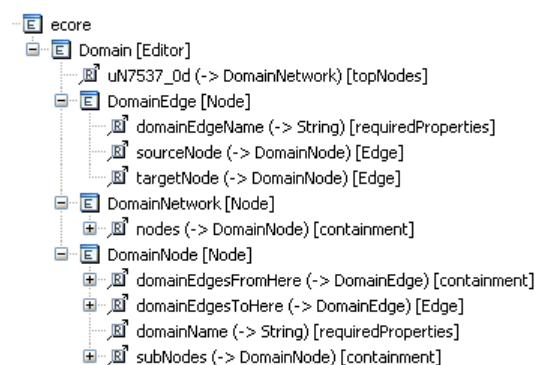


Figure 5.5: The ViatraDSM domain metamodel

### 5.6.2 Processing the *tooling definition*

The tooling definition is not currently used during the creation of the corresponding ViatraDSM domain-specific language.

### 5.6.3 Processing the *graphical definition*

I created a conventional graph transformation module to transform the graphical definition (\*.gmf-graph) into a valid ViatraDSM diagram metamodel. The graphical definition is the key component in GMF, it contains information on the actual graphical properties/elements, which will appear in the GEF-based runtime generated from it. On the contrary, the ViatraDSM diagram metamodel has no information regarding the design of the objects, so most of the information in the graphical definition model is redundant in our point of view. However it also contains the information about the structure of the diagrams, which is needed to create the ViatraDSM diagram metamodel. These elements, with the appropriate transformations are described here: (see Figure 5.6 as well)

- *Canvas*: is the root element of a diagram in the GMF. In the ViatraDSM there is no such dedicated root element, a *NodeFigure* is created instead.
- *Node*: has the same role as the *NodeFigure* in ViatraDSM. For each Node a *NodeFigure* is created as a subElement under the ‘top’ *NodeFigure*.
- *Connection*: has the same role as the *EdgeFigure* in ViatraDSM. For each Connection an *EdgeFigure* is created. To set the *fromEdges* and *toEdges* relations pointing towards the *EdgeFigure* some parts of the relevant GMF mapping model are also used as well.
- *Compartment*: describes the Nodes which are embedded into one-another. The subElement relation in ViatraDSM describes the same abstraction.

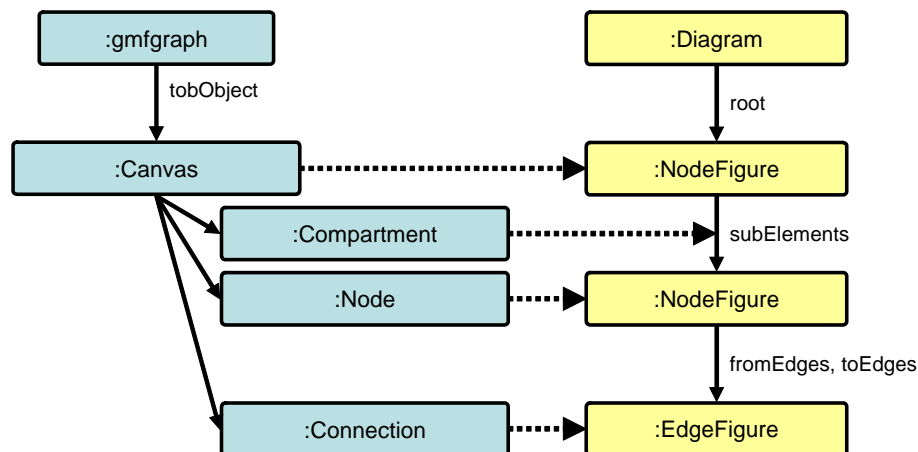


Figure 5.6: Processing the graphical definition

Four GT rules were written to transform these structures into a valid ViatraDSM diagram metamodel. With this metamodel the construction of diagrams can already be started in the ViatraDSM framework. Here I would like to present the `convertCanvas` GT rule:

```
gtrule convertCanvas ( out GMFCanvas, in GMFGraph, inout DiagramMetaPos ) =
{
  precondition pattern lhs (GMFCanvas, GMFGraph, Name) =
```

```

{
  emf.compactdomains.gmfgraph(GMFGraph);
  entity(GMFCanvas) below GMFGraph;
  emf.ecoremetamodel.resource.'ResourceContainer'.topObject(RA, GMFGraph, GMFCanvas);
  datatypes.String(Name);
  emf.compactdomains.gmfgraph.'Identity'.name(RB, GMFCanvas, Name);
}
action
{
  let NF= undef, RA= undef in
  seq {
    // Create NodeFigure metamodel element
    new (entity(NF) in DiagramMetaPos );
    new (supertypeOf('DSM'.coremetamodel.'Diagram'.NodeFigure, NF));
    rename(NF, value(Name));

    // Create a relation to this metamodel element
    new (relation(RA, DiagramMetaPos, NF) );
    new (supertypeOf('DSM'.coremetamodel.'Diagram'.root, RA));
    rename(RA, "gmfgraph_root ");
    update DiagramMetaPos = NF;
  }
}
}

```

The created complete ViatraDSM diagram metamodel can be seen in Figure 5.7 (this was generated from the GMF diagram definition shown in Figure 5.2(b)).

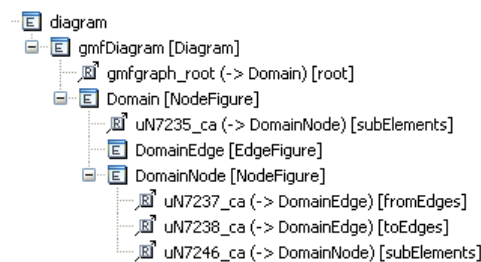


Figure 5.7: The ViatraDSM diagram metamodel

#### 5.6.4 Processing the *mapping model*

From the mapping model (\*.gmfmap) a ViatraDSM mapping metamodel should also be transformed. The GMF example's mapping model was shown in Figure 5.3.

Each graphical element presented in the last Section also has a relevant Mapping element. Each of these elements has a relation to the mapped domain and graphical elements. My mapping metamodel follows the same structure, so with four GT rules, every NodeMapping and EdgeMapping element can be created. The elements which have to be transformed are:

- *CanvasMapping*: is transformed into a NodeMapping, because the Canvas object has been converted to a NodeFigure.
- *NodeMapping*: is (obviously) transformed into a ViatraDSM's NodeMapping.
- *LinkMapping*: is transformed into an EdgeMapping.

- *CompartmentMapping*: is the only element which can be omitted, as the ViatraDSM handles the subNodes differently.

Here I would like to present the `convertCanvasMapping` GT rule:

```

gtrule convertCanvasMapping( out CanvasMapping, in MappingMeta, in DiagramMeta) =
{
  precondition pattern lhs(CanvasMapping, Canvas, Node)=
  {
    emf.compactdomains.gmfmap.CanvasMapping(CanvasMapping);
    emf.compactdomains.gmfgraph.Canvas(Canvas);
    emf.compactdomains.gmfmap.CanvasMapping.diagramCanvas(RA, CanvasMapping, Canvas);
    entity (Node);
    emf.compactdomains.gmfmap.CanvasMapping.domainMetaElement(RB, CanvasMapping, Node);
  }
  action
  {
    let NodeMappingMeta= undef, Rel_NodeMapping_NodeFigure= undef,
      Rel_NodeMapping_Node= undef in
    seq {
      // Create NodeMapping metamodel element
      new (entity(NodeMappingMeta) in MappingMeta);
      new (supertypeOf('DSM'.coremetamodel.'Diagram'.
        'DiagramMapping'. 'NodeMappingElement', NodeMappingMeta));

      // Create a relation from the NodeMapping object to the NodeFigure
      try choose NodeFigure with
        find findDiagramElement(NodeFigure, DiagramMeta, Canvas) do seq {
          println("NodeFigure selected for CanvasMapping : " + fqn(NodeFigure));
          new (relation(Rel_NodeMapping_NodeFigure, NodeMappingMeta, NodeFigure));
          new (supertypeOf('DSM'.coremetamodel.'Diagram'. 'DiagramMapping'.
            'MappingElement'. diagramElement, Rel_NodeMapping_NodeFigure));
        }

      // Create a relation from the NodeMapping object to the Node
      new (relation(Rel_NodeMapping_Node, NodeMappingMeta, Node));
      new (supertypeOf('DSM'.coremetamodel.'Diagram'. 'DiagramMapping'.
        'NodeMappingElement'. nodeMapping, Rel_NodeMapping_Node));
    }
  }
}

```

With this mapping metamodel my generic diagram triggers (presented in Section 4.3) can be put to use. They will handle every deletion and creation, and will keep the domain elements in the background updated incrementally, as the diagram model changes. The created complete ViatraDSM mapping metamodel can be seen in Figure 5.8.

## 5.7 Summary

After the import and process of our example GMF domain-specific language (which was described in Section 5.3) a fully functional ViatraDSM domain-specific language was created. A screenshot of this editor can be seen in Figure 5.9.

The process and its implementation described in this chapter can be used to automatically derive ViatraDSM editors based on GMF's domain description models. The trigger engine, running the transformations driven by the mapping metamodel is capable of providing a transformation-based solution to the abstract-concrete syntax mapping technology hard-coded into GMF.

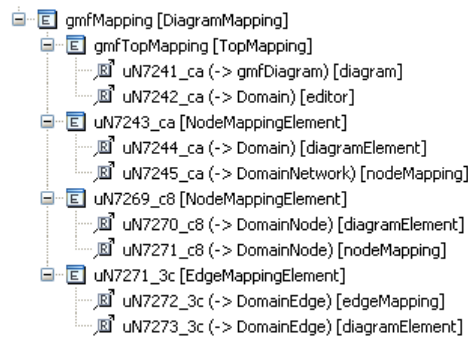


Figure 5.8: The ViatraDSM mapping metamodel

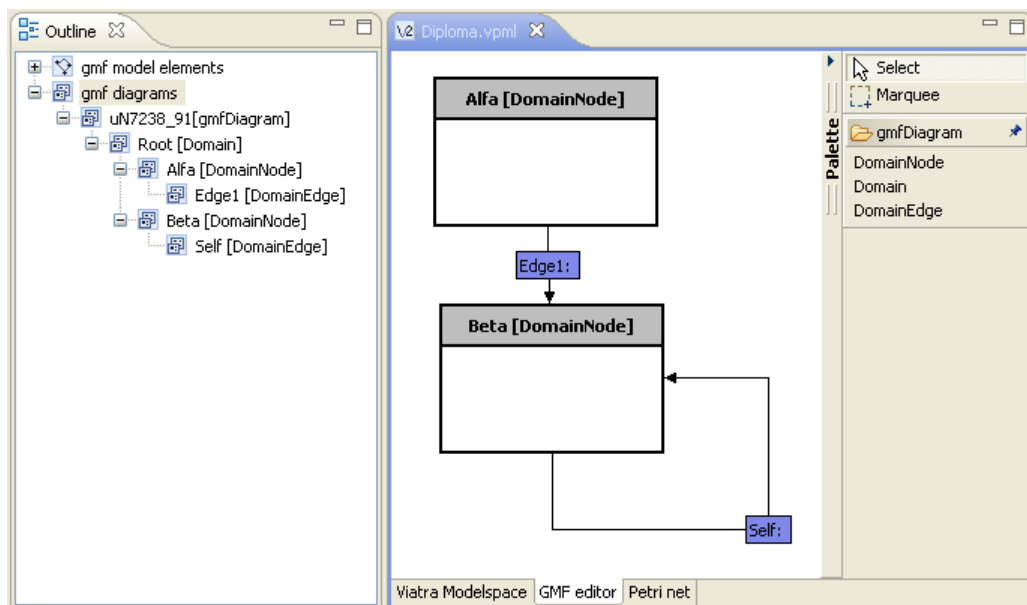


Figure 5.9: The equivalent ViatraDSM editor

However, as an important conceptual addition, it has to be noted that this is only an automatically provided foundation onto which an arbitrary mapping can be built. By using additional triggers constructed with a similar design pattern, one might easily augment this default synchronization support for an additional domain onto which the abstract syntax models have to be mapped. Or, one can customize the triggers to provide a different mapping semantics. Thus, in contrast to GMF, where the mapping is limited to the approach hard-coded in the framework, in ViatraDSM, the designer has complete freedom in creating mappings from concrete syntax models.

As a result, one can automatically derive functionally equivalent editors from GMF descriptions and use VIATRA2's powerful transformation infrastructure to create advanced domain-specific environments.

## Chapter 6

# Summary

### 6.1 Overview

In this report, I presented our effective **incremental model transformation** technology, and to enhance it I extended it with new features.

Despite the significant demand, currently there is no satisfactory support for efficient incremental synchronization of models in any state-of-the-art model transformation systems. As I have shown, the event-driven transformation engine provides a solution for this problem.

The synchronization of the abstract and concrete syntax representations of the domain models is also greatly enhanced: I presented **generic triggers** which can speed up the domain-specific language development.

For complex language constraints, I have outlined an approach based on event-driven transformations and incremental pattern matching which provides efficient support for certain constraint classes.

Tool-to-tool integration is a key issue to increase productivity. I designed a bridge between the ViatraDSM and the Eclipse GMF, so domain-specific languages created in the GMF can be reused in ViatraDSM, thus speeding up the development.

### 6.2 Scientific contributions

- I designed an extended methodology for **event-driven model transformations**.
- I proposed a generic method for the **incremental synchronization** of logical models and their graphical representations in a domain specific modeling environment.
- I proposed a method for incremental, **on-the-fly checking** of complex well-formedness constraints during modeling.
- I designed an automated **software bridge** between the ViatraDSM framework and the Eclipse Graphical Modeling Framework.

### 6.3 Practical accomplishments

- I **implemented** the proposed changes on the incremental model transformation technology in the VIATRA2 model transformation framework.

- I **applied** the proposed improvements in the ViatraDSM framework to address the outlined issues with domain specific modeling.
- I **implemented** the required functions for the ViatraDSM and the Eclipse GMF integration.

## 6.4 Future work

There are still many open questions regarding event-driven transformations.

As I demonstrated in this report, triggers are very versatile, although our formalism is a bit complex for solving simpler problems. The possibility of automatically generating triggers for various problem types should be analyzed. (Note: in this report I demonstrated the use of metamodel-driven generic triggers.) To ease the formulation of triggers, a graphical GT rule / graph pattern designer tool could be created. Furthermore, at the moment the VIATRA2 framework does not allow multiple threads to access the model space, which is why parallel triggers are still executed in a serial manner. To further improve the speed of the engine, these parts of the VIATRA2 framework should be revised.

Although the incremental pattern matcher can speed up the execution of the traditional model transformations (e.g. GT rule execution), triggers could be a more convenient solution in many cases, so the possible automatic generation of triggers from these transformation codes should be examined. To increase the compatibility between triggers and GT rules, the trigger execution engine could be extended to support triggers that were defined by the common left hand side - right hand side graph transformation formalism. Moreover, the use of reference models could be further examined, the use of ASMFunctions should be elaborated.

In terms of constraint evaluation, various extensions could be developed. First, the constraint classes should be analyzed to decide which classes could be handled on-the-fly with triggers. OCL definitions should be examined to decide whether triggers can be generated from them. Even a new metamodel-driven generic constraint handling mechanism could be developed to handle various constraint types. Another challenge is to design the method of how the actual feedback can be shown to the users. Several issues have to be analyzed in this domain, for example, whether inconsistent states should be permitted in some cases, to allow the user to edit and correct the models. Another challenge is to design a quick-fix feature for some constraint classes.

## Appendix A

# Generic DSM diagram triggers

```

machine generic_DSM_diagram_triggers
{
  asmfunction link / 1
  {
    ("diagramroot ") = 'DSM'.diagram;
    ("modelspace root ") = 'DSM'.model;
    ("mappingroot ") = 'DSM'.mapping;
  }

  //*****

  // Create (or just find and map) Editor elements for every Diagram element
  @Trigger(priority='100')
  gtrule linkTopMapping( inout TopMapping, inout DiagramInstance, inout Editor,
    inout RelationDiagram, inout RelationEditor) =
  {
    precondition pattern lhs(TopMapping, DiagramInstance,
      Editor, RelationDiagram, RelationEditor) =
    {
      'DSM'.coremetamodel.'Diagram'(DiagramInstance);

      entity(MetaTopMapping);
      entity(TopMapping);
      supertypeOf(MetaTopMapping, TopMapping);

      entity(Diagram);
      instanceOf(DiagramInstance, Diagram);

      entity(MetaEditor);
      entity(Editor);
      supertypeOf(MetaEditor, Editor);

      relation(RelationEditor, TopMapping, Editor);
      relation(RelationDiagram, TopMapping, Diagram);

      neg find currentTopMapping(DiagramInstance);
      check ((MetaTopMapping == 'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.'TopMapping')
        && (MetaEditor == 'DSM'.coremetamodel.'Editor'));
    }
    action
    {
      try choose EditorInstance with find editorExist(Editor, EditorInstance) do
      let RA= undef, RB= undef, TopMappingInstance= undef in

```

```

seq {
  new (entity (TopMappingInstance) in link( "mappingroot " ));
  new (instanceOf (TopMappingInstance, TopMapping));
  new (relation (RA, TopMappingInstance, EditorInstance));
  new (instanceOf (RA, RelationEditor));
  new (relation (RB, TopMappingInstance, DiagramInstance));
  new (instanceOf (RB, RelationDiagram));
} else let RA= undef, RB= undef,
  TopMappingInstance= undef, EditorInstance= undef in
seq {
  new (entity (EditorInstance) in link( "modelspaceeroot " ));
  new (instanceOf (EditorInstance, Editor));
  new (entity (TopMappingInstance) in link( "mappingroot " ));
  new (instanceOf (TopMappingInstance, TopMapping));
  new (relation (RA, TopMappingInstance, EditorInstance));
  new (instanceOf (RA, RelationEditor));
  new (relation (RB, TopMappingInstance, DiagramInstance));
  new (instanceOf (RB, RelationDiagram));
}
}
}

pattern currentTopMapping (Diagram) =
{
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.TopMapping (TopMapping);
  'DSM'.coremetamodel.'Diagram' (Diagram);
  'DSM'.coremetamodel.'Diagram'.
    'DiagramMapping'.'TopMapping'.diagram (RA, TopMapping, Diagram);
}

pattern editorExist (Editor, EditorInstance) =
{
  entity (Editor);
  'DSM'.coremetamodel.Editor (EditorInstance);
  instanceOf (EditorInstance, Editor);
}

// *****
// Create Nodes for NodeFigures.
@Trigger (priority='90', execution='iterate')
gtrule linkNodeFigure ( inout NodeFigureInstance, inout NodeMappingElement,
  inout Editor, inout RelationTopNodes, inout EditorInstance, inout Node,
  inout RelationNode, inout RelationNodeFigure) =
{
  precondition pattern lhs (NodeFigureInstance, NodeMappingElement, EditorInstance,
    RelationTopNodes, Editor, Node, RelationNode, RelationNodeFigure) =
  {
    'DSM'.coremetamodel.'Diagram' (DiagramInstance);
    'DSM'.coremetamodel.'Diagram'.NodeFigure (NodeFigureInstance);
    'DSM'.coremetamodel.'Diagram'.root (RA, DiagramInstance, NodeFigureInstance);

    'DSM'.coremetamodel.Editor (EditorInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.TopMapping (TopMappingInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
      'TopMapping'.diagram (RB, TopMappingInstance, DiagramInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
      'TopMapping'.editor (RC, TopMappingInstance, EditorInstance);
  }

  entity (MetaNode);
}

```

```

entity (Node);
supertypeOf (MetaNode, Node);

entity (MetaNodeMappingElement);
entity (NodeMappingElement);
supertypeOf (MetaNodeMappingElement, NodeMappingElement);

entity (NodeFigure);
instanceOf (NodeFigureInstance, NodeFigure);

relation (RelationNode, NodeMappingElement, Node);
relation (RelationNodeFigure, NodeMappingElement, NodeFigure);

entity (Editor);
instanceOf (EditorInstance, Editor);
relation (RelationTopNodes, Editor, Node);

neg find mappedNodeFigure (NodeFigureInstance);
check ( (MetaNode == 'DSM'.coremetamodel.'Editor'. 'Node') &&
  ( MetaNodeMappingElement ==
    'DSM'.coremetamodel.'Diagram'. 'DiagramMapping'. 'NodeMappingElement' ) );
}
or
{
'DSM'.coremetamodel.'Diagram'.NodeFigure (DiagramInstance);
'DSM'.coremetamodel.'Diagram'.NodeFigure (NodeFigureInstance);
'DSM'.coremetamodel.'Diagram'.
  'DiagramElement'.subElements (RA, DiagramInstance, NodeFigureInstance);

'DSM'.coremetamodel.'Editor'.Node (EditorInstance);
'DSM'.coremetamodel.'Diagram'. 'DiagramMapping'.NodeMappingElement (TopMappingInstance);
'DSM'.coremetamodel.'Diagram'. 'DiagramMapping'.
  'MappingElement'.diagramElement (RB, TopMappingInstance, DiagramInstance);
'DSM'.coremetamodel.'Diagram'. 'DiagramMapping'.
  'NodeMappingElement'.nodeMapping (RC, TopMappingInstance, EditorInstance);

entity (MetaNode);
entity (Node);
supertypeOf (MetaNode, Node);

entity (MetaNodeMappingElement);
entity (NodeMappingElement);
supertypeOf (MetaNodeMappingElement, NodeMappingElement);

entity (NodeFigure);
instanceOf (NodeFigureInstance, NodeFigure);

relation (RelationNode, NodeMappingElement, Node);
relation (RelationNodeFigure, NodeMappingElement, NodeFigure);

entity (Editor);
instanceOf (EditorInstance, Editor);
relation (RelationTopNodes, Editor, Node);

neg find mappedNodeFigure (NodeFigureInstance);
check ( (MetaNode == 'DSM'.coremetamodel.'Editor'. 'Node') &&
  ( MetaNodeMappingElement ==
    'DSM'.coremetamodel.'Diagram'. 'DiagramMapping'. 'NodeMappingElement' ) );
}
action

```

```

{
  let RA= undef, RB= undef, RC= undef, NodeInstance= undef,
      NodeMappingElementInstance= undef, RMappings= undef,
      MetaMappings='DSM'.coremetamodel.'Diagram'.'DiagramElement'.mappings in
  seq {
    new (entity(NodeInstance) in EditorInstance );
    new (instanceOf(NodeInstance, Node));
    new (relation(RA, EditorInstance, NodeInstance));
    new (instanceOf(RA, RelationTopNodes));
    new (entity(NodeMappingElementInstance) in link( "mappingroot " ));
    new (instanceOf(NodeMappingElementInstance, NodeMappingElement));
    new (relation(RB, NodeMappingElementInstance, NodeInstance));
    new (instanceOf(RB, RelationNode));
    new (relation(RC, NodeMappingElementInstance, NodeFigureInstance));
    new (instanceOf(RC, RelationNodeFigure));

    new (relation(RMappings, NodeFigureInstance, NodeMappingElementInstance));
    new (instanceOf(RMappings, MetaMappings));
  }
}

pattern mappedNodeFigure (NodeFigureInstance) =
{
  'DSM'.coremetamodel.'Diagram'.DiagramElement (NodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.NodeMappingElement (NodeMappingInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'MappingElement'.diagramElement (RA, NodeMappingInstance, NodeFigureInstance);
}

//*****

// Create Edge relations for EdgeFigures.
@Trigger(priority='80', execution='iterate')
gtrule linkEdgeFigure( inout FromNodeInstance, inout ToNodeInstance,
  inout FromNodeFigureInstance, inout ToNodeFigureInstance,
  inout EdgeFigureInstance, inout EdgeMappingElement, inout Edge,
  inout RelationEdge, inout RelationEdgeFigure) =
{
  precondition pattern lhs(FromNodeInstance, FromNodeFigureInstance, ToNodeInstance,
    ToNodeFigureInstance, EdgeFigureInstance, EdgeMappingElement,
    Edge, RelationEdge, RelationEdgeFigure) =
  {
    'DSM'.coremetamodel.'Diagram'.EdgeFigure (EdgeFigureInstance);

    'DSM'.coremetamodel.'Diagram'.NodeFigure (FromNodeFigureInstance);
    'DSM'.coremetamodel.'Diagram'.NodeFigure.
      fromEdges (RA, FromNodeFigureInstance, EdgeFigureInstance);

    'DSM'.coremetamodel.'Editor'.Node (FromNodeInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
      NodeMappingElement (FromNodeMappingInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
      'MappingElement'.diagramElement (RB, FromNodeMappingInstance, FromNodeFigureInstance);
    'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
      'NodeMappingElement'.nodeMapping (RC, FromNodeMappingInstance, FromNodeInstance);

    'DSM'.coremetamodel.'Diagram'.NodeFigure (ToNodeFigureInstance);
    'DSM'.coremetamodel.'Diagram'.NodeFigure.

```

```

    toEdges (RA2, ToNodeFigureInstance, EdgeFigureInstance);

'DSM'.coremetamodel.'Editor'.Node (ToNodeInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
  NodeMappingElement (ToNodeMappingInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
  MappingElement'.diagramElement (RB2, ToNodeMappingInstance, ToNodeFigureInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
  NodeMappingElement'.nodeMapping (RC2, ToNodeMappingInstance, ToNodeInstance);

entity (FromNode);
entity (ToNode);
entity (MetaNode);
supertypeOf (MetaNode, FromNode);
supertypeOf (MetaNode, ToNode);
instanceOf (FromNodeInstance, FromNode);
instanceOf (ToNodeInstance, ToNode);

relation (Edge, FromNode, ToNode);

entity (MetaEdgeMappingElement);
entity (EdgeMappingElement);
supertypeOf (MetaEdgeMappingElement, EdgeMappingElement);

entity (EdgeFigure);
instanceOf (EdgeFigureInstance, EdgeFigure);

relation (RelationEdge, EdgeMappingElement, Edge);
relation (RelationEdgeFigure, EdgeMappingElement, EdgeFigure);

neg find mappedEdgeFigure (EdgeFigureInstance);
check ( (MetaEdgeMappingElement ==
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.EdgeMappingElement')
  && (MetaNode == 'DSM'.coremetamodel.'Editor'.Node) );
}
or
{
  // Second pattern because of the injectivity
  'DSM'.coremetamodel.'Diagram'.EdgeFigure (EdgeFigureInstance);

  'DSM'.coremetamodel.'Diagram'.NodeFigure (FromNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.NodeFigure.
    fromEdges (RA, FromNodeFigureInstance, EdgeFigureInstance);

  'DSM'.coremetamodel.'Editor'.Node (FromNodeInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    NodeMappingElement (FromNodeMappingInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    MappingElement'.diagramElement (RB, FromNodeMappingInstance, FromNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    NodeMappingElement'.nodeMapping (RC, FromNodeMappingInstance, FromNodeInstance);

  'DSM'.coremetamodel.'Diagram'.NodeFigure (ToNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.NodeFigure.
    toEdges (RA2, ToNodeFigureInstance, EdgeFigureInstance);

  'DSM'.coremetamodel.'Editor'.Node (ToNodeInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    NodeMappingElement (ToNodeMappingInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.

```

```

'MappingElement'.diagramElement(RB2, ToNodeMappingInstance, ToNodeFigureInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
'NodeMappingElement'.nodeMapping(RC2, ToNodeMappingInstance, ToNodeInstance);

entity(FromNode);
entity(ToNode);
FromNode = ToNode; // Edge between same type of Nodes
entity(MetaNode);
supertypeOf(MetaNode, FromNode);
supertypeOf(MetaNode, ToNode);
instanceOf(FromNodeInstance, FromNode);
instanceOf(ToNodeInstance, ToNode);

relation(Edge, FromNode, ToNode);

entity(MetaEdgeMappingElement);
entity(EdgeMappingElement);
supertypeOf(MetaEdgeMappingElement, EdgeMappingElement);

entity(EdgeFigure);
instanceOf(EdgeFigureInstance, EdgeFigure);

relation(RelationEdge, EdgeMappingElement, Edge);
relation(RelationEdgeFigure, EdgeMappingElement, EdgeFigure);

neg find mappedEdgeFigure(EdgeFigureInstance);
check ( (MetaEdgeMappingElement ==
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.EdgeMappingElement')
&& (MetaNode == 'DSM'.coremetamodel.'Editor'.Node' ) );
}
action
{
let RA= undef, RB= undef, RMDE1= undef, RMDE2= undef, EdgeInstance= undef,
EdgeMappingElementInstance= undef, RMappings= undef,
MetaMappings='DSM'.coremetamodel.'Diagram'.DiagramElement'.mappings,
MetaDiagramElementRelation='DSM'.coremetamodel.'Diagram'.
'DiagramMapping'.MappingElement'.diagramElement in
seq {
new (relation(EdgeInstance, FromNodeInstance, ToNodeInstance));
new (instanceOf(EdgeInstance, Edge));
new (entity(EdgeMappingElementInstance) in link("mappingroot "));
new (instanceOf(EdgeMappingElementInstance, EdgeMappingElement));
new (relation(RA, EdgeMappingElementInstance, EdgeInstance));
new (instanceOf(RA, RelationEdge));
new (relation(RB, EdgeMappingElementInstance, EdgeFigureInstance));
new (instanceOf(RB, RelationEdgeFigure));

new (relation(RMDE1, EdgeMappingElementInstance, FromNodeFigureInstance));
new (instanceOf(RMDE1, MetaDiagramElementRelation));
new (relation(RMDE2, EdgeMappingElementInstance, ToNodeFigureInstance));
new (instanceOf(RMDE2, MetaDiagramElementRelation));

new (relation(RMappings, EdgeFigureInstance, EdgeMappingElementInstance));
new (instanceOf(RMappings, MetaMappings));
}
}
}

// Diagram element with mapping
pattern mappedEdgeFigure(EdgeFigureInstance) =

```

```

{
  'DSM'.coremetamodel.'Diagram'.DiagramElement (EdgeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    EdgeMappingElement (EdgeMappingInstance);
  'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    'MappingElement'.diagramElement (RA, EdgeMappingInstance, EdgeFigureInstance);
}

//*****

// Complex delete handling
@Trigger (priority='30', execution='iterate')
gtrule deleteHandling( inout Instance, inout MappingElement) =
{
  precondition pattern lhs(Instance, MappingElement)=
  { // Deleted NodeFigure in the model space
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.NodeMappingElement (MappingElement);
    'DSM'.coremetamodel.'Editor'.Node (Instance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
      'NodeMappingElement'.nodeMapping (RA, MappingElement, Instance);
    neg find matchingNodeFigure (MappingElement);
  }
  or
  { // Deleted Node in the model space
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.NodeMappingElement (MappingElement);
    'DSM'.coremetamodel.'Diagram'.NodeFigure (Instance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
      'MappingElement'.diagramElement (RA, MappingElement, Instance);
    neg find matchingNode (MappingElement);
  }
  or
  { // Deleted or moved NodeFigure in the model space
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.EdgeMappingElement (MappingElement);
    'DSM'.coremetamodel.'Editor'.Node (SourceNode);
    'DSM'.coremetamodel.'Editor'.Node (TargetNode);
    'DSM'.coremetamodel.'Editor'.Node'.Edge (Instance, SourceNode, TargetNode);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
      'EdgeMappingElement'.edgeMapping (RA, MappingElement, Instance);
    neg find matchingEdgeFigure (MappingElement);
  }
  or
  { // Deleted Edge in the model space
    'DSM'.coremetamodel.'Diagram'.EdgeFigure (Instance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.EdgeMappingElement (MappingElement);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
      'MappingElement'.diagramElement (RA, MappingElement, Instance);
    neg find matchingEdge (MappingElement);
  }
  action
  {
    seq {
      delete (MappingElement);
      !! Delete model/diagram element also.
      delete (Instance);
    }
  }
}

// NodeFigure element with mapping
pattern matchingNodeFigure (NodeMappingElement) =

```

```

{
  'DSM'.coremetamodel.'Diagram'.NodeFigure(NodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.NodeMappingElement(NodeMappingElement);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'MappingElement'.diagramElement(RA,NodeMappingElement,NodeFigureInstance);
}

// Node element with mapping
pattern matchingNode(NodeMappingElement) =
{
  'DSM'.coremetamodel.'Editor'.Node(NodeInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.NodeMappingElement(NodeMappingElement);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'NodeMappingElement'.nodeMapping(RA,NodeMappingElement,NodeInstance);
}

// EdgeFigure element with mapping (match if _not_ moved edge)
pattern matchingEdgeFigure(EdgeMappingElement) =
{
  'DSM'.coremetamodel.'Diagram'.EdgeFigure(EdgeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.EdgeMappingElement(EdgeMappingElement);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'MappingElement'.diagramElement(RA,EdgeMappingElement,EdgeFigureInstance);

  // Check moved edges:
  'DSM'.coremetamodel.'Diagram'.NodeFigure(FromNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.NodeFigure.
    fromEdges(RB,FromNodeFigureInstance,EdgeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.NodeFigure(ToNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.NodeFigure.
    toEdges(RC,ToNodeFigureInstance,EdgeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'MappingElement'.diagramElement(RD,EdgeMappingElement,FromNodeFigureInstance);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'MappingElement'.diagramElement(RE,EdgeMappingElement,ToNodeFigureInstance);
}

// Edge element with mapping
pattern matchingEdge(EdgeMappingElement) =
{
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.EdgeMappingElement(EdgeMappingElement);
  'DSM'.coremetamodel.'Editor'.Node(SourceNode);
  'DSM'.coremetamodel.'Editor'.Node(TargetNode);
  'DSM'.coremetamodel.'Editor'.Node'.Edge(EdgeInstance,SourceNode,TargetNode);
  'DSM'.coremetamodel.'Diagram'.'DiagramMapping'.
    'EdgeMappingElement'.edgeMapping(RA,EdgeMappingElement,EdgeInstance);
}

//*****

// Name synchronisation
@Trigger(priority='10')
gtrule synchNames( inout ModelInstance, inout FigureInstance,
  inout MappingElementInstance) =
{
  precondition pattern lhs(ModelInstance, FigureInstance, MappingElementInstance) =
  {
    // Synchronise the name of edges
    'DSM'.coremetamodel.'Editor'.Node(SourceNode);
    'DSM'.coremetamodel.'Editor'.Node(TargetNode);
  }
}

```

```

'DSM'.coremetamodel.'Editor'.Node'.Edge (ModelInstance, SourceNode, TargetNode);
'DSM'.coremetamodel.'Diagram'.EdgeFigure (FigureInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    EdgeMappingElement (MappingElementInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    'EdgeMappingElement'.edgeMapping (RA, MappingElementInstance, ModelInstance);
'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
    'MappingElement'.diagramElement (RB, MappingElementInstance, FigureInstance);
check ( name (ModelInstance) != name (FigureInstance) );
}
or
{
    // Synchronise the name of nodes
    'DSM'.coremetamodel.'Editor'.Node (ModelInstance);
    'DSM'.coremetamodel.'Diagram'.NodeFigure (FigureInstance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
        NodeMappingElement (MappingElementInstance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
        'NodeMappingElement'.nodeMapping (RA, MappingElementInstance, ModelInstance);
    'DSM'.coremetamodel.'Diagram'.DiagramMapping'.
        'MappingElement'.diagramElement (RB, MappingElementInstance, FigureInstance);
    check ( name (ModelInstance) != name (FigureInstance) );
}
action
{
    rename (ModelInstance, name (FigureInstance));
    rename (MappingElementInstance, name (FigureInstance));
}
}

//*****

rule main() =
seq
{
    println ( startTrigger( "linkTopMapping " ) );
    println ( startTrigger( "linkNodeFigure " ) );
    println ( startTrigger( "linkEdgeFigure " ) );
    println ( startTrigger( "deleteHandling " ) );
    println ( startTrigger( "synchNames " ) );
}
}

```

# Bibliography

- [1] The Eclipse project. [www.eclipse.org](http://www.eclipse.org) .
- [2] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, 2006. In press.
- [3] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró. Incremental pattern matching in the VIATRA transformation system. In *GRaMoT'08, 3rd International Workshop on Graph and Model Transformation*. 30th International Conference on Software Engineering, 2008. Submitted.
- [4] Gábor Bergmann and András Ökrös. Event-based model transformations with incremental pattern matching. Student Report (TDK), Budapest University of Technology and Economics, October 2007.
- [5] Bernd Kolb, Markus Völter. *openArchitectureWare and Eclipse*. <http://architekturware.sourceforge.net/data/oawEclipse.pdf> .
- [6] E. Börger and R. Särk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [7] Eclipsepedia. Gmf tutorial. [http://wiki.eclipse.org/index.php/GMF\\_Tutorial](http://wiki.eclipse.org/index.php/GMF_Tutorial) .
- [8] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
- [9] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as Eclipse plug-ins. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 134–143. ACM, 2005.
- [10] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.
- [11] Esther Guerra and Juan de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2004.
- [12] I. Ráth and G. Bergmann and A. Ökrös and D. Varró. Live model transformations driven by incremental pattern matching. 4th International Conference on Model Transformations, 2008. Submitted.

- [13] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in vmts. In *Proc. GraBaTs 2004: International Workshop on Graph Based Tools*. Elsevier, 2004.
- [14] Metacase. MetaEdit+. <http://www.metacase.com/mep/> .
- [15] Microsoft. DSL Tools. <http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx> .
- [16] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, 2000. ACM Press.
- [17] Object Management Group. *Meta Object Facility 2.0 Queries / Views / Transformations*. <http://www.omg.org> .
- [18] Object Management Group. *Action Semantics for the UML*, August 2001. <http://www.omg.org> .
- [19] Object Management Group. *Model Driven Architecture — A Technical Perspective*, September 2001. <http://www.omg.org> .
- [20] Object Management Group. *Object Constraint Language Specification (in UML 1.4)*, 2001. <http://www.omg.org> .
- [21] Object Management Group. *Meta Object Facility Version 2.0*, 2003. <http://www.omg.org> .
- [22] Object Management Group. *UML Semantics Version 2.0*, May 2003. <http://www.omg.org> .
- [23] The Eclipse Model Development Tools project. Official homepage. <http://www.eclipse.org/modeling/mdt/?project=ocl#ocl> .
- [24] István Ráth. Declarative specification of domain specific visual languages. Master's thesis, Budapest University of Technology and Economics, 2006.
- [25] István Ráth, András Schmidt, and Dávid Vágó. Automated model transformations in domain specific visual languages. Student Report (TDK), Budapest University of Technology and Economics, October 2005.
- [26] Arend Rensink. Representing first-order logic using graphs. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy*, volume 3256 of LNCS, pages 319–335. Springer, 2004.
- [27] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
- [28] Andy Schürr. Specification of graph translators with triple graph grammars. In B. Tinhofer, editor, *Proc. WG94: International Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer, 1994.
- [29] The Eclipse Project. Eclipse Modeling Framework. <http://www.eclipse.org/emf> .
- [30] The Eclipse Project. Graphical Editing Framework. <http://www.eclipse.org/gef> .

- [31] The Eclipse Project. Graphical Modeling Framework. <http://www.eclipse.org/gmf> .
- [32] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
- [33] Dániel Varró. *Automated Model Transformations for the Analysis of IT Systems*. PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, May 2004.
- [34] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.