



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

**Event-based model transformations with incremental
pattern matching**
and their application in domain specific modeling

Scientific Students' Associations Report

Gábor Bergmann
András Ökrös

Supervisors:

István Ráth, PhD student

Dr. Dániel Varró, assistant professor
Gergely Varró, assistant professor

Budapest, October 2007

The authors would like to thank the supervisors István Ráth, Dr. Dániel Varró and Gergely Varró for their continued support, friendly advice, and enthusiasm. We would also like to thank Ákos Horváth for helping us with many VIATRA2 related suggestions. We would like to thank Máté Kovács for providing one of the benchmark problems, Bálint Kasza for providing a complex model transformation example to be used in our measurements, and Dániel Tóth for his numerous valuable observations.

Contents

1	Introduction	6
1.1	Model based development in software engineering	6
1.1.1	Model Driven Architecture	6
1.1.2	MDA Development steps	7
1.2	Transformations in MDA	7
1.2.1	Graph transformation and metamodeling	7
1.3	Problems and challenges in model transformations	8
1.3.1	Incremental synchronisations with QVT	8
1.3.2	Triple graph grammars	8
1.4	Domain-Specific Modeling	9
1.4.1	Domains in MDA	9
1.4.2	Problems and challenges in domain-specific modeling	9
1.5	Objectives	11
1.6	Structure of the Report	11
2	Background technologies and concepts	12
2.1	Model transformation in VIATRA2	12
2.1.1	Metamodeling: Definition of Abstract Syntax	12
2.1.2	The VTML language	13
2.1.3	The VTCL language	15
2.1.4	VIATRA2 Architectural overview	19
2.2	The ViatraDSM framework	22
2.2.1	Domain-specific modeling in depth	22
2.2.2	Architecture	23
2.2.3	Modeling	24
2.2.4	Diagrams	26
2.2.5	Transformations, simulation and code generation	28
2.2.6	Required steps for a complete domain-specific editor	28
2.3	RETE networks	29
2.3.1	Origin and applications	29
2.3.2	Components and structure	29
2.3.3	Operations	30
2.3.4	Example	30
2.3.5	Alternatives	31

3	Incremental pattern matching	34
3.1	Goal of this chapter	34
3.2	Applying the RETE concept on VPM modelspaces	34
3.2.1	Tuples	35
3.2.2	Inputs	35
3.2.3	Nodes	37
3.2.4	Arbitrary term evaluation	41
3.2.5	Applications in pattern matching	43
3.3	Building a RETE net from GTASM patterns	44
3.3.1	GTASM patterns as constraint systems	44
3.3.2	The construction algorithm	45
3.3.3	Employed node configurations	46
3.3.4	Constraint ordering	49
3.3.5	Example pattern matcher	50
3.4	Performance	50
3.4.1	Improvements and optimizations	50
3.4.2	Benchmark: Sierpinski triangles	53
3.4.3	Measurements with a practical application	54
3.4.4	Conclusions	54
3.5	Summary	55
4	Event-driven incremental transformations with triggers	56
4.1	Goal of this Chapter	56
4.2	Event-driven model transformations with triggers	57
4.2.1	The definition of our trigger concept	57
4.2.2	Adapting triggers into the VIATRA2 model transformation framework	58
4.2.3	Trigger example	58
4.3	Execution semantics	59
4.3.1	Events and the delta set	59
4.3.2	Operational semantics	60
4.4	Handling synchronisational problems	61
4.4.1	Description of the problem	61
4.4.2	Our approach	62
4.5	Architecture of the implementation	63
4.5.1	Delta Monitor	64
4.5.2	Transaction handling	64
4.5.3	VIATRA2 user interface extension	64
4.6	Summary	64
5	Event-driven transformations in domain-specific modeling	65
5.1	Goal of this Chapter	65
5.2	Challenges in the domain-specific modeling world	65
5.2.1	Separation of abstract and concrete syntax representations	65
5.2.2	Evaluation of complex constraints	66
5.2.3	Transformations in domain-specific visual languages	67
5.3	Abstract-concrete syntax synchronisation	67
5.3.1	Architectural changes in ViatraDSM	67
5.3.2	Petri net abstract-concrete syntax synchronisation	67
5.3.3	Conclusion	72

5.4	Incremental constraint checking	72
5.4.1	Architectural changes in ViatraDSM	72
5.4.2	Petri net capacity constraint	73
5.4.3	Conclusion	76
5.5	Incremental code generation	77
5.5.1	The task	78
5.5.2	Our solution	79
5.5.3	Code generation with triggers	81
5.5.4	Conclusion	85
5.6	Summary	85
6	Summary	86
6.1	Overview	86
6.2	Scientific contributions	86
6.3	Practical accomplishments	87
6.4	Future work	87

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* [25] to provide a visionary paradigm of software development. MDA relies on OMG's other flagship technologies: Unified Modeling Language (UML) [28], the Meta Object Facility (MOF) [27], 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.

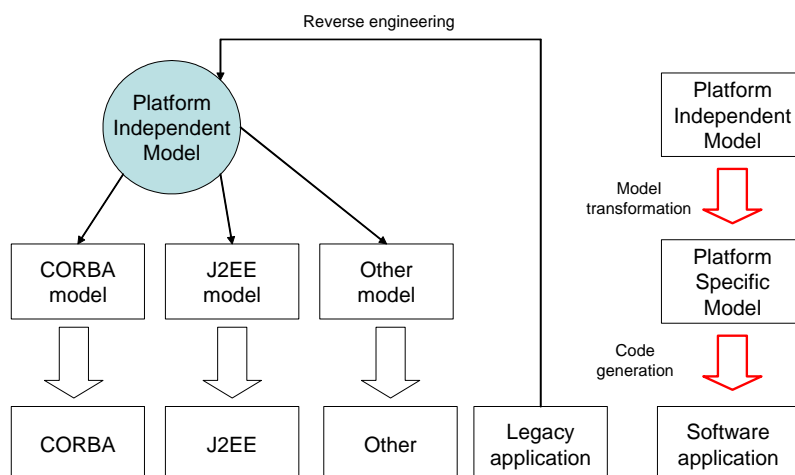


Figure 1.1: Model Driven Architecture

1.1.2 MDA Development steps

As it can be seen on Figure 1.1, MDA emphasizes the clear distinction between Platform Independent Models (PIM) and Platform Specific Models (PSM), thus, software development in MDA is envisioned as a three-step process. First, the 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) [26], and behavioral semantics described in Action Semantics (AS) [24] 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 [41]:

- 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 standard.

1.2.1 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 [36] 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, SOS rules, dataflow nets, etc.). [39]

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

1.3 Problems and challenges in model transformations

As we have shown in Section 1.1, model transformation has a crucial importance in every model based development process. Moreover, there are scenarios, where an interactive synchronization of models would be desirable, with the changes applied to one model immediately reflected on the other model. A possible example for such a scenario would be an object-relational mapping, with a graphical object model being edited by the user, and the generated relational database schema being continuously updated and displayed.

1.3.1 Incremental synchronisations with QVT

In the QVT standard, supporting the incremental propagation of changes to models is a required feature. Many model transformation systems claim to be fully QVT-compliant. However, in reality, as QVT is a recent standard, these implementations are still in their early stages, and none of them can efficiently accomplish incremental change propagation yet.

Graph rewriting tools have matured greatly since their introduction, and they have a strong mathematical background. Thus adopting the QVT standard in a graph modeling tool could be an obvious choice. However, these systems were initially designed for batch transformation, so they are inefficient in incremental synchronisation. They also suffer at ALAP (as long as possible) type transformations, where they have to match patterns often, with minor modifications to the graph each time. See [42] for measurements.

The core of the problem is that incrementally updating target models is inefficient if the transformation (pattern matching) engine itself cannot incrementally update the precondition occurrences of the transformation rules; most current QVT-compliant model transformation engines do not have this feature.

1.3.2 Triple graph grammars

In graph modeling environments, synchronisation can be done by adopting triple graph grammars. Although this was introduced by Andy Schürr [37] in 1994, it is still yet to become a widely used technique. TGGs can be applied for defining the relationship between two different models. The power of this definition is that it can be made operational [17], so that one model can be transformed into another in either direction. Moreover, multiple TGGs can be used to synchronise two models incrementally.

The main problem with TGG is that in most implementations three conventional GT rules are created from a TGG definition, and the execution of these rules is done in the same non-incremental way as described in the last section. Thus, this approach is inefficient, if the transformation (pattern matching) engine itself cannot incrementally update the precondition occurrences of the transformation rules.

1.4 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 informatics industry.

1.4.1 Domains in MDA

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

1.4.2 Problems and challenges in domain-specific modeling

Many of the original ideas of DSM are not achieved in the available DSM framework implementations. These ideas were easy, automated language engineering, domain integration, etc., here we present three problems which we think are of great importance.

The first is that the on-the-fly evaluation of complex language specific constraints (e.g. design pattern conformity checking) is expensive, or even impossible in the current DSM frameworks.. Expensive means, that these evaluations usually carried out by running 'batch evaluators', which are time consuming task, especially on large models. The current DSM frameworks on-the-fly

constraint handling (if there are any) is usually limited to checking simple static constraints (e.g. attributes, multiplicities). But we think it is important to evaluate complex well-formedness rules incrementally (on-the-fly) on our models, because the sooner we found an error, the less it will cost to repair.

The second is the synchronisation of the logical (*abstract syntax*) and graphical (*concrete syntax*) representations of the domain models. To ensure incremental synchronisation of these models the possible graphical representation for a given logical model is limited by the currently available DSM frameworks. This is a crucial problem, because one of the main intentions of DSM, was to provide an easily readable and usable visualization for the domain experts.

The third is the incremental code generation. Normal code generation is usually supported by DSM tools, but incremental code generation is not supported by any of them. Conventional code generation is usually a time consuming single-pass top-down process (i.e., for each change in the model the code has to be regenerated from the beginning). On the other hand, with incremental code generation, not just code generation can become faster, but even the whole design process, as the user, who is editing the model, can immediately see the changes in the generated code.

So we can conclude that, to improve DSM framework implementations the handling of general model transformations seems to be inevitable. Thus the concept of ViatraDSM, which is a DSM framework built on a graph transformation system, seems to be adequate. Although as we discussed, these system were initially designed for batch transformation, so they are inefficient in incremental synchronisation.

1.5 Objectives

In this paper we propose an effective incremental model transformation technology, and apply it for various model synchronisation and constraint checking problems in domain specific modeling.

Our detailed objectives for scientific contribution:

- We propose an efficient **incremental pattern matcher** that stores partial and complete pattern occurrences and updates them incrementally on modifications to the model.
- We design a methodology for **event-driven model transformations**, where actions are **triggered** when conditions and constraints become satisfied or violated in the model.
- We propose a method for incremental, **on-the-fly checking** of complex well-formedness constraints during modeling.
- We propose a method for the **incremental synchronisation** of logical models and their graphical representations in a domain specific modeling environment.

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

- We **implement** the proposed components of incremental model transformation technology in the VIATRA2 model transformation framework, to provide support for truly incremental change propagation.
- We **measure** the efficiency of the implemented pattern matcher.
- We **apply** the proposed domain specific modeling improvements in the ViatraDSM domain specific modeling framework to address the outlined issues with domain specific modeling.

1.6 Structure of the Report

In Chapter 2, we introduce the technical and theoretical background of our work. The first section is dedicated to VIATRA2, the model transformation framework utilized by our work. The next part introduces the domain specific modeling framework that Chapter 5 relies on: ViatraDSM, extension of VIATRA2. The chapter concludes with a brief introduction to the RETE algorithm, that will be adapted in Chapter 3 as an incremental pattern matcher for the VIATRA2 framework.

In Chapter 3, a new, incremental pattern matcher component for the VIATRA2 framework is proposed. The RETE algorithm, a classical incremental pattern matcher from the field of rule based expert systems, is conceptually adapted into the VIATRA2 environment. An algorithm for constructing RETE networks from the patterns of the VIATRA2 framework is also given. The chapter additionally discusses optimizations and improvements to the pattern matcher, and shows measurement results comparing the performance of different pattern matchers.

In Chapter 4, an event-driven transformation methodology is elaborated, that relies on the incremental pattern matcher, and enables transformation rules to be triggered by conditions defined on the model. An event-driven transformation engine for the VIATRA2 framework is designed.

In Chapter 5, the event-driven transformation engine of Chapter 4 is put to use, to enhance the capabilities of the ViatraDSM framework with interactive constraint checking and incremental abstract syntax synchronisation. The designed system is demonstrated via a code generation example.

Finally, Chapter 6 gives an overview of our 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, we 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) [27] metamodeling standard issued by the OMG. However, as stated in [41], 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) [41] 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.

The VPM language consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). An *entity* represents a basic concept of a (modeling) domain, while a *relation* represents the relationships between other model elements¹. 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).

¹Most 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.

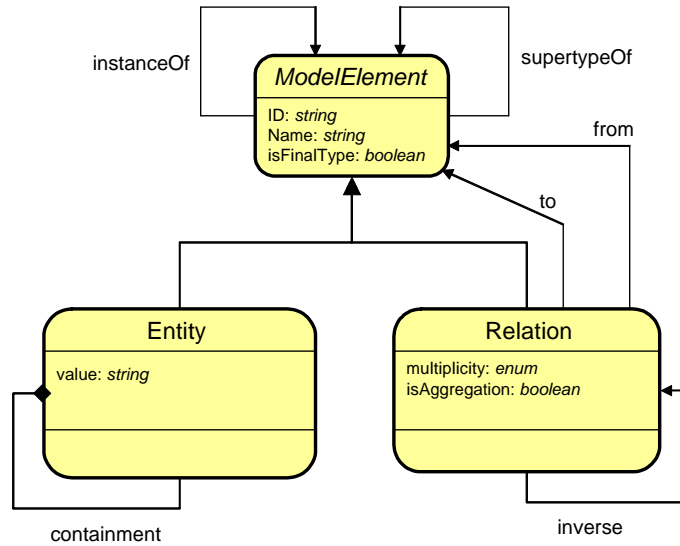


Figure 2.1: The VPM Metamodel

- There is an entity with no parent: the *root entity* is the root of name hierarchy. The fully qualified names of the children of the root entity equal the name of the child entity.

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

All elements have a globally unique ID, which 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:

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

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 VTML language

In VIATRA2, the textual metamodeling language supporting VPM is called VTML (Viatra Textual Metamodeling Language). The technicalities of VTML are demonstrated in Fig. 2.2 on a simplified UML metamodel presented originally in the model transformation benchmark of [11].

The VTML equivalent of the metamodel is as follows.

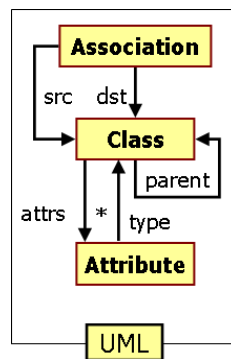


Figure 2.2: Sample UML metamodel

```

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

```

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

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

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

Special relationships can be represented by the keywords *supertypeOf*, and *subtypeOf* for generalization, and *typeOf*, and *instanceOf* for instantiation. The syntax is the following: $\langle relationship \rangle(\langle supplier \rangle, \langle client \rangle)$. For example, `typeOf(UML.Class, Dog)` defines that the entity `Dog` is an instance of the metamodel element `UML.Class`. This way, a model element may have multiple types to support multi-domain modeling.

2.1.3 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) [9] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [5] rules can be used for the description of control structures.

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

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 [40]. 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 [35].

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 [9], 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.

The sample graph transformation rule in Figure 2.3 defines a refactoring step of lifting an attribute from child to parent classes. This means that if the child class has an attribute, it will be lifted to the parent.

The VTCL language allows both popular notation for defining graph transformation rules. The first syntax of a GT rule specification corresponds to the traditional notation: it contains a

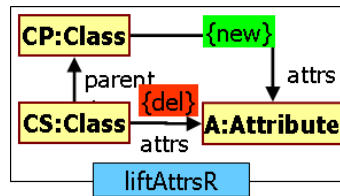


Figure 2.3: Sample graph transformation rule

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 [23]) 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.

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, Attr);

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

// 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 [5] that have formal semantics and correspond to the constructs in conventional programming languages.

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

In VTCL, a special class of functions, called *native 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 modelspace. 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);
}

```

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

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

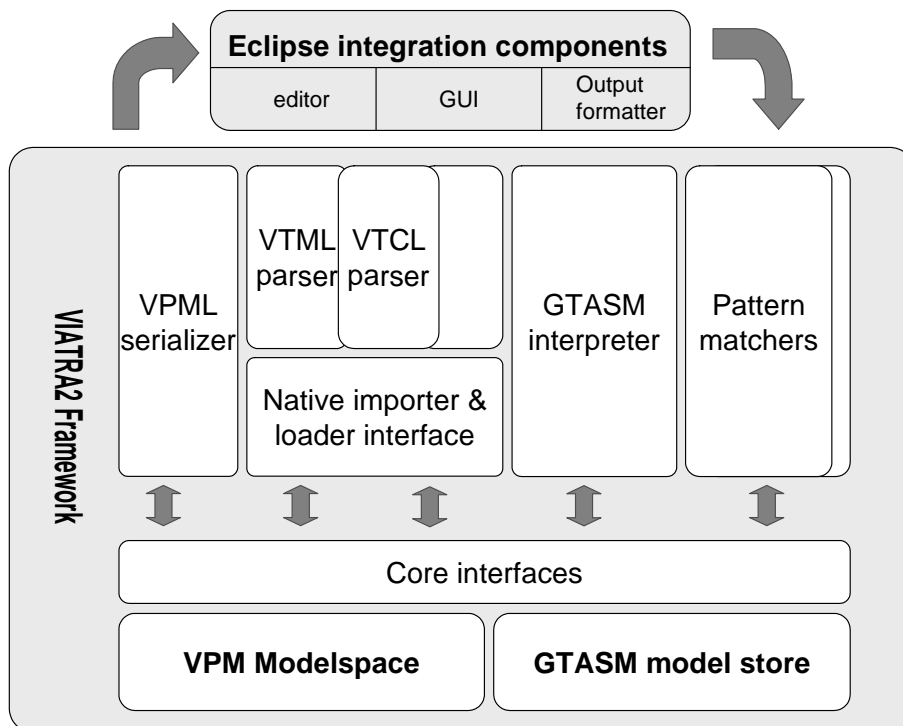


Figure 2.4: The architecture of the VIATRA2 framework

As it can be seen on Fig. 2.4, the internal structure of the VIATRA2 framework can be split up into four major components:

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

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

Import/export facilities To facilitate the integration of the VIATRA2 framework into an existing model-driven development infrastructure, the *native importer* interface provides support for the construction of import plug-ins which read native formats and instantiate models in the VPM modelspace. The 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.

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.

2.2 The ViatraDSM framework

In this section, we 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 [34], since then, it has matured greatly. During the introduction of this framework, we conceptually follow [33].

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.

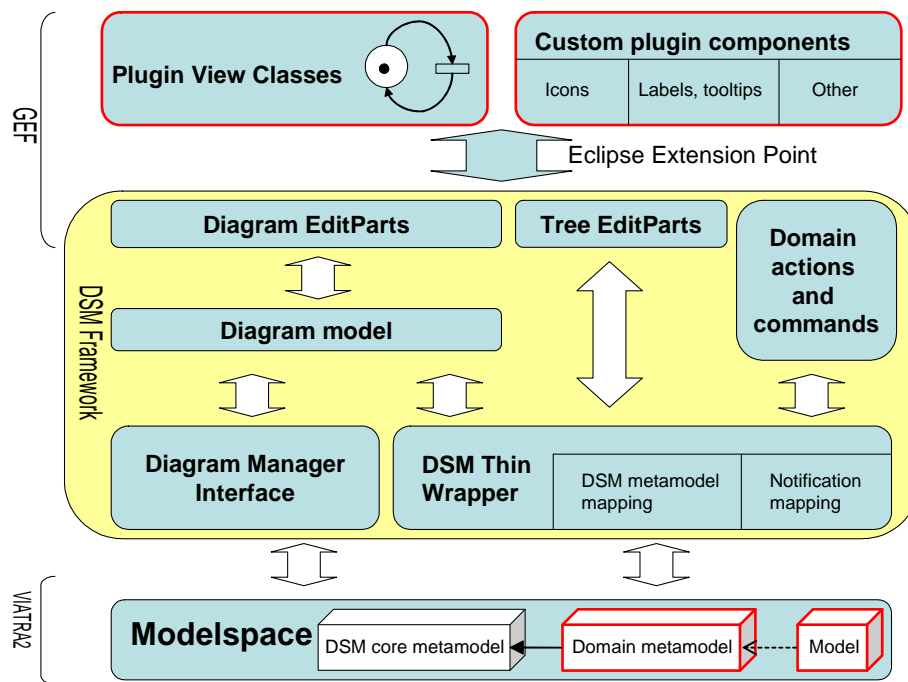


Figure 2.5: Detailed architecture of the DSM framework

While most state-of-the-art implementations, such as the Eclipse project’s Graphical Modeling Framework [38], or Microsoft’s DSL Tools suite [20], 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 [18], 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 [4], as well as numerous research projects such as TIGER [10] have implemented domain-specific modeling at various levels, ViatraDSM offers a different, model transformation-based perspective on this technology.

2.2.2 Architecture

Figure 2.5 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 2.5) the runtime framework approach is reflected by the element *DSM metamodel mapping*. That component is responsible for mapping a domain metamodel inside the VIATRA2 modelspace 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* on Figure 2.5. 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 2.6 shows the structure of the core domain metamodel; in the following paragraphs, the elements of this metamodel are described.

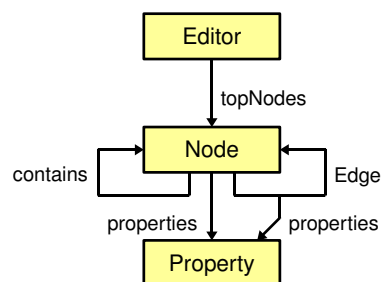


Figure 2.6: The Core domain metamodel of the DSM framework

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. On Figure 2.6, 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.

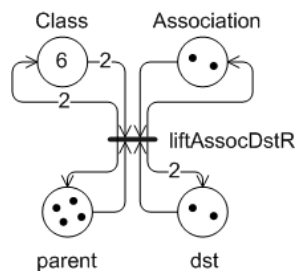


Figure 2.7: A sample Petri net with place capacities and arc weights

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.

On Figure 2.8, 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 Transitions, whereas *InArc* is the opposite, and edge from a Transition to a Place. Both kind of edges may have a property entitled *weight*.

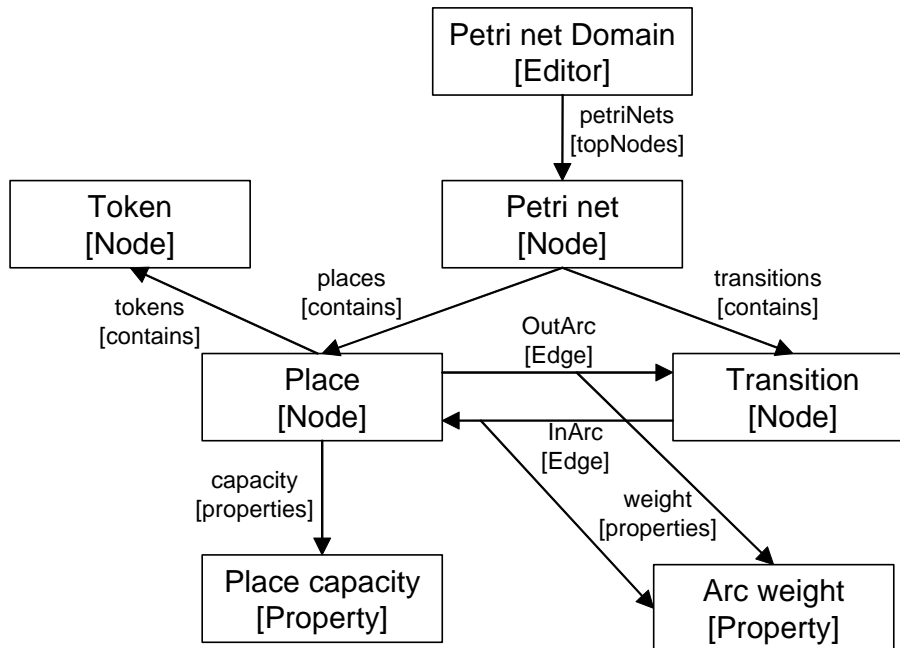


Figure 2.8: Example of a domain metamodel (Petri net domain)

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 Fig. 2.9).

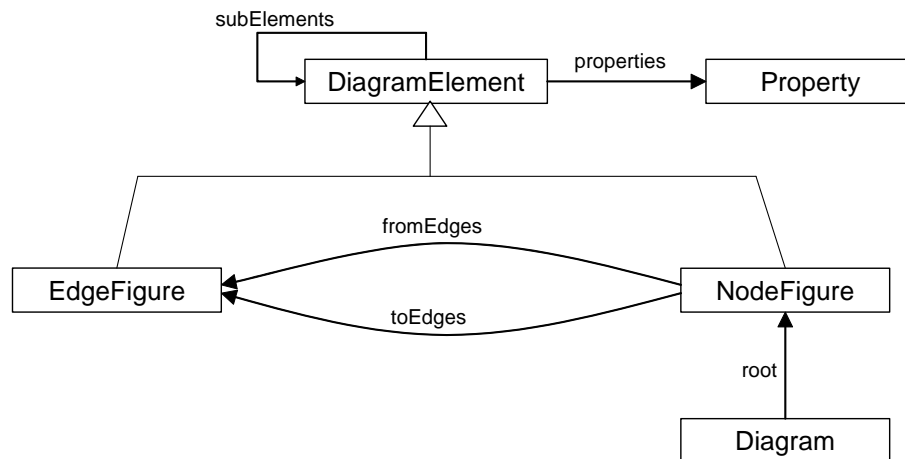


Figure 2.9: Diagram metamodel of the DSM framework.

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*).

On Figure 2.10, an example diagram metamodel can be seen. (The names inside the square brackets give the core metamodel supertypes of each element.)

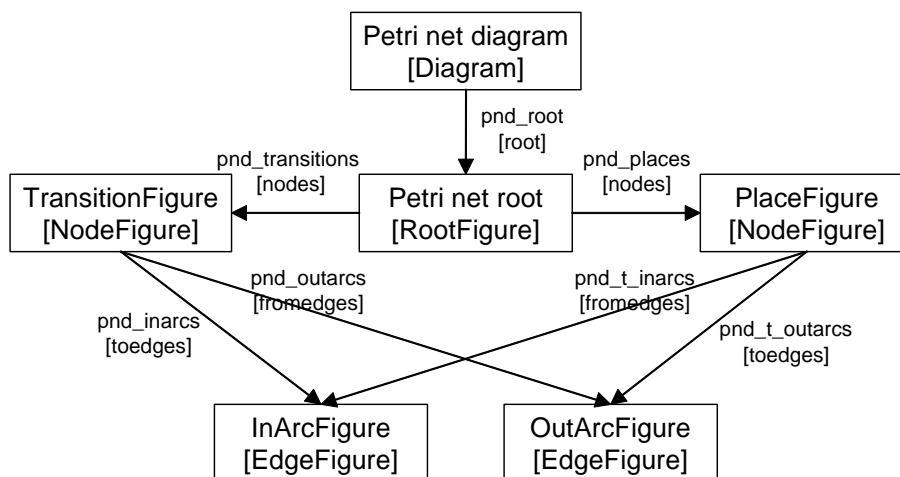


Figure 2.10: Example of a diagram metamodel (Petri net domain)

2.2.5 Transformations, simulation and code generation

An essential concept in the ViatraDSM framework is the support for transformations, both model transformations and code generators. Basically every existing DSM tool provides code generation functions, but only a few support model transformations. The advantage of model transformations is that using them, the models can be simulated with the DSM tool. Simulations is just one use of model transformation, but model transformations are more powerful than just a plain simulation engine. Using model transformations, automated model manipulation becomes possible.

2.2.6 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.
5. Define dynamic semantics using the VIATRA2 Textual Command Language, or the model transformation editor.
6. Define a code generator using the VIATRA2 Textual Command Language, or the model transformation editor.

A finished Petri net editor with custom graphical elements can be seen on Figure 2.11. Note that the language engineer only has to learn *one* expressive domain-specific language for *all* aspects of language engineering.

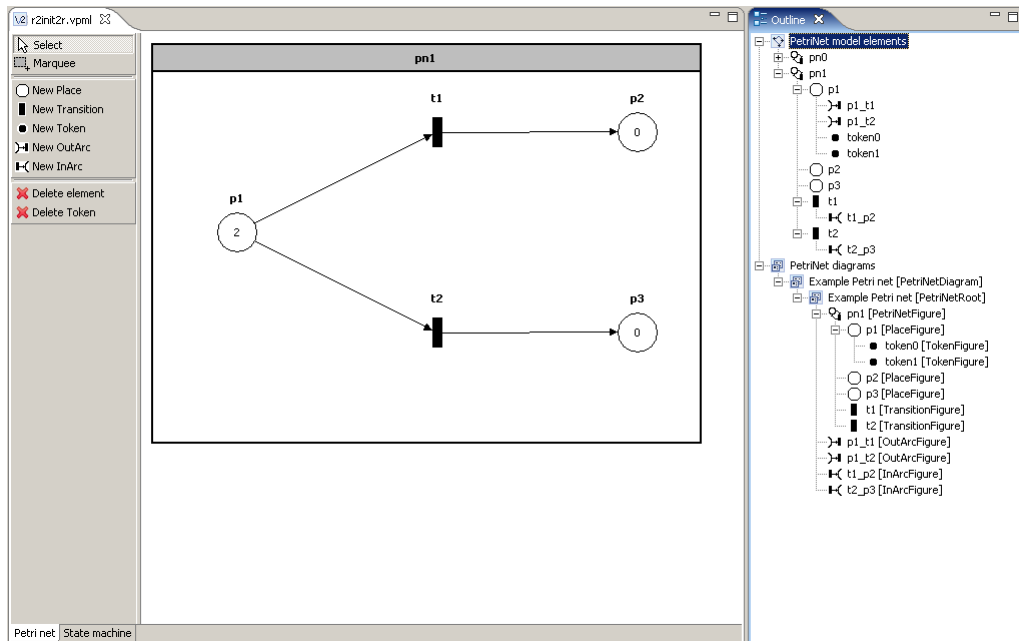


Figure 2.11: Petri net editor with custom graphical elements

2.3 RETE networks

2.3.1 Origin and applications

Model transformation is not the only field where the demand for incremental pattern matching has arisen. In fact, incremental pattern matching has been widely utilized in rule based expert systems for more than two decades.

The RETE algorithm (see [13]) is a widely known forward chaining approach of pattern matching frequently used in production rule systems. The algorithm builds a network of data processing nodes that recognises multiple patterns in a large set of facts. The RETE network keeps track of partial and complete pattern matches with each modification to the knowledge base.

The main benefit is that the set of matches (known as the *conflict set*) of any pattern can be found virtually instantly, without having to re-evaluate the pattern conditions over the large fact base. The drawbacks are the considerable memory consumption of the network and the additional overhead of updating the network upon each modification.

The concept was originally developed by Charles L. Forgy and published in a series of papers. Several rule based expert systems employ variants of this idea, including JBoss Rules (also known as Drools, [12]), RC++ [44], Jess.

2.3.2 Components and structure

The RETE network is a DAG (directed acyclic graph)² whose nodes contain and process units of data (called tokens or working memory elements, *WME*) and transmit them along the edges of the network. Although in practice, there are several variations of the basic idea; now we try to give an overview of commonly encountered features of RETE.

RETE networks contain nodes of various types. There is a distinguished set of nodes (usually one single node) called *input nodes* that contain the asserted facts of the knowledge base. So-

²If recursive patterns are involved, the DAG property may be violated

called *alpha nodes* are connected with an edge to a parent node (usually the input node or another alpha node); they filter the contents of the parent node according to some constant criteria (e.g. type). The key components are the *beta nodes*, that have two separate input slots, each connected to a node³ in the network. The contents of a beta node are compound WMEs built from two input WMEs (one from each slot) that are paired by some criteria. Typically, beta nodes perform a natural join operation (as in relational algebra) on the contents of their parent nodes. Finally, a distinguished *production node* for each pattern collects the matches of the pattern.

The RETE matcher is highly flexible, making a wide range of pattern matching strategies possible. A single node may have any number of children. This enables nodes to be shared between patterns or between parts of the same pattern. A node can have several incoming edges⁴ and treat the union of the contents of its parents as its input. A pattern can be matched by a linear sequence of beta nodes, each expanding the partial match by an additional fact, or a more complex (but less deep) network composed of converging subnetworks responsible for different parts of the pattern.

Note that while this introduction refers to WMEs as actually being stored at nodes, this is merely a way of explaining the basics of the concept. In actual implementations this may not be the case, as it is possible for some nodes not to contain a memory. Some RETE network descriptions put emphasis on isolating local *memory* storages, that are components responsible for storing (and possibly indexing) WMEs, and all memories together form a distributed working memory. It is possible to distinguish between alpha memories and beta memories, based on whether they store WMEs that are simple asserted facts or compound WMEs output by beta nodes. Section 3.2 covers the memory aspects of our implementation in detail.

2.3.3 Operations

Once the RETE net is built, finding the matches of a pattern is as simple as retrieving the contents of the production node corresponding to the pattern.

If the knowledge base undergoes changes, the RETE network has to be updated in order to keep the conflict set up-to-date. Whenever a new fact is asserted, a *positive update* token containing the new fact is passed to the input node. Update tokens will propagate through the network along edges, reaching and influencing a part of it, possibly even modifying the conflict set. Alpha nodes in filtering roles will pass a token to their children if the fact enclosed in the token satisfies the condition associated with the alpha node. Beta nodes look for WMEs from their other input slot that are pairable with the incoming token; for each suitable pair is found, a new compound WME is created from them and propagated to the children of the beta node. The WMEs involved in the process are added to memories encountered along the way.

If a fact is revoked from the knowledge base, the network has to be notified. This procedure is very similar to the previous one, *negative update* tokens are propagated in the network. The only key difference is that WMEs have to be retracted from the memories instead of being added.

2.3.4 Example

Let's consider the following example problem: given a set of letters from the English alphabet, the objective is to find among them pairs of successive letters, where either the first one is a consonant and the second one is a vowel, or the first one is a vowel and the second one is a

³Some sources require the secondary input to be a child of an alpha node

⁴nodes having multiple parents is a case significantly different from beta nodes having two input slots, each with its own parent

bilabial⁵ consonant. A RETE network built for finding this pattern and preloaded with the input set {a,b,e,h,i,m,o,p,s,t,u} is illustrated by figure Figure 2.12. Various shapes represent the different kind of nodes, small boxes represent the WMEs contained by the nodes, and continuous lines represent the network edges (ignore the dashed lines for now). Two alpha nodes are connected to the input node, one for filtering vowels, the other is for filtering consonants. Additionally, the alpha node responsible for consonants has another alpha node as a child, for the purpose of filtering bilabials from all consonants. One beta node joins consonants with vowels, the other one joins vowels with bilabial consonants to find pairs that are successive in the alphabet. The two result sets are united in a production node.

The Figure 2.12 demonstrates how a positive update affects the RETE network. First, the input node is notified that the fact base has been changed: a new element, *d* has just been added. The input node propagates the WME containing *d* as a positive update token to both its children. While the alpha node responsible for vowels ignores the update, *d* passes the filter of the consonant node, so from now on that node contains *d*, and also propagates it further from there. As *d* is not bilabial, the bilabial node ignores it. The beta node on the left receives the update at its left-side input, then it checks the new WME against WMEs contained by its other parent: *a*, *e*, *i*, *o*, and *u*. Out of these, *e* satisfies the link criteria (*e* comes after *d* in the alphabet), so the beta node forms a new WME from them, and sends the update to its child, the production node. The new occurrence of the pattern appears at the production node. If *d* was now removed from the fact base, a negative update token would enter the network and travel the very same path that the preceding positive update has, ultimately removing the new pattern occurrence from the production node.

2.3.5 Alternatives

While the successful RETE algorithm has numerous variations itself, there are also several alternatives, many of which more or less resemble the idea behind RETE. The most important target of improvement is the high memory consumption of the RETE network.

TREAT [21] aims at minimizing memory usage while retaining the incremental property of pattern matching and instant accessibility of conflict sets. Only the input facts and the conflict sets are stored, no memories are used for partial patterns. Asserting a new fact requires combining it with other input WMEs in order to calculate the changes to the conflict sets; all input memories involved in a pattern must be used as partial matches are not available. Deletion is done by removing WMEs containing the revoked fact from input nodes and production nodes, which is often less time-consuming than performing joins. Some sources claim that TREAT is faster than RETE, others disagree ([22] states various arguments and measurements in favor of RETE). It is also important to note that the TREAT concept does not seem to offer the same level of flexibility as RETE does.

RETE* [44] is a generalization of RETE that attempts to strike a balance between memory size and performance by keeping beta memories stored for frequently used nodes and generating them on-the-fly for the rest; the two extreme cases for the memory retention policy are TREAT and RETE.

[43] describes a pattern matching tree for graph transformation purposes that is analogous to a RETE network. This matcher is characterised by a tree-shaped, remotely RETE-like search plan. It relies on the assumption that facts (graph edges) can be described as pairs. The main advantage of this solution is that (partial) pattern occurrences need not be physically stored, possibly saving a significant amount of memory. WMEs consist of a reference to a parent WME and a single graph node that extends the parent WME; the WMEs form a tree similar to the pattern matcher tree.

⁵bilabial consonants are articulated with both lips

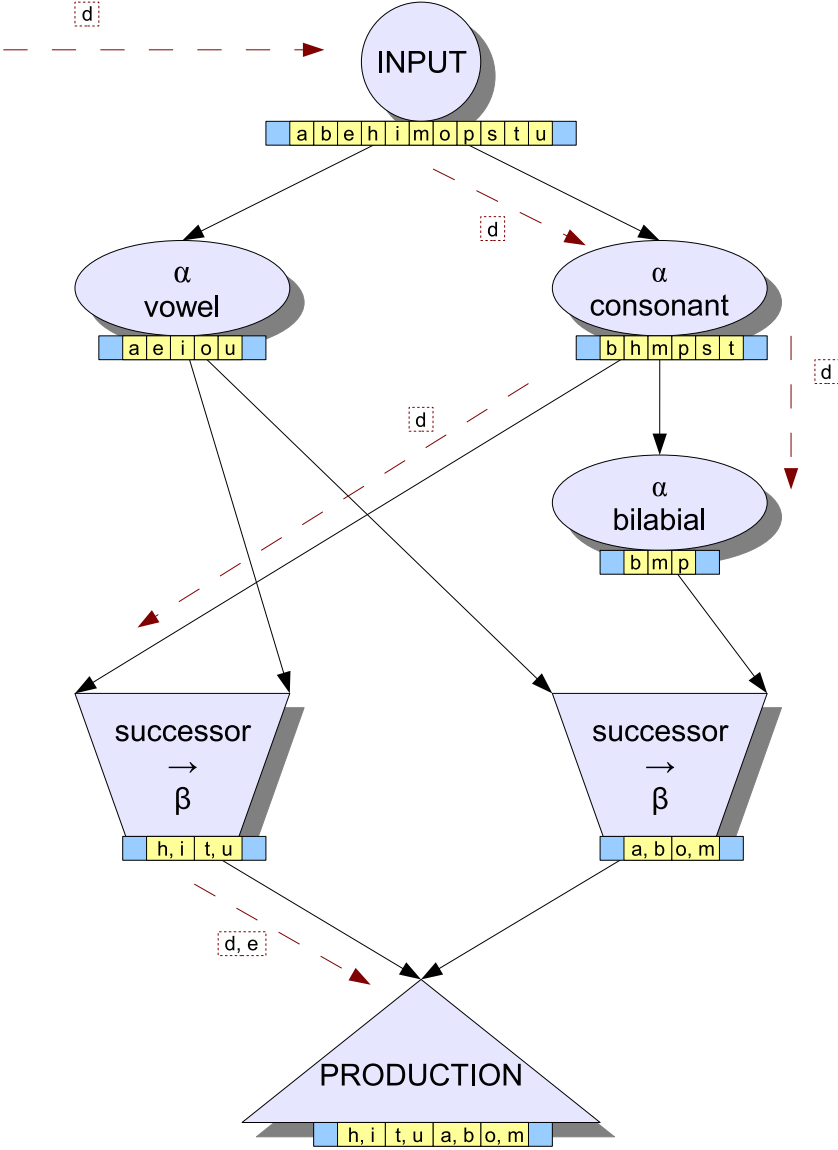


Figure 2.12: Consonants and vowels - a sample network illustrating the basic RETE concept

[16] presents a pattern matcher that builds and maintains Prolog-like resolution trees for evaluating patterns formulated as logical predicates. Both patterns and facts are asserted as logical clauses; facts are simple statements, while patterns are possibly multi-clause predicates. While the approach does not seem very efficient, this uniform treatment of facts and patterns gives rise to an interesting feature: the patterns can also be changed at runtime, resulting in incremental updates to the conflict set.

[3] is an introduction to the LEAPS algorithm, which is claimed by several sources to be substantially better than RETE or TREAT at both time and space complexity. The algorithm, however, is not easy to understand and downright difficult to implement; the cited paper introduces a powerful, general and advanced data structure definition framework before even beginning to explain the basics of LEAPS. Key features are using lazy evaluation to avoid unnecessarily manifesting tuples; looking up pattern occurrences newly satisfied by a new element and firing the corresponding rules in a depth-first-search fashion; applying time stamps on elements to achieve temporal constraints and handle deletion; keep all deleted elements with deletion timestamps to achieve 'time-travel' for the depth-first-search. LEAPS implementations, even in the field of rule based expert systems, are mostly in their experimental phase.

From all these alternatives, RETE was tried and well-known, flexible, easy to implement, and reliable enough to be chosen as the basis for our research. Experimenting with alternatives is still among future plans.

Chapter 3

Incremental pattern matching

3.1 Goal of this chapter

In order to provide an alternative way of pattern matching in the VIATRA2 framework, and also establish support for event driven transformations, we have developed an incremental pattern matching engine. This chapter describes our conceptional results in applying the RETE network principle in a VIATRA2 environment, our practical results in building RETE networks for the incremental matching of GTASM patterns, and our efforts at measuring the performance of the new pattern matcher.

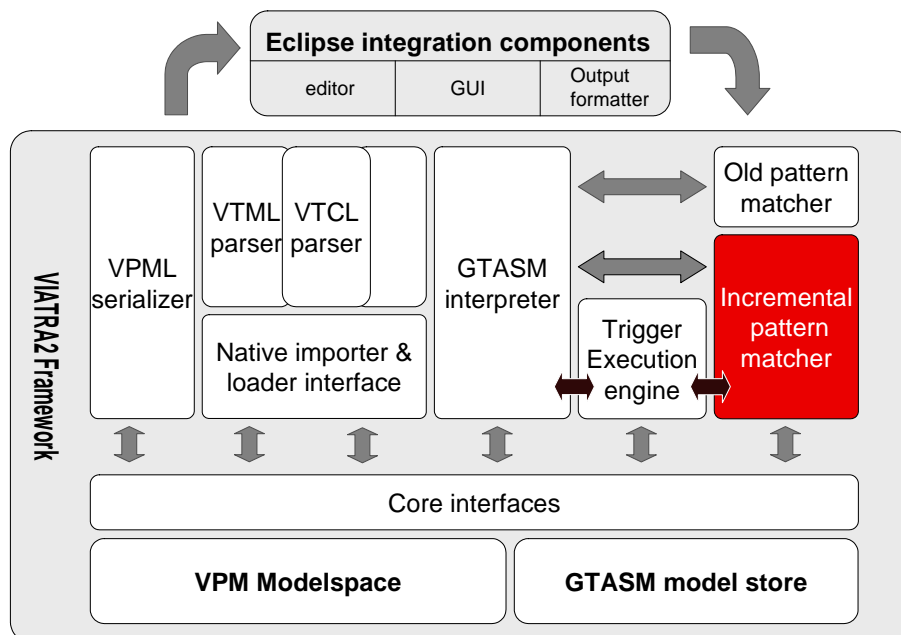


Figure 3.1: VIATRA2 R3 Framework extended with our Incremental Pattern Matcher

3.2 Applying the RETE concept on VPM modelspaces

We have introduced the basics of RETE networks in Section 2.3. The goal of this section is the application and adaptation of the theoretical concept as an incremental pattern matcher of the VIATRA2 framework.

3.2.1 Tuples

As the VIATRA2 modelspace contains relationships and properties of model elements, the contents of the RETE network must reflect this information. Therefore our RETE implementation employs *tuples* of model elements¹ as working memory elements. The graph-like semantics required to describe the contents of the VIATRA2 modelspace can be represented this way; the idea stems from [6] where a similar approach is used for graph pattern matching.

The general semantics of RETE nodes is that they store a set of tuples (regardless of whether their implementation actually includes a memory). Changes to this set are propagated along the edges starting from the node. This implicates that if an insert message containing a given tuple traverses an edge, an insert message containing the same tuple cannot be sent on the same edge again before an appropriate revoke message is sent. In other words, update messages should not be duplicated. This is the *uniqueness principle* of the RETE network, and there is one notable case where it is violated, see Section 3.2.3.

Quite often only a number of elements in a tuple are needed for some purpose, with their order also specified (and possibly different from their original order in the tuple). It is possible to regard this as a new tuple derived from the original one, much like the projection operation of relational algebra. It is important to notice that the derivation rules usually do not depend on the tuple itself, they are a property of the part of network under consideration. For usage in these scenarios as a description of derivation rules, we define the *pattern mask* as an array of indices. If a tuple ϕ is transformed by a mask μ , the result is a tuple ρ , called the *signature*, that has the same size as the mask, and at position i it contains the element that ϕ contains at the index specified by the i -th element of μ . Concisely: $\rho[i] = \phi[\mu[i]]$.

3.2.2 Inputs

Most properties of model elements, including relationships with other elements and even their existence, are subject to change. It follows that alpha nodes, employing a constant filter, are not suitable for checking conditions related them. Consequently these changes have to be input in the RETE network in the form of update tokens. Therefore all the RETE network knows of the model elements is their identity. No inner properties are used within the network²; all information regarding the modelspace is represented by tuples of model elements being present at certain nodes of the network.

The points where this information enters the system are the input nodes. The VIATRA2 modelspace broadcasts notifications of events such as creation or deletion of entities or relations. Input nodes receive update tokens by subscribing to these VIATRA2 modelspace notifications; some of them, however, might also have parent nodes in the network and thus receive update tokens from other sources.

Entity and relation roots

So-called *entity root* nodes convey the information that a certain entity belongs to a certain type. There is potentially an entity root for each entity type (however, for some types it will not be actually created, see on-demand construction); it contains tuples whose only element is the entity that is of this type. Similarly, *relation roots* belong to relation types, their tuples have three elements (the relation itself as a model element, source entity, destination entity).

¹actually, references to model elements

²with the exception of term evaluators, see 3.2.4

There are connections between these nodes: every entity or relation root has an incoming edge from root nodes representing its subtypes. This - for example - allows us to insert a new entity into the entity root responsible for the type specified for the entity; if there are child nodes (supertype roots), the update will be propagated and the tuple for the entity will appear in the other root node, from where it can be retrieved by queries on the supertype.

Containment roots

A single node called *direct containment root* is responsible for the containment hierarchy of the modelspace; this node contains pairs consisting of an entity and its container entity. However, it is convenient to mention that patterns will also need the transitive closure of direct containment, transitive containment. For this purpose, we define a *transitive containment root* as well, which is useful for building pattern matchers, while not an input node in the strict sense. When this node is constructed, a small network part is also created to fill and update it from the direct containment root. This network part is similar to what would be built by the pattern matcher builder (see 3.3) for a pattern describing the transitive closure of direct containment.

Though out of the scope of this paper, root nodes for the instance-of and supertype-of relationships and their respective transitive closures could also be created in a similar fashion should the need for generic transformations arise.

Problem with the uniqueness principle

Since entity and relation root nodes can have several parent nodes, they might receive the same update token multiple times, as illustrated by the following example.

If entity type A is the supertype of B and C, and D is a subtype of both B and C, then a newly created instance of D will induce updates in the root node of D, B, C and two updates in A. This is problematic, because if root nodes simply relayed updates to their child nodes, the uniqueness principle (see 3.2.1, page 35) would be violated. Therefore these root nodes are Uniqueness-EnforcerNode (see 3.2.3, page 41)s capable of suppressing duplicate updates. The problem is illustrated by figure 3.2.

On-demand construction and metamodel changes

Note that these nodes need not be created at startup. They will not be constructed until there is a need for them, i.e. they are accessed to be incorporated in a pattern matcher for the first time, or they are required by another root node (entity and relation roots by supertypes, containment root by transitive containment root), whichever comes first. When the need for a certain root node finally arises, the node is created and its contents are initialized; for transitive the containment root, the network section producing the contents have to be built for initialization and further updates; similarly, for entity or relation types, subtype roots have to be connected (created if they do not exist yet). This on-demand construction principle helps keep the RETE network as small (thus memory-saving and fast) as possible.

Changing the metamodel on-the-fly is possible (with some limits). Introducing new types bears no effect on the network until a matcher for a pattern referencing the new type is built - at that time, the root node of the type can be constructed on demand. If the type hierarchy is changed, creating and removing edges between root nodes will reflect these changes. To preserve consistency, the contents of the source node will be fed to the target node as positive updates if an edge is created, and as negative updates if the edge is removed. However, once a type root node is created, there is no established procedure to get rid of the type associated with it.

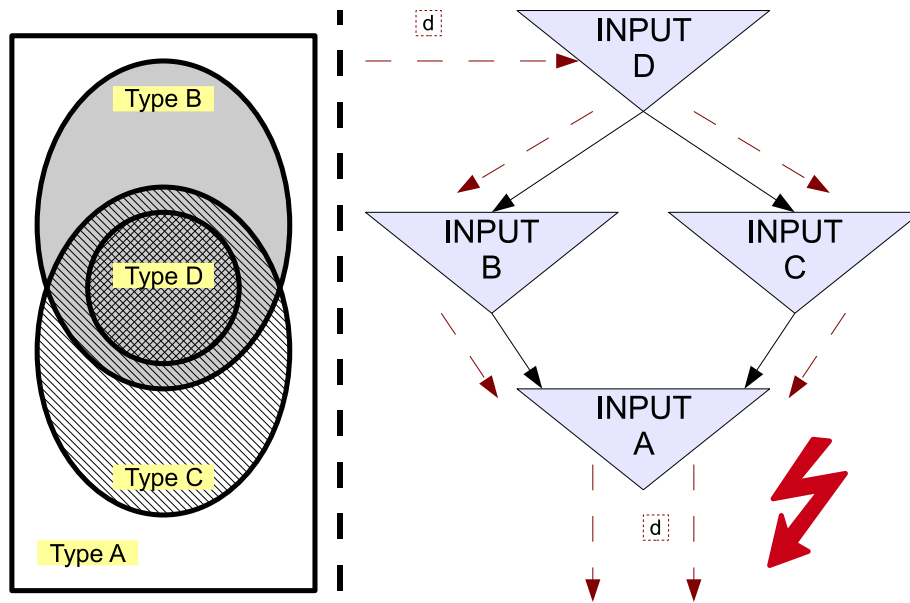


Figure 3.2: a problematic type hierarchy and the corresponding input nodes; the root node for type A must suppress duplicate updates

3.2.3 Nodes

One of the most important aspects of adapting the RETE concept in an environment is designing the common features and distinct types of RETE nodes used. Here an overview of building bricks of the pattern matcher network is given. Configurations built from them will be discussed in 3.3.3.

General features

Nodes are always built to contain tuples that conform to some subpattern, a subset of constraints imposed by a pattern body (or complete patterns in the case of production nodes). This set of constraints or partial pattern is the *semantical contents* of the node. This is in contrast to the term *actual contents*, which refers to the set of tuples actually contained by a node at a given time.

All nodes support receiving and possibly sending update messages along RETE edges. If the network was static, this would suffice; in reality, however, new parts of the network can be built for new patterns, so nodes must be able to accept new children. As explained in Section 3.2.2, changes to the metamodel can also cause the creation or deletion of edges. To meet these needs, all nodes are required to support a *pull operation* that queries their actual contents. For nodes without an internal memory, this operation usually involves querying parent nodes for their contents and repeating the process the node was built for. This is an example for sacrificing speed for memory saving, but the speed impact is not significant, since these operations (new pattern to match, change to metamodel) occur rarely compared to normal updates or pattern matching.

Special nodes

For fixing constant values in patterns, *ConstantNodes* are useful. They unalterably store a single tuple; they ignore all updates and send none. Their contents can only be retrieved via a pull

operation. See figure 3.3 for graphical notation.

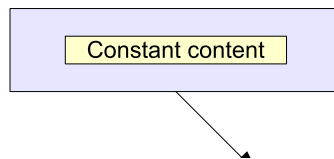


Figure 3.3: Graphical notation of ConstantNode

Indexers are special nodes associated with a pattern mask (see 3.2.1, page 35). The node contains an associative store of tuple memories, indexed by the signature of tuples according to the mask. Whenever an update is received, a signature is generated by transforming the tuple with the mask, the memory belonging to the signature is retrieved, and the tuple is inserted into / removed from the memory (in accordance with the nature of the update). If this was the first inserted / last deleted tuple with this signature, we say that it was an *important update*. For Indexers, the pull operation is performed by reading the contents of all the memories in the associative store. The benefit of this node is that it can retrieve tuples with given elements at the positions specified by the mask. It can also emit update messages with extra information concerning the signature of the pattern and whether the update was important. Indexers also double as input slots for beta nodes. See figure 3.4 for graphical notation.

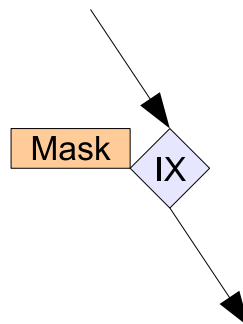


Figure 3.4: Graphical notation of Indexer

Dual input (beta) nodes

As an analogy to the beta node (see 2.3.2, page 30) of the original RETE concept, there are two kinds of nodes with dual input slots. They are connected as children to two Indexers, referred to as primary and secondary (or left-hand and right-hand) slots. They take advantage of the extended update messages of Indexers, and may use the included signature to look up tuples of that signature from their other parent Indexer. They do not contain a memory. For the ease of discussion, the grandparents can be defined as the parents of the input slots. The masks of the Indexers must be of the same size, because the signatures produced by the two slots are expected to have the same semantics.

The first kind of beta node is the *JoinNode*, that basically calculates the natural join of the contents of its parents. This is probably the single most important element of the RETE network; most related work contains a node with similar functionality, sometimes referred to as an AND node (as it enforces that two conditions must be met). As with the rules of natural join, the contents of this node are tuples that combine two tuples (one from each input slot) whose signature matches (each signature generated by the mask of the appropriate slot). The combined tuple contains all elements of the tuples it was created from; but includes only one instance of those elements that were selected by the pattern masks and matched to be equal on both sides. Whenever an update arrives from one of the input slots, tuples with the same signature are retrieved from the other Indexer. Then each of them is combined with the incoming tuple and the result is propagated to the children of the *JoinNode*. See Figure 3.5 for graphical notation.

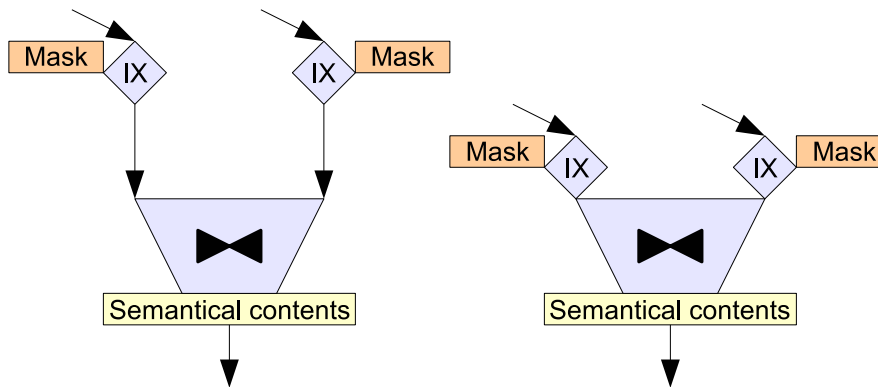


Figure 3.5: Graphical notation of *JoinNode* with Indexers as slots - concise form on the right

The other kind of dual input nodes is the *NotNode*, that filters all tuples from its primary input slot that do NOT have a matching tuple on the secondary side. Not every work mentions a node with a similar role; some of those that do, refer to it as a NAND node (although the semantics are closer to the Boole expression $\alpha \wedge \neg\beta$). If an update is received from the primary slot, the tuples with the same signature are looked up from the secondary slot; the update is propagated on outgoing edges if and only if there are no matching tuples in the secondary slot. If a positive update is received at the secondary slot, no action is taken unless it was an important update, in which case the set of tuples with the same signature is retrieved from the primary node and every one of them is propagated as a negative update message. Receiving negative updates at the secondary slot involves a similar procedure, but this time positive updates will be propagated. See figure 3.6 for graphical notation.

Single input filtering and transformation nodes

The *EqualityNode* and *InequalityNode* are types of alpha node (see 2.3.2, page 30). They check whether certain elements in the tuple, selected by a pattern mask associated with the node, are all equal (*EqualityNode*) or all different from a subject element specified by its index (*InequalityNode*). These nodes propagate updates that match these criteria and ignore those that does not. They have no internal memory; the pull operation is performed by pulling the contents of parent nodes and filtering them. The roles of these nodes will become apparent when they are put to use in 3.3.3. See figures 3.7 and 3.8 for graphical notation.

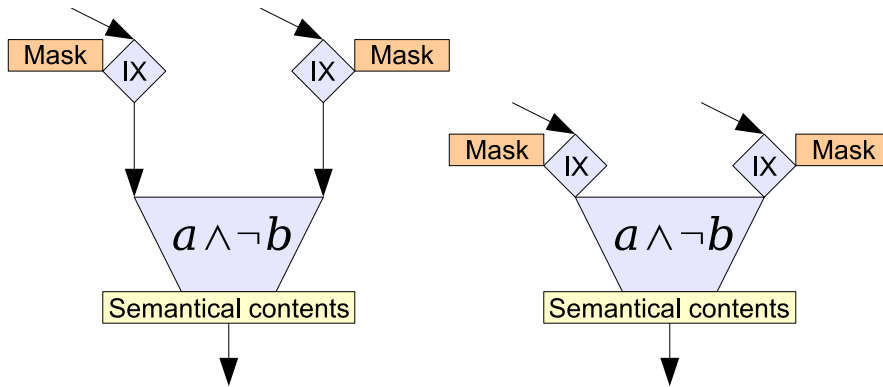


Figure 3.6: Graphical notation of NotNode with Indexers as slots - concise form on the right

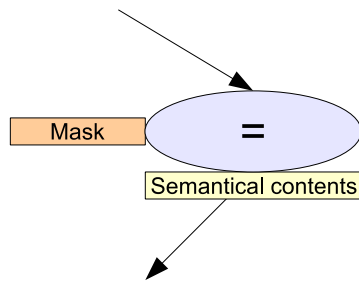


Figure 3.7: Graphical notation of EqualityNode

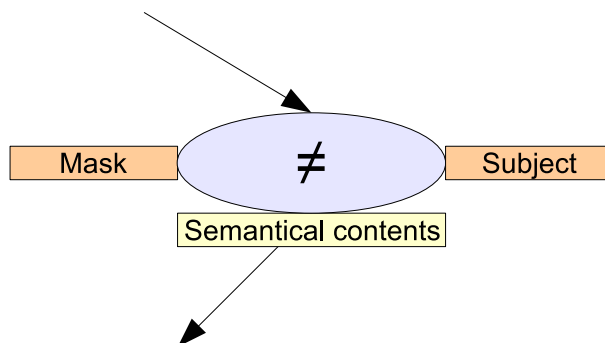


Figure 3.8: Graphical notation of InequalityNode

The *TrimmerNode* has a pattern mask and outputs the contents of its parent transformed by the mask. On receiving an update, it uses the mask to transform the tuple contained in the update and propagate the result as an update. It does not have an internal memory; the pull operation is performed by pulling the parents and transforming their contents again. It is important to note that several tuples can have the same signature, so even when receiving updates containing different tuples, the updates sent by this node can contain the same tuple, thus violating the uniqueness principle (see 3.2.1, page 35). See figure 3.9 for graphical notation.

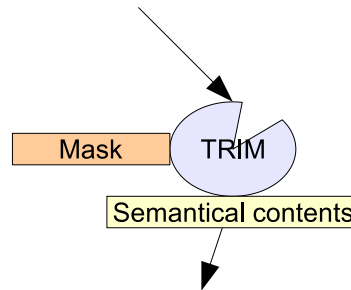


Figure 3.9: Graphical notation of TrimmerNode

If a node is the child of several parent nodes, or the child of a TrimmerNode, it cannot rely on the fact that updates received will be unique. *UniquenessEnforcerNode* has a memory that works like a multi-set (also known as bag) and enforces the uniqueness principle (see 3.2.1). Upon receiving a positive update, the tuple is added to the memory; the update is propagated if and only if the tuple was not present in the memory prior to the reception. Upon receiving a negative update, the tuple is removed from the memory; the condition for propagation tests whether the removed instance was the last one of that tuple in the memory. Pull operations are served from the memory.

Since root nodes can have multiple parents, they are UniquenessEnforcerNodes. Having multiple TrimmerNode parents (for an explanation, see 3.3.2), the production nodes have to be of this type as well. See figure 3.10 for graphical notation.

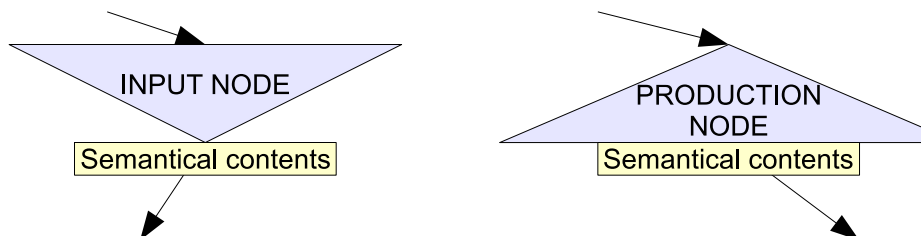


Figure 3.10: Graphical notation of root nodes; input node on the left, production node on the right

3.2.4 Arbitrary term evaluation

There is one other type of node called *TermEvaluatorNode* that has not been mentioned yet. It deserves special mention because it diverges significantly from the classic RETE concept. It eval-

uates a GTASM expression on tuples and filters those tuples for which it evaluates to true. It is similar to an alpha node, with one key difference: the filtering condition is not required to be constant. The filtering condition is an arbitrary GTASM term, it is considered as a black box. Its value may depend on internal properties of VIATRA2 model elements like name, value, location in the modelspace containment hierarchy; or the values of referenced global permanent associative storages (ASM functions, see 2.1.3 on page 19). See figure 3.11 for graphical notation.

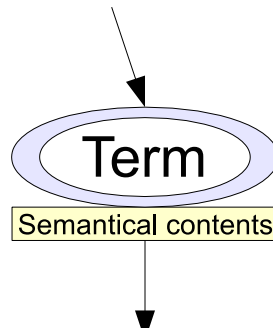


Figure 3.11: Graphical notation of TermEvaluatorNode

The original concept of alpha nodes required the filtering condition to be constant (to yield the same result on a given tuple whenever it is evaluated). However, the value of GTASM terms may change over time; a tuple that was ignored when it was inserted may pass the test now, while another one that originally passed the test and was propagated may have to be revoked now. If the TermEvaluatorNode can be notified each time the result potentially changes, the term can be re-evaluated on the affected tuples and the previous decision can be revisited, the appropriate updates can be sent.

With some reasonable assumptions, such a notification can be provided. The required assumptions are:

- The value of the expression can only change when one of the model elements or ASM functions experience change. External factors can only be taken into account if they are constant, e.g. the current datetime should not factor in a term. This is a reasonable assumption given the nature and goals of pattern matching. Note that external factors can still be used in the action part of a GTASM rule, but not in the precondition pattern.
- All model elements whose change may induce a change in the value of the expression have to be in the footprint (see 3.3.2, page 45) of the expression. The footprint of a GTASM term consists of variables referenced in the expression as arguments of function invocations (value comparisons, builtin arithmetic functions, arbitrary native functions, etc.). If the value depends on any other model element, it should be kept constant (e.g. constant values in the expression, references to metamodel elements, etc). This assumption is reasonable, as expressions looking up model elements outside their footprint are considered bad practice, those model elements should be involved in the graph pattern and appear as pattern variables included in the footprint of the expression.
- Likewise, all ASM functions whose change may induce a change in the value of the expression have to be directly called by the expression, not indirectly through native functions.

- The model elements in the footprint can only influence the expression with their name, value, or location. The value of the expression can only change when one of them is renamed, assigned a new value or moved. This is a reasonable assumption considering the features of the VPM modelspace and considering that relationships of model elements should be handled by the graph pattern, not the GTASM terms.

The `TermEvaluatorNode` is associated with a GTASM term, a mapping between variables in the term and corresponding positions in a tuple, and information regarding how the result of the term evaluation should be interpreted. Either the term should evaluate to true, or it should evaluate to an element in the tuple specified by index. Apart from the above four assumptions, the GTASM expression is treated as a black box, the `TermEvaluatorNode` does not depend on the structure of the term. So by providing arbitrarily implemented native functions, the set of available expressions can be extended to meet the needs of the pattern.

The `TermEvaluatorNode` has a memory that stores the tuples that passed the filter. The pull operation is accomplished by returning the value of the memory. Whenever a positive update is received, the term is evaluated on the tuple; if the check is successful, the tuple is inserted into the memory and the update is propagated along the outgoing edges. Whenever a negative update is received, the memory is checked; if it does not contain the tuple, it did not pass the filter, so no action is required; if the tuple is present in the memory, then it is removed and the negative update propagated on outgoing edges.

Furthermore, whenever a new positive update is received, the node subscribes to notifications of changes to those elements in the tuple that correspond to the footprint of the expression, and of changes to the ASM functions invoked in the expression. Whenever a negative update is received, the node unsubscribes from the very same notification services. These notifications are administered by a single object listening for VIATRA2 modelspace change notifications.

The node is notified whenever the value of an invoked ASM function is changed, or an element in its subscription is renamed, assigned a new value or moved. In these cases, the node re-evaluates the expression for every tuple that is influenced by the changed model element / ASM function. If the tuple now passes the filter but did not pass it before (it is not present in the memory), it is inserted in the memory and propagated as a positive update. If the tuple does not pass the filter but is present in the memory, it is removed and propagated as a negative update.

With this node, it is possible to include arbitrary check expressions in the pattern (with the above restrictions), to enhance the expressivity without sacrificing incrementality. Terms can also appear in containment constraints and as arguments of pattern calls; in these cases, it evaluates to a model element. For dealing with the latter cases, *term-substitute* variables are introduced. A pattern call

```
find pattern1(A, B, someterm, C);
```

is substituted with

```
find pattern1(A, B, X, C);  
X = someterm;
```

Containment constraints are processed in a similar manner.

Note that in theory, the `TermEvaluatorNode` could be enhanced to evaluate the term and append the results to the end of the pattern. This is a planned future improvement that would eliminate the problem with terms as parameters of negative pattern calls, explained in 3.3.3.

3.2.5 Applications in pattern matching

All pattern matching operations require access to the production node. If the pattern matcher has not yet been built, it must be constructed according to the process that will be introduced in 3.3.

If the task is to retrieve all occurrences of a pattern, one simply needs to access the memory of the production node and cycle through the tuples stored there. Tuples will contain substitutions for each of the symbolic parameters, in order. The result set can be filtered with specifying containment scopes for the parameters.

In a more common case, one needs to find all occurrences of the pattern where a number of parameters have a fixed value. This operation can be performed efficiently by preparing an Indexer node attached to the ProductionNode, that indexes tuples according to a mask, which contains the indices of fixed parameters. The fixed parameters can be treated as a signature (see 3.2.1, page 35), the memory corresponding to that signature can be retrieved from the Indexer, and result set is contained in that memory.

A special case to the former: verifying whether a given tuple satisfies the pattern. An Indexer should be prepared with a mask containing all indices; as all parameters are fixed, the given tuple is the signature itself; the answer can be given by checking whether the corresponding memory is empty.

The incremental approach has several advantages here: not only can it start to enumerate the result set virtually instantly, it can also count the cardinality of the set in constant time, without enumerating it. Because production nodes and appended Indexers have memories that store the result set in an efficient data structure, the number of results can be found instantly.

There is another possible application of the RETE network: if one registers an object as a child node to the production node, one will receive notifications whenever a new occurrence of the pattern is found or an old one is lost. This mechanism will be the basis of event-driven model transformations, see 4.

3.3 Building a RETE net from GTASM patterns

We propose a simple yet versatile algorithm for the construction of RETE networks capable of matching GTASM patterns. The algorithm perceives the pattern as a set of constraints imposed on a set of variables. It generates a line of RETE nodes that progressively assert more and more of those constraints until a production node is finally built which will have the set of occurrences of the pattern as its semantical contents.

3.3.1 GTASM patterns as constraint systems

The following paragraphs revisit the concept of the VIATRA2 graph pattern (see Section 2.1.3 on page 15) from the perspective of the incremental pattern matcher.

Each pattern defines a logical relation on a sequence of symbolic parameters. This relation is expressed as the disjunction of logical relations defined by the bodies of the pattern. Each pattern body itself defines a logical relation on a set of variables that include the symbolic parameters (the rest are local variables), and projecting this logical relation onto the symbolic parameters yields a component that, in disjunction with projected relations of the other pattern bodies, defines the logical relation of the pattern. The pattern body defines this logical relation with a set of conjunctive constraints on the acceptable combinations of variables.

Various types of constraints can be asserted by a pattern body (most of them are also meaningful when constants or arbitrary terms are used instead of variables). The most important ones are asserting that a variable be an entity of a given type, and asserting that a variable be a relation of a given type and connect to variables as source and target. There are also containment constraints between variables, both direct containment ("in") and transitive ("below"). For generic patterns

(which are out of the scope of this paper), there can be instance-of relationships (i.e. variable-typed variables), as well as supertype-of-relationships.

With a special constraint type called *variable assignment*, the value of variables can be assigned to other variables, e.g. $X=Y$.

Another important constraint type is the (positive) pattern call, which asserts that a given pattern is valid with a specified sequence of variables as its parameters. This feature is complemented by the negative pattern call, that constrains a sequence of variables not to be valid parameters for a pattern. Finally, any boolean-valued GTASM term can be used to constrain the variables as a check expression, thus providing a blackbox-like check feature.

Apart from the constraints discussed above, there are also implicit injectivity constraints that are associated with pattern bodies even without explicitly asserting them. Injectivity principles dictate that all variables of a pattern body (regardless of being local or a symbolic parameter) should take different values, with some exceptions (the most notable being $X=Y$ explicitly stated).

3.3.2 The construction algorithm

Here is the algorithm that was employed in our implementation to create (or, later, extend) the RETE network for the recognition a given pattern.

As the pattern is simply a disjunction of bodies, the difficult part is the algorithm for building a matcher for a pattern body. It basically keeps track of some auxiliary values during construction. One of them is the *current stub* Σ , which is the last node built for the current pattern body. The *current variable tuple* Γ defines the variables the output of Σ refers to, thus the tuple of pattern variables and the tuple of their respective substituted values are handled in an analogous way. The *constraint pool* Π consists of the constraints that are yet to be enforced. We also refer to elements of Γ as bound variables. We define the *footprint*³ $F(\phi)$ of a constraint ϕ as the set of variables referenced by ϕ .

The variables involved in this algorithm include not only symbolic parameters and local variables, but also *virtual variables* that are introduced for one of two reasons. First, all constant values referenced in the pattern body are treated as variables that are bound from the very beginning. Additionally, when using GTASM terms in containment constraints or as pattern call arguments, the transformation described in 3.2.4 is applied to substitute them with virtual variables.

Variable assignments (see 3.3.1) deserve special attention. A Union-Find algorithm (see [7] for an introduction) is executed on the variables, with variable assignments as unifications, to determine the equivalence classes of variables that are asserted to equal each other. From now on, the term 'variable' will actually refer to unified variable classes.

Recognition of a pattern body

1. Initialize the constraint pool Π with the set of constraints defining the body; initialize the current variable tuple Γ with a (possibly empty/nullary) tuple formed by the constants (as virtual variables) referenced in the pattern body; initialize Σ as a new ConstantNode with the tuple of values of the constants in Γ as its contents.
2. If Π is empty, the semantical contents of Σ will now satisfy the constraints of the pattern body. Connect a TrimmerNode to Σ that trims Γ into the required symbolic parameter tuple, mark it as the body terminator node and stop.
3. If Π is not empty, select from it a constraint α for which a checker will be built.

³This definition is the generalisation of the sense the word 'footprint' is used in Section 3.2.4

4. $\Gamma' := \Gamma \cup F(\alpha)$, i.e. extend the current partial pattern tuple to accommodate new variables needed for expressing α .
5. Construct a new Σ' (practically a descendant of Σ) whose semantical content conforms to the new Γ' and enforces α in addition to all constraints enforced by Σ .
6. Go to step 2 with Σ' substituted for Σ , Γ' substituted for Γ and $\Pi \setminus \{\alpha\}$ substituted for Π .

As we see, the algorithm will stop when all constraints are enforced; and since each variable must be affected by at least one constraint (this is a reasonable assumption and is actually enforced by the VTCL language), they will all be bound at the end. Therefore the final value of Γ will contain the symbolic parameters as well, so there exists a mask that maps Γ into the tuple of symbolic parameters (in the correct order). This proves that building the body terminator in step 2 is always possible.

When all pattern bodies are ready, a single production node responsible for the recognition of the pattern is built as a child of every single body terminator. This node has to be a Uniqueness-EnforcerNode and maintain a memory to reject duplicate updates for two reasons: first, as the child of possibly multiple nodes, it might receive the same update (i.e. finding the same occurrence of the pattern) from different pattern bodies; additionally, due to the nature of TrimmerNodes (see 3.2.3), the body terminators themselves emit duplicate update messages.

3.3.3 Employed node configurations

This section introduces the node configurations built to handle pattern body constraints

The start of the chain

The initialization of Σ in step 1 involves creating a ConstantNode (see 3.2.3, page 37) that contains a tuple composed of the constant values in the pattern. Often there are no constants in the pattern (or only on the metamodel level, where they will specify root nodes), which means that this ConstantNode contains an empty tuple. This stub is the first in a string of nodes onto which all further checkers will be sequentially appended.

JoinNode-based configurations

Entity checkers, relation checkers, containment constraint checkers and positive pattern call checkers built in step 5 all conform to the same basic structure. This structure introduces new information to the main string of nodes from an external source: a root node. The checker employs a JoinNode (see 3.2.3, page 39) to perform a natural join operation on the contents of the stub and the contents of the appropriate entity root, relation root, containment root, production node (if necessary, root nodes are to be created; for production nodes, a pattern matcher has to be built following this procedure). The JoinNode needs an Indexer (see 3.2.3, page 38) at each of its slots. The primary slot should be equipped with a pattern mask that selects the common variables of the semantical contents of the stub and the root node. The secondary slot should be equipped with another mask that selects the same variables (from a different tuple of pattern variables).

If variables are explicitly assigned to each other ($X=Y$), they are treated as the same variable. Consequently, if some of the variables in the footprint of the constraint are already bound, the join node will enforce their equality. The only task left is to check if the checker introduces new variables to the stub, and whether some of the new variables are supposed to equal each other (but not the old variables). In this case, optional EqualityNode (see 3.2.3, page 39) instances are

applied to the stub to enforce that a set of new variables are all equal. One such node is built for every equivalence class of new variables with no equal old variable and ore than one instance among new variables. An example where such a node is needed, if the constraints are checked in this order:

```
someEntityType (A);
find some_pattern(A, B, B);
```

Without going into exact details of the injectivity rules, an optional InequalityNode (see 3.2.3, page 39) for each new variable is usually appended to the stub at this point, in order to enforce that new variables are different from certain other variables.

Figure 3.12 illustrates this configuration with one EqualityNode, one InequalityNode and an input node as the root; the configuration would remain the same with a production node instead of the input node.

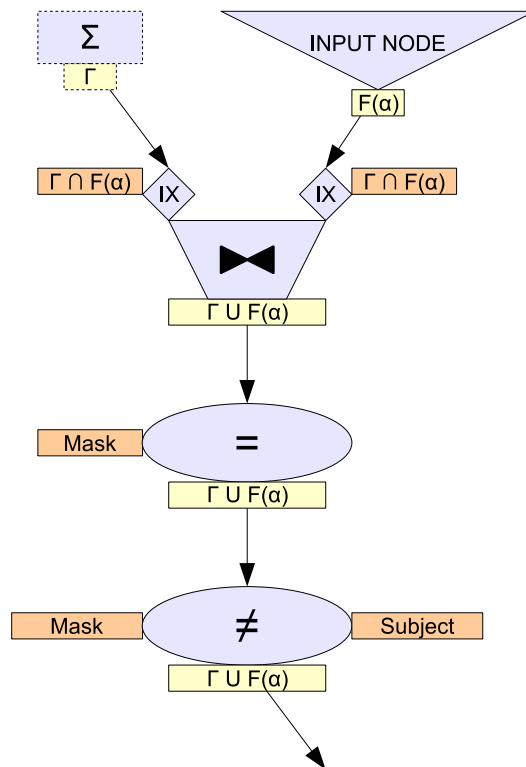


Figure 3.12: Node configuration for checking entity type, relation type, containment constraints and positive pattern calls

Negative condition checking and GTASM term evaluations

Negative pattern calls are checked in a way that has similarities with the positive pattern checking. A NotNode (see 3.2.3, page 39)NotNode is built; its primary slot is appended to the stub and the secondary to the production node of the pattern; the masks of the slots are set up the same way as with the JoinNode. The nature of the NotNode dictates that no new variables are introduced, which

conforms to the semantics of negative pattern calls. This has several consequences: no EqualityNodes and InequalityNodes are needed; the negative pattern, however, can only be checked when all elements in the footprint are already bound. Figure 3.13 illustrates this configuration.

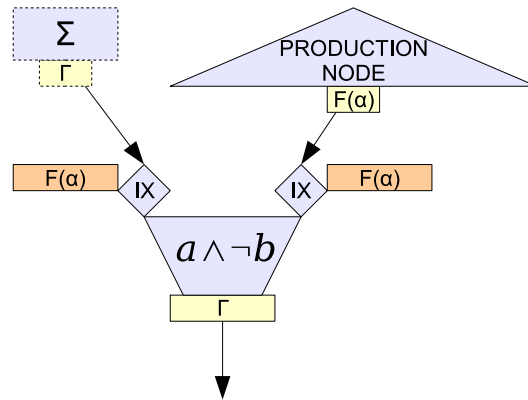


Figure 3.13: Node configuration for checking negative pattern calls

Checking GTASM terms is as simple as appending a TermEvaluatorNode (see 3.2.4, page 41) to the stub, and setting it up with the GTASM term and a mapping between variables in the term and their respective positions in the tuples. As with negative patterns, this checker does not introduce new variables; no EqualityNodes and InequalityNodes are needed; all elements in the footprint are to be bound when the checker is applied. Figure 3.14 illustrates this configuration.

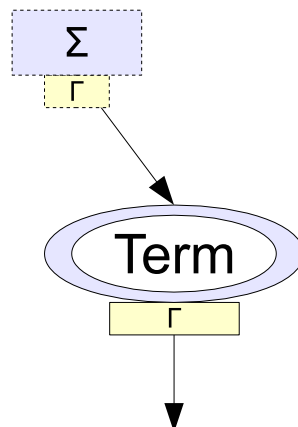


Figure 3.14: Node configuration for checking GTASM terms

The checkers described above have a limitation; its importance is dubious, it could even be called obscure, as the authors have not yet encountered a situation where it would matter; but it

should nevertheless be documented and directions towards a solution given. Both negative pattern calls and term evaluators need their complete footprint bound. Most variables are forced to have a type declaration, so they have at least one pattern whose checker can introduce this variable before any associated negative pattern calls or term evaluators are built. However, term-substitute (see 3.2.4, page 43) variables “sandwiched” between these two types of constraints pose an exception. If a GTASM term is used as an argument of a negative pattern call, the term-substitute virtual variable representing the value of the term will only be constrained by the negative pattern call and the check expression generated with the term substitution. It can only be bound by checking one of those constraints, but neither can be checked before the variable is bound, resulting in a deadlock. The solution would be to augment the `TermEvaluatorNode` with the ability to extend the incoming tuples with the result of the term, in which case they could be used to bind that variable.

Pattern body termination

The pattern bodies are terminated by a `TrimmerNode` (see 3.2.3, page 41), equipped with a mask that selects the variables that are symbolic parameters, and ordering them as required. The production node of the pattern is a `UniquenessEnforcerNode` (see 3.2.3, page 41) that is a child of the terminating `TrimmerNodes` of the pattern bodies.

3.3.4 Constraint ordering

We employed a heuristical method to define the order in which constraints are to be processed; it determines which constraint should be selected from the constraint pool in step 3.

The applied selection criteria, in the order of descending priority:

1. The most important condition of selecting a given constraint is the feasibility of checking it at this point of the RETE net under construction. Some constraints (negative patterns and `check()` expressions, see 3.3.3) cannot be enforced until all the variables in the footprint of the constraint have been bound. Constraints of these types are considered unsafe and must not be selected, unless their footprint becomes completely bound.
2. Constraints with a completely bound footprint are preferred over other constraints. This choice is justified by the consideration that these constraints do not introduce new variables into the current tuple Γ , they only check properties of the existing tuple. Therefore, they do not contribute to the combinatorial escalation of the content size of the stub Σ ; on the contrary, by imposing additional conditions on the contents, they narrow the set. Selecting and checking these constraints as early as possible helps keeping the number of tuples loaded into the RETE network low. Smaller node content size means lower memory consumption and faster updates.
3. Constraints with more bound variables are preferred. In addition to the reasons stated above, having more bound variables suggests that fewer tuples conform to both the constraint and the bound variables. This means that fewer tuples will be present on the secondary side of the `JoinNode`, resulting in faster node operation and presumably fewer products, once again easing the load on the RETE net. Even among constraints with a completely bound footprint, those with more (bound) variables are intuitively assumed to have a lower probability of being satisfied by a fixed tuple.

4. If the number of bound variables match, the constraint with the lower number of unbound variables in its footprint is preferred. Fewer unbound variables bear the promise of fewer combinatorial possibilities to satisfy the constraint, and thus fewer results.
5. Miscellaneous tie-breaking.

3.3.5 Example pattern matcher

The process of generating a RETE network from a pattern definition is now illustrated.

If one is looking for vegetarians whose boss is the father of a boy scout, the following patterns will find the boys and subordinates in question. It will sometimes be called with the *Boy* as a fixed input parameter.

```

pattern boyScoutSonOfBoss (Subordinate, Boy) =
{
    metamodel.boyscout (Boy);
    metamodel.person.child (R, Father, Boy);
    metamodel.male (Father);
    find superior (Subordinate, Father);
    metamodel.vegetarian (Subordinate);
}
pattern superior (Worker, Boss) =
{
    metamodel.employee (Worker);
    metamodel.employee.department (R1, Worker, Department);
    metamodel.companydepartment (Department);
    metamodel.companydepartment.head (R2, Department, Boss);
    metamodel.employee (Boss);
}

```

It is easy to see that defining a separate *superior* pattern makes it easier to define the complicated *boyScoutSonOfBoss* pattern; the pattern matcher network will also be easier to show.

The algorithm introduced in this chapter will build a matcher network similar to the one illustrated on figure 3.15. For brevity, we omitted the (now empty) ConstantNode that should have been created in step 1, and started with the first entity checker instead; we also omitted the InequalityNodes that are supposed to follow the JoinNodes. When the pattern matcher is executed with a fixed boy for the first time, another Indexer is appended to the production node that makes it easy to retrieve subordinates belonging to a given boy; this Indexer is shown on the illustration as well.

3.4 Performance

In this section, we discuss various issues and ways of improvement regarding to the RETE network; apply synthetical benchmarks to find the general advantages and disadvantages of the RETE network versus the traditional pattern matcher; we also conduct experiments on a practical model transformation problem to measure the efficiency of implemented optimizations and compare the incremental matcher to the traditional VIATRA2 pattern matcher.

3.4.1 Improvements and optimizations

There are many ways to improve the performance of the pattern matcher component described in the previous sections.

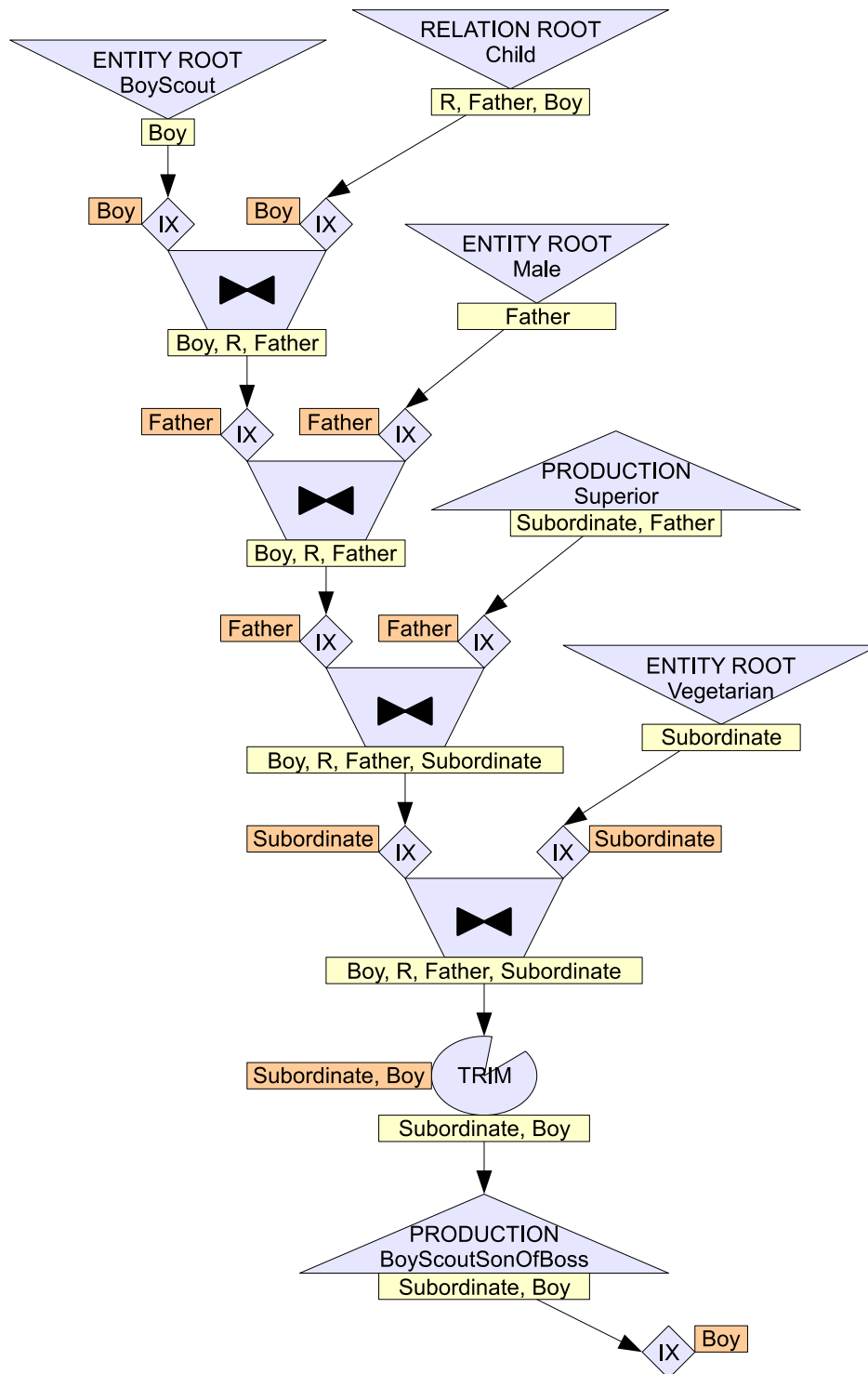


Figure 3.15: A RETE pattern matcher for the example pattern in 3.3.5; omitted details: the leading ConstantNode and InequalityNodes

RETE node sharing

A single RETE network may recognise several patterns. A classical RETE optimization technique, *node sharing* aims to fight redundancy and save network size (and thus space and time complexity) by sharing some nodes between patterns. An extreme case would be when two patterns have the same definition; in this case, no new nodes have to be built for the second pattern, it may simply reuse the production node of the first. Exploiting this benefit, however, is actually quite difficult. Identifying large similar subpatterns that could be shared between patterns is an interesting data mining problem, related to the field of graph-based substructure pattern mining; while there are algorithms elaborated for this problem (see [45]), they are out of the scope of this paper. The designer of patterns, on the other hand, can extract frequently used features into separate patterns, and call that pattern from other patterns; in this case, the network part for the called pattern is automatically shared between called patterns, as the production node is joined into them.

Indexer sharing

However, there is a special case for which we did implement node sharing. Due to the nature of our adaptation of the beta node and its slots, many Indexer (see 3.2.3, page 38) nodes are built, even with the same mask and attached to the same parent node, especially as children of root nodes. For example, many patterns may use a certain relation type indexed by its source element. It is also important to remember that Indexers have their own tuple memory, consuming valuable space. For these reasons, the RETE network has the ability to reuse Indexers that are attached to the same node and operate according to the same pattern mask. Our future plans include experimenting with sharing other types of nodes.

Tuple inheritance

If patterns have a large number of local variables, tuples can grow big. Some of the storage consumed by them is redundant, since most tuples are the result of a JoinNode (see 3.2.3, page 39) combining two tuples from the primary and the secondary side, and they store elements already present at other tuples. Space can be conserved if the result tuple simply references the primary tuple that spawned it, as the result tuple contains the entire primary tuple; only elements stemming from the secondary tuple, need to be stored at the result tuple. This is *tuple inheritance*, an idea borrowed from [43] (see Section 2.3.5, page 31). When JoinNodes combine two tuples, they create a new tuple that inherits the contents of the primary tuple, and extend it by elements from the secondary tuple that correspond to variables not included in the primary tuple. This solution sacrifices the speed of accessing elements (an ancestry tree of tuples might have to be traversed), but might save memory for large tuples constructed in several steps (the contents of the parent tuple need not be stored more than once). We have implemented this feature and conducted measurements to determine whether it would have any significant impact on performance for model transformation problems. Although not implemented, this principle can be further extended to two-way inheritance: tuples produced by JoinNodes do not contain the elements from the secondary tuple either, just a reference to the secondary tuple and the reference to a pattern mask describing the mapping from the secondary tuple to the rest of the positions of the result tuple.

Type inferencing

The current RETE builder algorithm presented in Section 3.3 can be considered naive for a number of reasons. It checks some constraints redundantly; for example, if the target of a relation type is declared to be of type A, and in a pattern a variable has a type declaration B where B is a superset of

A, then checking if the variable is the target of the relation eliminates the need for type-checking it. Type inferencing can save some type checks, which results in a smaller and faster pattern matcher network. And domain/range type is not the only information that is not put to use yet: potentially powerful constraint ordering heuristics could be based on multiplicity declarations.

Network shaping

The flexibility of the RETE concept allows an arbitrary shape for the network, which is a feature that most alternatives do not have. The current builder constructs a linear chain of nodes for each pattern. An other possible extreme would be a network shaped like a balanced binary tree. The middle ground could be checking type and containment constraints for each variable in a separate branch. The benefit of a wider, but less deep network would be that updates have to travel a shorter path, resulting in less time-consuming updates, a lighter overhead. On the other hand, the more constrained a subpattern is, the less occurrences it is expected to have, so having nodes at greater depths could result in fewer tuples.

3.4.2 Benchmark: Sierpinski triangles

To compare the efficiency of the incremental and the traditional pattern matcher, the Sierpinski triangles (see [14]) benchmark was adopted from the AGTIVE 2007 graph transformation tool contest. To perform this benchmark, we used VIATRA2 transformations, that were created by Máté Kovács.

The task is to build a fixed number of generations of the Sierpinski triangle, a graph derived from the popular fractal structure. The transformation initializes with a triangle of nodes and edges. Then, in each generation, the triangles consisting the graph are replaced with a 6-node structure forming 4 new triangles (one of which is excluded from further expanding). Each generation yields a graph that is approximately 3 times the size of the previous one.

We used two separate implementations of this algorithm. The *forall* version uses the *forall* construct of the VTCL language; for each generation, first it searches for all occurrences of the triangle pattern, then it replaces each occurrence with the 6-node structure discussed earlier. The *iterate* version uses the *iterate choose* construct of the VTCL language; for each generation, it repeatedly searches for an “old” occurrence of the triangle pattern to replace it with “new” edges; they will be marked “old” by the end of the generation. It is easy to see that the two versions have different characteristics; the *iterate* version calls the pattern matcher much more frequently and additionally keeps track of “old” and “new” triangles.

The transformation execution time was measured with the traditional pattern matcher and the (optimized) incremental pattern matcher on both version of the six-generation Sierpinski triangles transformation; for each of them, 10 experiments were performed. The average execution times (and corresponding standard deviations) are displayed on Table 3.1.

runtime(ms)	iterate	forall
traditional	11 856 (\pm 566)	997 (\pm 146)
incremental	2 410 (\pm 188)	1 793 (\pm 315)

Table 3.1: Average execution times (and standard deviation) in milliseconds for the Sierpinski triangles benchmark with six generations

Since the *iterate* version transformation is more complex, it is not surprising that it takes longer than the *forall* version. For the *forall* version, pattern matching calls are issued only once every generation, and practically the entire model is replaced between calls; this implies that incremental

approach has virtually no benefits, and the overhead of continuously updating the RETE network makes the incremental pattern matcher slower than the traditional one. The iterate version, however, issues pattern matching calls much more frequently, with little change to the model between these calls; this implies that the incremental approach has a great advantage over the traditional pattern matcher.

3.4.3 Measurements with a practical application

The behaviour of the pattern matcher depends heavily on the properties of the patterns and the contents of the model. For different problem fields, different methods may prove to be more efficient; the profile of model transformation problems is not obvious. While synthetic benchmarks were already provided, in this section the performance is measured on a practical problem.

The measurement was conducted with a model transformation designed and implemented by Bálint Kasza. The source model is a platform independent model describing a graphical user interface (UiPIM; see [8]), constructed in a graphical editor. The target model is a platform specific model describing a HTML/AJAX implementation of the designed user interface. Note that this is a batch transformation designed with the original pattern matcher in mind.

A general observation was that the first and second runs of the transformation machine produced different results; from the third run onwards, the results did not change significantly. In the case of the traditional pattern matcher, this discrepancy is due to search plans being preserved from the first run to the second. In the case of the incremental pattern matcher, there is a more substantial difference, since the RETE network was constructed and populated in the first run; in later runs, pattern matching can be done virtually instantly as the RETE network is ready, so the time cost reflects the execution time of the transformation itself (with some added overhead of updating the RETE network).

The transformation execution time was measured with the traditional pattern matcher, the unoptimized incremental pattern matcher and the optimized (see Section 3.4.1) incremental pattern matcher; for each of them, 10 experiments were performed, all with a first and a second run of the transformation. The average execution times (and corresponding standard deviations) are displayed on Table 3.2.

runtime(ms)	first run	second run
traditional matcher	3 806 (\pm 255)	3 438 (\pm 114)
unoptimized incremental	1 841 (\pm 216)	1 247 (\pm 47)
optimized incremental	1 844 (\pm 149)	1 242 (\pm 44)

Table 3.2: Average execution times (and standard deviation) in milliseconds for the UiPIM \rightarrow AJAX model transformation

The results show that for a practical model transformation problems, even though they were designed as batch transformations, the benefits of the incremental pattern could significantly outweigh its disadvantages, and it could prove as a strong alternative to the traditional pattern matcher. The implemented optimizations, however, had no detectable effect on the performance, at least not on the time complexity.

3.4.4 Conclusions

Our measurements have shown that for different types of problems, different pattern matchers are the most suitable. The incremental approach is desirable when patterns are frequently matched with only minor modifications between them. It is also shown that for ALAP (as long as possible)

type transformations (the iterate version), it has great advantages; see [42] for a report on how current model transformation engines tend to suffer from inefficiency at this problem class, making this an important achievement.

Practical model transformation problems can easily fall into the category where an incremental engine is advantageous, even if they were designed as batch jobs⁴. In these cases, the incremental pattern matcher is faster than the traditional matcher.

The measurements on practical model transformation problems did not show any benefit for the implemented optimizations, impeding the implementation of further optimization techniques. Future plans include determining whether this observation is general among practical model transformations and finding optimizations that do have an impact on them.

3.5 Summary

In this chapter, the RETE principle was shown to be suitable for assuming the role of an incremental VIATRA2 pattern matcher, both from a conceptional and a practical point of view. A RETE-based pattern matching method for the VIATRA2 framework was designed, an algorithm for constructing RETE networks for VIATRA2 graph patterns was also provided, and the proposed RETE-based pattern matcher was implemented. Measurements have shown that the new, incremental pattern matcher can prove to be more efficient than the traditional one, for application scenarios including model synchronisation. Beside its efficiency, the incremental pattern matcher is also the basis for event-driven transformations, to be introduced in the next chapter.

⁴for event driven transformations (see Chapter 4), an incremental matcher is a trivial requirement

Chapter 4

Event-driven incremental transformations with triggers

4.1 Goal of this Chapter

In this chapter, we present the new event-driven model transformation engine, which is built on top of the incremental pattern matcher introduced in Chapter 3. This event-driven transformation engine was conceptionally designed to work in a graph transformation environment; in our 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 (creation, deletion) (as proposed in [15]). 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 [33], 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 [34, 33], this approach could be extended to the execution of constraint-checking transformations as well as general model-to-code transformations.

Our approach embraces this idea, however we 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, for instance, 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 5, we provide examples of using this technology in the context of domain-specific languages; further research is expected in the applications unexplored in this paper.

With to this new transformation engine, various problems become easy and effective to implement. The main reason for creating this engine was to provide better support for the ViatraDSM domain-specific modeling environment, which is based on the VIATRA2 framework. The use of this new event-driven transformation engine with the ViatraDSM framework is demonstrated in Chapter 5.

4.2 Event-driven model transformations with triggers

4.2.1 The definition of our trigger concept

The condition of a trigger is a graph pattern. Thus an *event* in our 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 as the *action sequence*.

Finally, to ensure the proper functioning of the triggers we 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, with the first pattern occurrence.

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 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 modelspace at the trigger's startup. Whereas with *passive startup* all current matches that are present in the modelspace when the trigger is added to the framework are discarded. The default setting of a trigger is *active startup*.

4.2.2 Adapting triggers into the VIATRA2 model transformation framework

We adapted our 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*. These control options have to be set when starting our triggers. A trigger is started with the `startTrigger(gtrulename, priority, mode, sensitivity, startup)` function.

Although this works suitably, this is not the most adequate way to define the *control options*. Our first conceptional idea was to extend the Viatra Textual Command Language to use Java5-like *annotations* on the transformation rules to set these options, but the VIATRA2 framework's VTCL parser does not handle annotations perfectly. In all of the examples of this paper, this annotation syntax will be used in comments, however, with the continuous development of the VIATRA2 framework, this small coding detail is expected to be fixed in the near future, when the new VIATRA2 VTCL parser will be finished.

Note that currently we do not use the left hand side (LHS) - right hand side (RHS) GT representation to define our trigger, instead a format which is more analog to the procedural programming languages is used (the FUJABA notation [23]). 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 our future work we would like to extend our framework to handle the LHS-RHS format as well.

4.2.3 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 3.3.5. With the precondition and the action sequence, we have our transformation rule ready.

```
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 defined the following way:

```
startTrigger("suicide", 350, "always", "rise", "active");
```

A RETE network similar to that shown in Section 3.3.5 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.

4.3 Execution semantics

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

4.3.1 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 (see 4.5.1, page 64) 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 Δ) 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 Δ ; 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 modelspace changes belonging to the same transaction, or updates initiated by the same change but traveling different paths in the RETE network.

4.3.2 Operational semantics

Figure 4.1 shows the lifecycle of a trigger. Before the trigger is registered and becomes ready for execution, the corresponding delta monitor has to be constructed. 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. The figure also shows how triggers with the 'once' mode will get unregistered and discarded after the first time their action part is executed, while 'always' triggers can be activated again and again indefinitely.

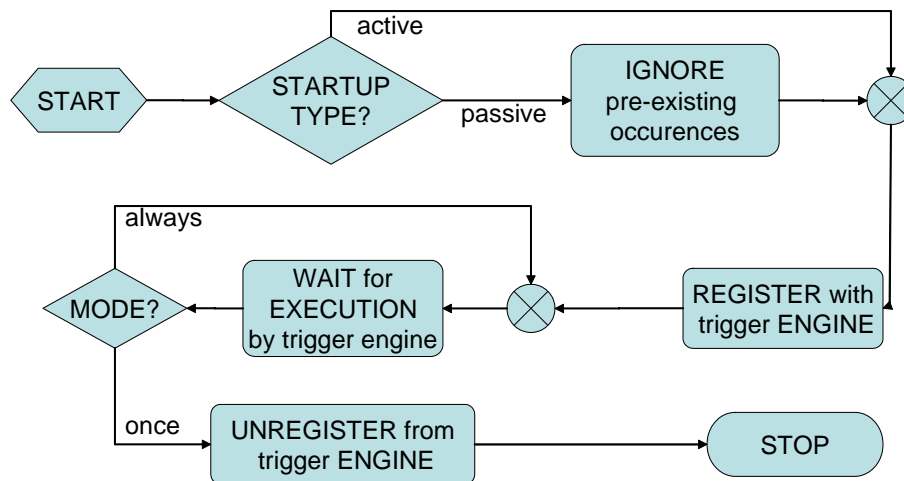


Figure 4.1: The lifecycle of a trigger

Now Figure 4.2 shows the operational semantics of the trigger engine. After a transaction is performed on the modelspace, the delta set is checked for new events. Events will be processed in the order of decreasing priority of the associated trigger. After the highest priority event is popped, the sensitivity of the corresponding trigger 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 action part of 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.

The action part of the trigger obviously influences the modelspace when executed. Consequently, it can generate events, both of higher and lower (or equal) priority; they will be handled 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 own responsibility to keep this in mind.

Other possibilities include that an action part performs a change that cancels out an unhandled

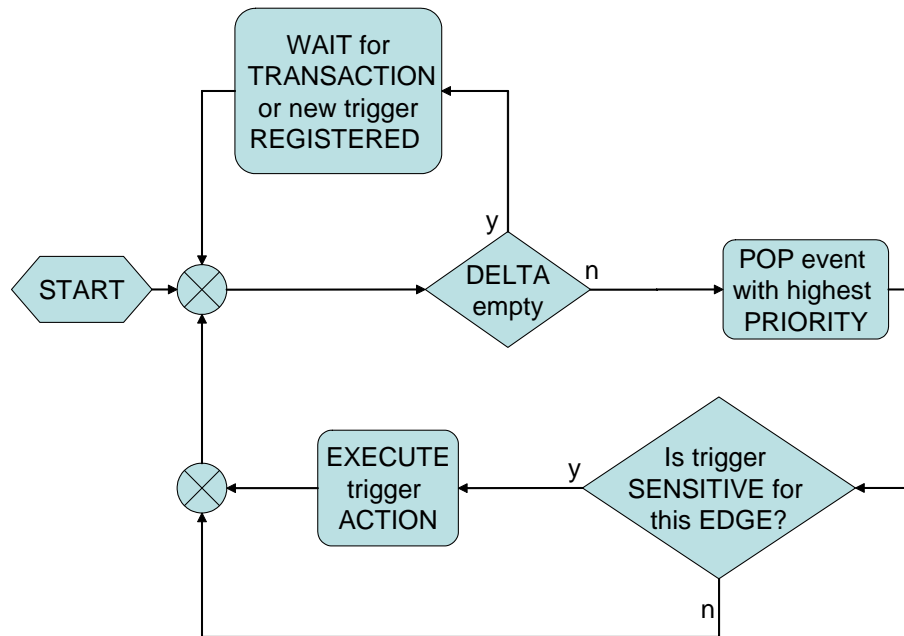


Figure 4.2: The operational semantics of the trigger engine

event. 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.

4.4 Handling synchronisational problems

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

4.4.1 Description of the problem

In a synchronisational 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. In each technology featured in Section 1.3 reference models are used. The essential difference between them is in the implementation and handling of these models. QVT defines automatically generated reference

models (or ‘trace’ models, as they call them), so the designer does not have to deal with this. In TGG on the other side, these additional correspondence model elements and rules are included in a TGG rule definition.

4.4.2 Our approach

In Section 4.2 the description of our triggers was given. Thanks to the versatility of the VIATRA2 graph patterns and ASM rules, triggers alone can solve numerous problems. But as we discussed in 4.4.1, the use of a reference model is inevitable to handle synchronisational 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 we think is one of the most important domains where our 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 5. Here a very simple example is described, just to show how easily the triggers with reference models can be used for synchronisation.

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 modelspace looks like that shown in Figure 4.3.

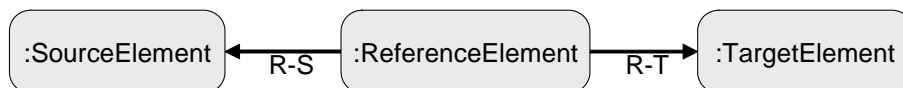


Figure 4.3: A consistent state of the modelspace

Creating a new model element In Figure 4.4 we can see that a new *SourceElement* can be easily determined, because no *ReferenceElement* exists with the appropriate *R-S* relation in the modelspace. 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 4.4: SourceElement created

Deleting a model element As long as every *SourceElement* is created properly we should have a consistent modelspace, 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 4.5 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.

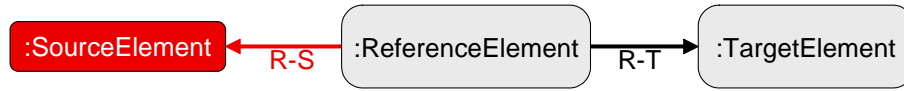


Figure 4.5: SourceElement deleted

4.5 Architecture of the implementation

Having settled the architectural concepts of the triggers in Section 4.3, 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 4.6).

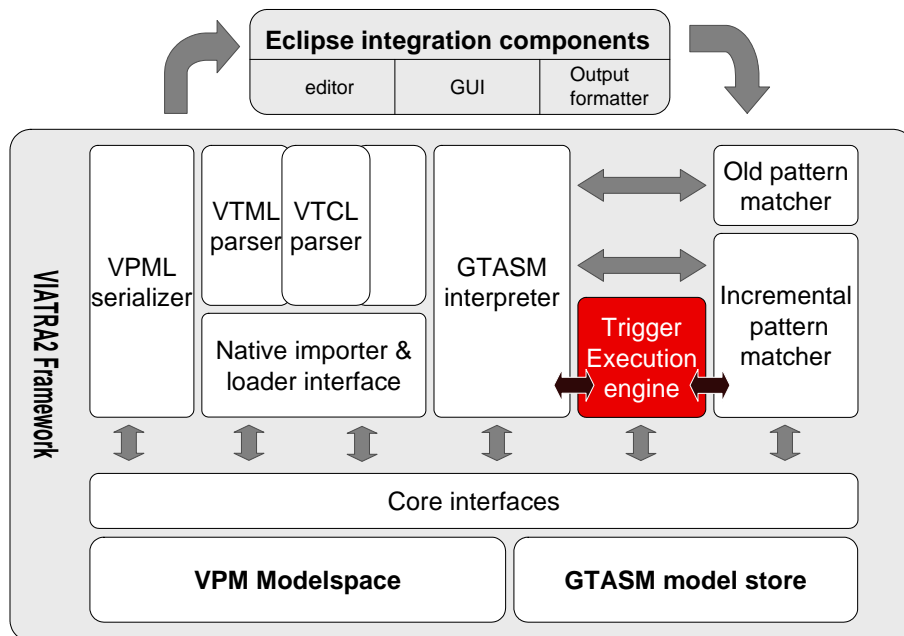


Figure 4.6: VIATRA2 R3 Framework extended with the Trigger Execution Engine

Every VIATRA2 VPML modelspace 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 4.5.1. 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 4.5.2.

4.5.1 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 incremental pattern matcher (see Chapter 3); falling edge events reflect negative updates.

Upon initialisation, the delta monitor accesses the production node (see 2.3.2, page 30) 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 (see 4.3.1, page 59), when the engine handles an event, it is removed from the delta monitor.

4.5.2 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 (see 4.3.1, page 59) 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.

Transaction handling also helps us to deal with situations when multiple triggers are ready to run. In this case one trigger is selected to run first, which was selected in compliance with the control options (see 4.2.1, page 57). At this point we do not have to deal with other triggers which are ready to run, in accordance with the fact that the selected trigger's action sequence is also run in a transaction. I.e. when this trigger's transaction ends, the Trigger Execution Engine starts again, and rechecks the delta set. This is important, because a trigger's action sequence may generate new matches for other triggers.

4.5.3 VIATRA2 user interface extension

The VIATRA2 framework has an Eclipse view, the 'Viatra R3 Frameworks' view. This view provides an interface to the import/export/parser facilities, and the loaded GTASM program models (machines) can also be seen here. We extended this view, so that the imported triggers can be viewed and deleted here.

4.6 Summary

In this Chapter our new event-driven transformation engine was presented. Although it can be used for any general incremental synchronisation 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 their example triggers are shown in Chapter 5.

Chapter 5

Event-driven transformations in domain-specific modeling

5.1 Goal of this Chapter

In this Chapter, the use of our event-driven model transformation engine will be presented. The base of this engine, the incremental pattern matcher, was presented in Chapter 3. The transformation engine based on this engine, along with the definition of our triggers, were discussed in Chapter 4. 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.

The extension of the VIATRA2 framework with a new event-driven transformation engine enables us to handle some aspects of domain specific modeling (e.g. logical and graphical representation synchronisation) more efficiently 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 can not be solved in the commercial domain specific modeling tools without writing program code.

In Section 5.2 a detailed presentation about the current challenges in the domain-specific modeling world is presented. In the following three sections, detailed examples are presented.

5.2 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.

5.2.1 Separation of abstract and concrete syntax representations

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

- (i) they are practically identical (as in the case of MetaCase MetaEdit+ [19]), 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 can 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, however, 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[38]: 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 section 5.3, we outline how this idea can be extended using model transformation technology to give the toolsmith maximum flexibility in completely separatin abstract and concrete syntax representations.

5.2.2 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 [10] 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 [26] 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 [29]). 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 [33], both cases can be formulated as model transformation problems, involving the execution of checking transformations. In section 5.4, we demonstrate the feasibility of this approach using *incremental* transformations, which allow for a high-performance implementation and also yield an intuitive way of specification.

5.2.3 Transformations in domain-specific visual languages

Two language engineering aspects which are not covered in state-of-the-art tools are

- **Dynamic (operational) semantics**, modeling the operational behaviour of language concepts;
- and **Transformations (Denotational/Translational semantics)** specifying how the abstract syntax can be translated into a semantic domain (e.g. programming language).

Model-to-model and model-to-code transformations have been traditionally implemented as separate frameworks, not integrated into a (domain-specific) modeling environment (with the exception of the Visual Modeling and Transformation System [18]). However, in order to gain industry-wide acceptance, such as UML did, DSMs have to integrate into an existing model-based infrastructure and the corresponding toolchain, which can be facilitated using integrated model transformation technology. Moreover, an integrated approach can offer various advantages; for instance, with a code generator integrated into the modeling environment, the domain expert can see the textual representation of the models under design instantly. Unfortunately, DSM tool providers have not yet recognized the importance of this; in section 5.5, we describe an example application utilizing our incremental transformation technology. This example demonstrates the flexibility of our approach and hints at the possibilities of further applications.

5.3 Abstract-concrete syntax synchronisation

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. As we described in Chapter 1, the possible graphical representations for a given logical model are limited by the currently available DSM frameworks. Here we propose a new approach for incremental abstract-concrete syntax synchronisation with the possibility of arbitrary abstract-to-concrete syntax mapping.

5.3.1 Architectural changes in ViatraDSM

The diagram metamodel (see Figure 5.1) of the ViatraDSM framework has been changed. Instead of the previous modelbindings, every DiagramElement has a MappingElement.

The new metamodel for the MappingElement is shown in Figure 5.2. 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.

5.3.2 Petri net abstract-concrete syntax synchronisation

In this example we demonstrate the use of our triggers with the new Mapping Elements to achieve incremental abstract-concrete syntax synchronisation in a Petri net editor. The domain metamodel of the Petri nets was shown in the previous Section, see Figure 5.5, while the diagram metamodel can be seen in Figure 5.3. The mapping between these two metamodels is simple, for each PlaceFigure a Place, for each TransitionFigure a Transition should be created.

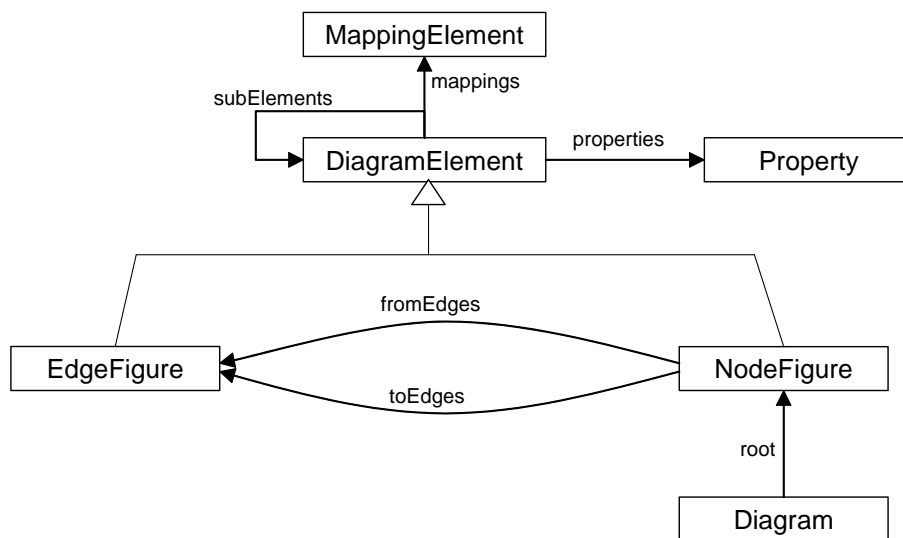


Figure 5.1: Diagram metamodel

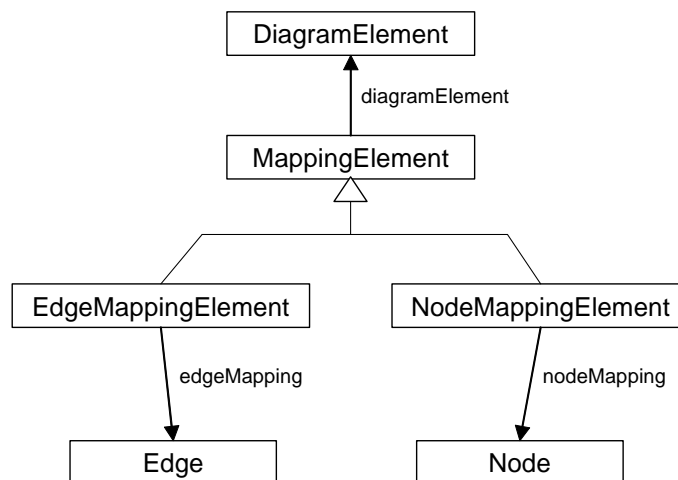


Figure 5.2: Diagram mapping metamodel

The highest priority trigger should be one which creates a ‘modelspace’ for the ‘figure container’. Thus, if a PetriNetFigure is created, then the `createDomainModelspace` trigger’s action sequence creates a PetriNet. The precondition has a negative pattern call to ensure that PetriNet-Figures with MappingElements will not match it.

```

/**
 * @Trigger(priority=50, mode='`always`',
 *          sensitivity='`rise`', startup='`active`')
 */
gtrule createDomainModelspace(inout PNF, inout D) =
{
  precondition pattern lhs(PNF, D)=
  {
    PetriNetDiagram(D);
    PetriNetDiagram.PetriNetFigure(PNF);
    'Diagram'.root(RA,D,PNF);
    neg find mappedfigure(PNF);
  }
}

```

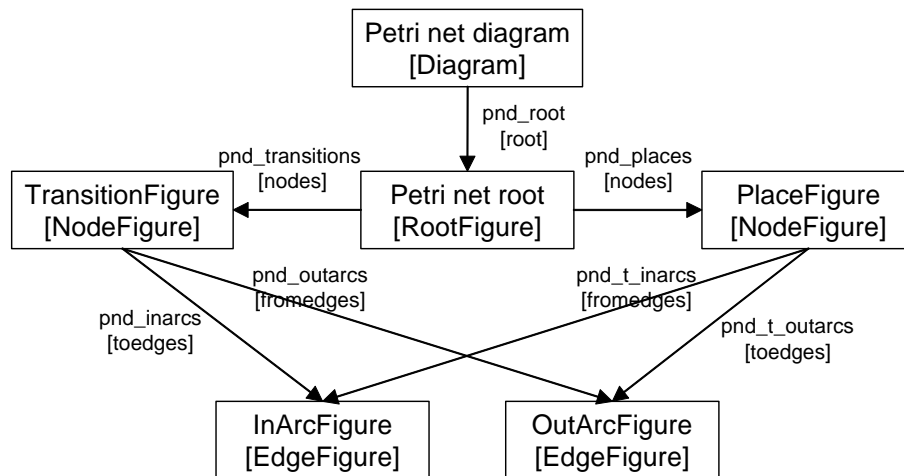


Figure 5.3: Petri net diagram metamodel

```

}
action
{
  let PN = undef, ME = undef, RA = undef, RB = undef, RC = undef in
  seq {
    new ('DiagramMapping'.NodeMappingElement (ME) in D );
    new ('Diagram'.DiagramMapping'.mappings (RA,PNF,ME) );
    new ('PetriNetEditor'.PetriNet (PN) in 'DSM'.model.'PetriNet');
    new ('PetriNetEditor'.petriNets (RB,'DSM'.model.'PetriNet',PN));
    new ('DiagramMapping'.NodeMappingElement'.nodeMapping (RC,ME,PN));
  }
}

pattern mappedfigure (PNF) =
{
  'Diagram'.DiagramElement (PNF);
  'Diagram'.DiagramMapping'.NodeMappingElement (ME);
  'Diagram'.DiagramElement'.mappings (RA,PNF,ME);
}

```

The next trigger is one that creates a Place for a PlaceFigure. This trigger has a larger precondition, which ‘finds’ the destination PetriNet, where the Place should be created. This can be seen in the `placeexist` pattern call. This is included here to support the drag-and-drop function of the ViatraDSM framework. If an element is dropped on the diagram, then the framework creates the PlaceFigure with the same name as the dropped Place. So a new match will be generated for this trigger. Here, the `placeexist` pattern call returns with true value in the action sequence, because there is a Place with the same name in the model space. In this case, the creation of a new Place is not needed, instead it is just mapped to the PlaceFigure.

```

/**
 * @Trigger(priority=30, mode='always',
 *          sensitivity='rise', startup='active')
 */
gtrule newPlaceFigureCreated (inout PF, inout PN, inout PNF) =
{
  precondition pattern lhs (PF, PN, PNF)=
  {
    PlaceFigure (PF);
  }
}

```

```

//'Find' the PetriNetFigure,
// which 'contains' this PlaceFigure:
'PetriNetDiagram'.PetriNetFigure(PNF);
'PetriNetDiagram'.'PetriNetFigure'.pnd_places(RA,PNF,PF);

//'Find' the relating MappingElement for this PetriNetFigure
'DiagramMapping'.NodeMappingElement(ME);
'DiagramElement'.mappings(RB,PNF,ME);

//'Find' the PetriNet for the PetriNetFigure
'PetriNetEditor'.PetriNet(PN);
'DiagramMapping'.'NodeMappingElement'.nodeMapping(RC,ME,PN);
neg find mappedfigure(PF);
}

//The action sequence creates a Place
action
{
let ME = undef, P = undef, RA = undef, RB = undef, RC = undef in
  seq {
    //Creation of the MappingElement for the PlaceFigure:
    new ('DiagramMapping'.NodeMappingElement(ME) in PNF );
    new ('Diagram'.'DiagramElement'.mappings(RA,PF,ME) );
    // If there is a Place with the same name, as the Placefigure
    // then this Place is linked to the MappingElement instead of
    // creating a new one.
    try choose P with find placeexist(P,PF) do
      new('DiagramMapping'.'NodeMappingElement'.nodeMapping(RC,ME,P) );
    else seq {
      new(Place(P) );
      new('PetriNet'.places(RB, PN, P) );
      new('DiagramMapping'.'NodeMappingElement'.nodeMapping(RC,ME,P) );
    }
  }
}

pattern placeexist(P,PF) =
{
  Place(P);
  PlaceFigure(PF);
  check ( name(P) == name(PF) )
}

```

The `newTransitionFigureCreated` trigger is not shown, because it is conceptionally the same as the `newPlaceFigureCreated` which was demonstrated above.

The next two triggers shown here are generic delete triggers. These two triggers handle deleting in a domain, where the `NodeFigures` are mapped to `Nodes` with 1:1 multiplicity. Moreover these triggers only use `ViatraDSM` coremetamodel elements, so they are domain metamodel independent.

The first trigger, named `deletedNode` is sensitive for a pattern which has a `Nodefigure`, with its related `MappingElement` which has no related `Nodes`. This occurs when a `ModelElement` is deleted, thus its related `Figure` should be deleted from the graphical representation as well. Naturally, the `MappingElement` used by them also has to be deleted.

```

/**
 * @Trigger(priority=10, mode='`always`',

```

```

*      sensitivity='`rise`', startup='`active`')
*/
gtrule deletedNode(inout NF, inout ME) =
{
  precondition pattern lhs(NF, ME)=
  {
    'Diagram'.NodeFigure(NF);
    'DiagramMapping'.NodeMappingElement(ME);
    'DiagramElement'.mappings(RA, NF, ME);
    neg find matchingnode(ME);
  }
  action
  {
    seq {
      delete (ME);
      delete (NF);
    }
  }
}

pattern matchingnode(ME) =
{
  'DiagramMapping'.NodeMappingElement(ME);
  'Editor'.Node(N);
  'DiagramMapping'.'NodeMappingElement'.nodeMapping(RA, ME, N);
}

```

The second trigger, named `deletedNodeFigure` is the opposite of the `deletedNode`. It is sensitive for a pattern which has a `Node`, with its related `MappingElement` which has no related `NodeFigures`. This occurs when a graphical element is deleted from the diagram. Its related model element could be deleted, but that is optional. Naturally, the `MappingElement` used by them has to be deleted.

```

/**
 * @Trigger(priority=10, mode='`always`',
 *      sensitivity='`rise`', startup='`active`')
 */
gtrule deletedNodeFigure(inout N, inout ME) =
{
  precondition pattern lhs(N, ME)=
  {
    'DiagramMapping'.NodeMappingElement(ME);
    'Editor'.Node(N);
    'DiagramMapping'.'NodeMappingElement'.nodeMapping(RA, ME, N);
    neg find matchingnodefigure(ME);
  }
  action
  {
    seq {
      delete (ME);
      ///! Delete model element also.
      // (Optional)
      delete (N);
    }
  }
}

pattern matchingnodefigure(ME) =
{
  'Diagram'.NodeFigure(NF);
}

```

```

'DiagramMapping'.NodeMappingElement (ME);
'DiagramElement'.mappings (RA, NF, ME);
}

```

These two triggers can handle all the deleting needed in the Petri net, i.e. deleting Place, Placefigure, Transition, TransitionFigure, PetriNet, PetriNetFigure elements.

5.3.3 Conclusion

In this example we proposed a way to use triggers and also the Mapping metamodel to achieve incremental abstract-concrete syntax synchronisation. Due to the versatility of triggers, domain-specific language designers do not have to write any sort of Java source code to handle arbitrary abstract-concrete syntax synchronisation. However, a good future improvement for the ViatraDSM could be to design fully generic triggers for simple abstract-concrete synchronisation problems, like this one.

5.4 Incremental constraint checking

As we described in Chapter 1, 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.

5.4.1 Architectural changes in ViatraDSM

A new metamodel (see Figure 5.4) for constraint checking was added to the ViatraDSM metamodels. Actually this is a reference model which can be used by the constraint designer to track the incremental changes in the modelspace. Although it should suit most constraint evaluation scenarios, it can be extended by the domain developer if needed.

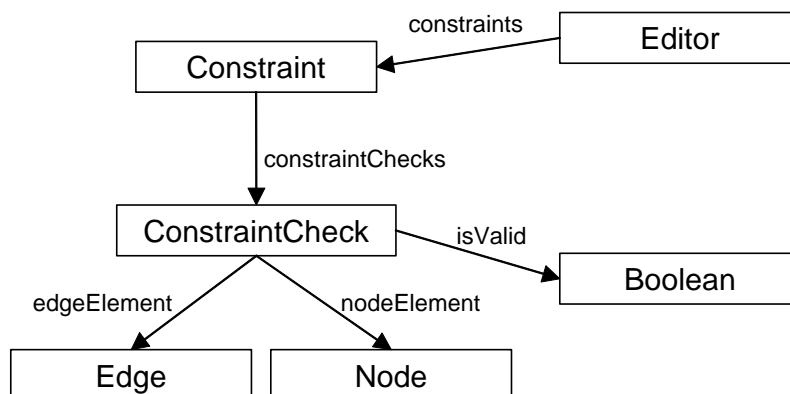


Figure 5.4: Constraint metamodel

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 *constraints* link the *Constraint* model elements to this single container. (One domain can have multiple constraints of course.)

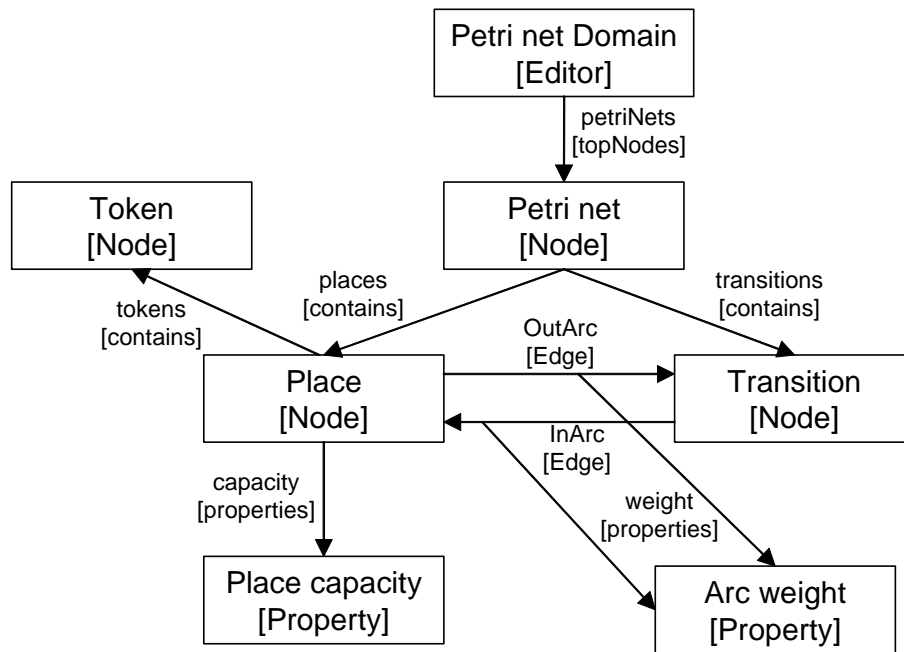


Figure 5.5: Example of a domain metamodel (Petri net domain)

Constraint represents a constraint in the model. The relation *constraintChecks* links the *ConstraintCheck* model elements to this model element.

ConstraintCheck model elements are connected to their appropriate *Constraint* element through the *constraintChecks* relation. This is the most important element in the metamodel. It can be connected to various node and edge elements in the domain's modelspace. The *ConstraintChecks* are used to tag the domain elements which have been checked. A *Boolean* property shows if this set of edges and nodes violates the constraint.

5.4.2 Petri net capacity constraint

In the followings, we will demonstrate the use of the new *Constraint* metamodel, by designing a simple incrementally evaluated constraint for the Petri net domain, which we introduced in Section 2.2.3.

On Figure 5.5, the domain metamodel of the Petri nets can be seen. There are two important Nodes, the *Place*, and the *Token*. *Place* has a *Place-Capacity* property that defines the number of tokens a place can contain at any one time. Tokens are represented by separate *Token* elements in the model. The constraint is violated, if the number of *Token* elements at a given *Place* exceeds the value defined in the *Place-Capacity* property. This capacity constraint has to be checked on each *Place*, which has a *Place-Capacity* property.

The first step is to create a *Constraint* element in the Petri nets' topmost *Editor* element, this will be used to store the *ConstraintCheck* elements.

The second step is to design the *ConstraintChecks*. Every *ConstraintCheck* element should be connected to a single *Place*, this way the *ConstraintCheck*'s *isValid* boolean property will state the constraint-satisfaction in that *Place*. Additionally, every *Token* present at the *Place* should be connected to the related *ConstraintCheck* as well. This is crucial to be able to handle the incremental changes in the modelspace (e.g. the creation of a token can be identified, by the fact

that it is not connected to a ConstraintCheck).

The first two triggers can now be written. The `ccToPlace` trigger will create ConstraintCheck elements for the places. It is an active trigger, which means, that it will create a ConstraintCheck for every Place already present in the modelspace, which satisfies the precondition pattern. It has a negative pattern call in the precondition to check whether it has already been connected or not.

```
/**
 * @Trigger(priority=32, mode='`always`',
 *          sensitivity='`rise`', startup='`active`')
 */
gtrule ccToPlace (inout P) =
{
  /**
   * Checks if it has Capacity property
   * Negative condition checks
   * if a ConstraintCheck has already been created for it
   */
  precondition pattern lhs (P) =
  {
    'Place' (P);
    'Place'.'Place_Capacity' (PC);
    'Place'.'capacity' (RA,P,PC);
    neg find placeSet (P);
  }

  /**
   * Creating the ConstraintCheck, and its relations
   */
  action
  {
    let CC = undef, RA = undef, RB = undef in
    seq {
      new ( 'Constraint'.ConstraintCheck (CC)
          in 'DSM'.model.'PetriNet'.petrinetcapacity );
      new ( 'Constraint'.constraintChecks (RA,
          'DSM'.model.'PetriNet'.petrinetcapacity, CC) );
      new ( 'Constraint'.'ConstraintCheck'.nodeElement (RB, CC, P) );
    }
  }
}

pattern placeSet (P) =
{
  'Constraint'.'ConstraintCheck' (CC);
  'Place' (P);
  'Constraint'.'ConstraintCheck'.'nodeElement' (RA,CC,P);
}
```

The `ccToToken` trigger will connect the tokens to the appropriate ConstraintChecks. It is similar to the `ccToPlace` trigger in a way that it is an active trigger, which means, that it will create a ConstraintCheck for every Place in the modelspace, which satisfies the precondition pattern. Both triggers have a negative pattern call in the precondition to check whether they have been connected already or not.

```
/**
 * @Trigger(priority=31, mode='`always`',
 *          sensitivity='`rise`', startup='`active`')
 */
gtrule ccToToken (inout T, inout CC) =
```

```

{
precondition pattern lhs(T, CC)=
{
  'Constraint'.ConstraintCheck(CC);
  Place(P);
  'Constraint'.'ConstraintCheck'.nodeElement(RA,CC,P);
  'Place'.Token(T);
  'Place'.tokens(RB, P, T);
  neg find tokenSet(T);
}

/**
 * ``Connects`` the token to the appropriate ConstraintCheck
 */
action
{
  let RA = undef in
    new ( 'Constraint'.'ConstraintCheck'.nodeElement(RA, CC, T) );
}

pattern tokenSet(T) =
{
  'Constraint'.ConstraintCheck(CC);
  'Place'.Token(T);
  'Constraint'.'ConstraintCheck'.nodeElement(RA,CC,T);
}

```

Next, an ASMFfunction has to be defined, where the actual number of tokens connected to the Place can be stored. The VIATRA2 ASMFfunction is a persistent associative array which can be reached from every trigger. As a domain in general can have multiple constraints to index this associative array, the ConstraintCheck's name is a better choice instead of the Place's name.

```

//An array to cache token numbers
asmfunction numberoftoken / 1;

```

To update this array and set the isValid boolean property of the ConstraintCheck, three small triggers have to be defined. As the numberoftoken will be initialized to 0, when the placeAdded trigger runs, it needs to have a higher priority than the tokenAdded and tokenRemoved triggers.

```

/** @Trigger(priority=23, mode='`always``,
 *     sensitivity='`rise``, startup='`active``')
 */
gtrule placeAdded(inout CC) =
{
  precondition pattern lhs(CC)=
  {
    'Constraint'.ConstraintCheck(CC);
    Place(P);
    'Constraint'.'ConstraintCheck'.nodeElement(RA,CC,P);
  }
  action
  {
    seq { //Initialize the numberoftoken array, with CC
      update numberoftoken(CC) = 0;
      call booleanUpdate(CC); }
  }
}

/** @Trigger(priority=22, mode='`always``,

```

```

*      sensitivity='rise', startup='active')
*/
gtrule tokenAdded(inout CC) =
{
  precondition pattern lhs(CC)=
  {
    'Constraint'.ConstraintCheck(CC);
    Token(P);
    'Constraint'.'ConstraintCheck'.nodeElement(RA,CC,P);
  }
  action
  {
    seq {
      update numberOftoken(CC) = numberOftoken(CC) + 1;
      call booleanUpdate(CC); }
  }
}

/** @Trigger(priority=21, mode='always',
*      sensitivity='fall', startup='passive')
*/
gtrule tokenRemoved(inout CC) =
{
  precondition pattern lhs(CC)=
  {
    'Constraint'.ConstraintCheck(CC);
    Token(P);
    'Constraint'.'ConstraintCheck'.nodeElement(RA,CC,P);
  }
  action
  {
    seq {
      update numberOftoken(CC) = numberOftoken(CC) - 1;
      call booleanUpdate(CC); }
  }
}

```

At first glance, one could think that the initialization of the numberOftoken ASMFuction could have been done in the ccToPlace trigger. The problem with that would be, that when the modelspace is closed, the data stored in the ASMFuctions are discarded. Thus when the modelspace is reopened, these values have to be reinitialised for the existing places. Alternatively, the number of tokens could have been stored in the ConstraintCheck's value field, this would have been better in this case. However, the authors intention was to present a more general approach, which is the use of ASMFuctions, as the ConstraintCheck's value field would not be suitable in some cases (e.g. when storing multiple items).

5.4.3 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 ConstraintCheck. These 'builder' triggers usually should not use ASMFuctions as the informations they store are discarded when closing the modelspace.

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.

5.5 Incremental code generation

In this section, we demonstrate the potential of our incremental model transformation technology for code generation. As an example, a simple ViatraDSM-based workflow editor, created for the SENSORIA Framework Programme 6 European Union research project [31], will be used.

The SENSORIA project has produced, among other results, an Eclipse-based tool integration framework which can be used to create workflows which make use of Eclipse-based tools to facilitate the automated execution of common development tasks requiring multiple steps. This framework, the SENSORIA CASE Tool [32, 30], uses the JavaScript language to orchestrate the workflows, and requires tool providers to create integration classes conforming to an interface so that the engine can access the functionality of the various tools.

In order to support the model-based design of workflows for the SENSORIA environment, the Department of Measurement and Information Systems has created a domain-specific metamodel for the ViatraDSM environment and a corresponding editor plug-in. This plug-in can be used to draw workflows and create model-based descriptions of tools interfaces along with datatype

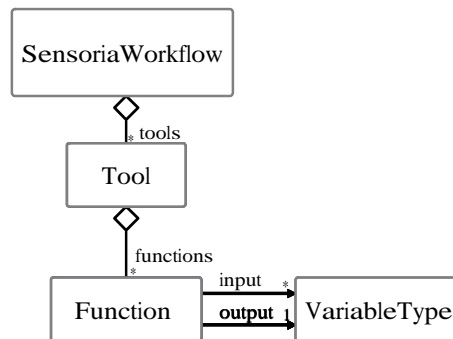


Figure 5.6: The SENSORIA Workflow Interface definition metamodel

On Figure 5.6, a portion of the SENSORIA workflow metamodel can be seen. A `SensoriaWorkflow` element contains `Tool` definitions, which are essentially programming interfaces consisting of `Function`s. Such a function can reference an arbitrary number of input arguments of a `Variable` type, and one single output type (if none is specified, the output type of “void” should be used).

The example code generation, which will be demonstrated in this section, generates the textual interface descriptions in the format like the following:

```

context ExampleWorkflow {
  interface ExampleTool {
    void Function1( )
    String Function2( String; Object; )
  }
}
  
```

5.5.1 The task

In this example, the goal of the toolsmith is to create a code generator for the SENSORIA workflow domain, which can generate a textual representation of the models ready to be deployed on the target platform. In the case of the SENSORIA Case Tool, as previously mentioned, the workflow itself needs to be constructed as a Javascript program, while interface descriptions are encoded using a specific C-like syntax. In this example, for the sake of simplicity, we only show the concept and basic ideas behind implementing the generator for the interface generation.

Normally, a code generator can be thought of as a set of templates, which contain static parts as well as some “glue code” which handles the mapping of dynamic (model-specific) elements to an appropriate textual format. The generation process involves a model traversal (typically in a single pass, since the cursor position on the output cannot be set arbitrarily), in which an appropriate template is chosen for a given model element and the output is produced by “applying” the template to that element. In many cases, the templates themselves contain the “control flow” code, making use of the document hierarchy which is typically present (i.e. the template for the “root” node calls all the others in a specific order).

Such a code generation process can be easily implemented as a VIATRA2 transformation; typically, ASM control structures are used to encode the control flow of output generation, while `print()` rules are used to produce the actual output. For instance, a small code generation snippet, capable of producing interface descriptions, could be written like the following VTCL program:

```
/**
 * Template used for interface functions. It will generate the following
 * output:
 * <return type> <function name> ( <semicolon separated list of input types> )
 */
rule generate_function_description(in Function) = seq
{
    print("\t");
    try choose ReturnType with
        find function_output_vartype(Function, ReturnType) do seq
    {
        // pretty print return type
        print(name(ReturnType)+" ");
    }
    else print("void ");
    // pretty print function input signature
    print(name(Function)+" ( ");
    forall InputType with
        find function_input_vartype(Function, InputType) do seq
    {
        // pretty print variable type definition
        call print_vartype_definition(InputType);
        print(";");
    }
    println(" )");
}

/**
 * Main rule to invoke code generation for all workflows and tool interfaces.
 * Output format:
 * context <Workflow name> {
 *   <list of tool interface descriptions>
 * }
 *
 * A tool interface description has the following format:
```

```

* interface <Tool interface name> {
*   <list of function descriptions>
* }
*/
rule main() = seq
{
    forall WF with find workflow(WF) do seq
    {
        println("context "+name(WF)+" ");
        forall T with find workflow_tool(WF,T) do seq
        {
            println("interface "+name(T)+" ");
            forall Func with find tool_function(T, Func) do call
                generate_function_description(Func);
            println("");
        }
        println("");
    }
}

```

Note that the trivial pattern definitions used above have been omitted for the sake of clarity and simplicity.

An important characteristic of this technology is that it involves a *single-pass top-down process* which always starts from the document root, i.e. for a small change in the workflow model, the whole output has to be regenerated. While this may not always be a problem – for instance, when application code only needs to be derived during deployment –, in many applications, textual representations are an integral part of the development process. Therefore, it is a serious hindrance if every small modeling operation results in a long-running generation procedure.

As a well-known example, one could think of graphical user interface designers. In Microsoft's popular Visual Studio environment, the developer can create an application form by drag-and-drop operations, customization using a property sheet etc. Although the user is primarily working on that interface, the actual code for the form is automatically generated and is kept in accordance incrementally with what the user is drawing. This is important since in the user needs to know exactly what code was generated, because of control purposes as well as checking whether the editing operation has been appropriately performed.

Since many domain-specific visual languages have some kind of corresponding textual representation, the need for such a facility for custom visual languages arises. The goal of this technology is to support incremental code generation, so that textual representation of domain model elements can be quickly derived independently of others, in an event-driven fashion, following the user's editing operations. High-level support for this is lacking in all current domain-specific language engineering tools, hence, one may be able to implement it, but it requires substantial manual coding in the environment the tool has originally been written – provided the given framework's API supports such a modification (or one has access to the source).

5.5.2 Our solution

In the context of our research, the idea behind implementing incremental code generation is essentially the event-driven execution of code generation templates such as the example in 5.5.1. However, there are two main differences to the traditional code generation problem:

- the code generation templates must not contain control flow code which can interfere with

the logic of the incremental generation (i.e. no generation should take place for elements not relevant in the change that triggered the generation process);

- additionally, since the output still needs to be syntactically correct, the framework has to support “cursor positioning” in order to locate the position in the output which has to be written to (overwritten).

The first requirement can be easily fulfilled by following a “design pattern” for templates which avoids document hierarchy jumps in a template’s control flow code.

The second requirement is more difficult to achieve. The most common solution to the “repositioning” problem involves the usage of an additional abstraction layer above the code output buffer in the form of an abstract syntax tree (AST). Such a tree encodes the structure of the output and maintains a small output buffer for each element. This way, a smaller portion of the output can be added or overwritten without invalidating the syntactic correctness (since the AST ensures the integrity of the output).

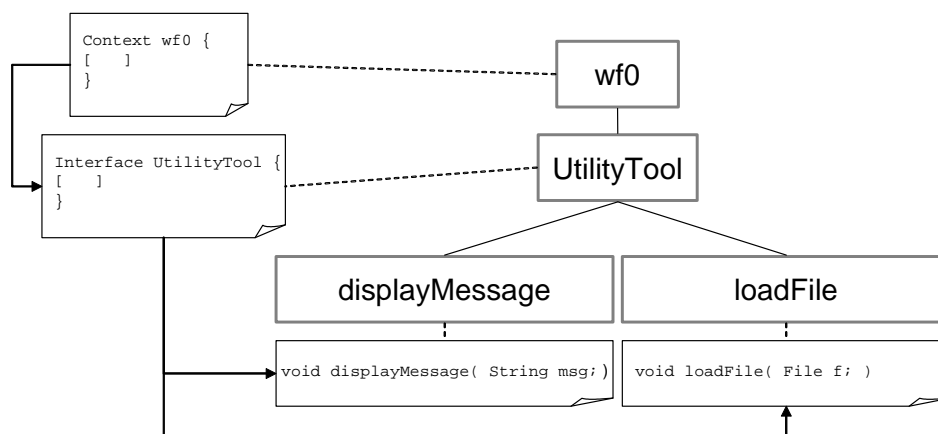


Figure 5.7: Simple Abstract Syntax Tree representation of the output

This representation should have a granularity corresponding to the granularity of the incremental code generation process, i.e. there should be a discrete element in the AST for each model node for which an incremental generation procedure should run. For instance, in the context of the SENSORIA example, if one wants to recreate code snippets corresponding to tool interface functions, then the AST (see Fig. 5.7) for the output needs to contain “tool interface function” elements (and an appropriate buffer for their textual representation). The final output is serialized by a simple inorder traversal of the abstract syntax tree.

Since the Eclipse-based JFace editor framework provides built-in functionality for creating AST-based textual editors, we have utilized its AST-based document model implementation for the purposes of this example. This means, that the code generating `print()` calls have to be replaced with a special VIATRA2 native function, which allows the manipulation of the AST’s output buffers from VTCL code:

```
rule generate_function_description(in Function) = seq
{
    // select the appropriate element in the AST
    locate_buffer(Function);
    print_buffer("\t");
    try choose ReturnType with
        find function_output_vartype(Function, ReturnType) do seq
    {
```

```

        // pretty print return type
        print_buffer(name(ReturnType)+" ");
    }
    else print_buffer("void ");
    // pretty print function input signature
    print_buffer(name(Function)+"( ");
    forall InputType with
        find function_input_vartype(Function, InputType) do seq
    {
        // pretty print variable type definition
        call print_vartype_definition(InputType);
        print_buffer(";");
    }
    println_buffer(")");
    release_buffer(Function);
}

```

5.5.3 Code generation with triggers

The code generation state metamodel

In order to be possible to use pattern-based triggers to invoke transformations, a correspondence (or reference) metamodel, similar to those introduced in Sections 5.1 and 5.3, has to be established. The purpose of this metamodel is to describe auxiliary (helper) model elements which store the persistent state of the transformation process relevant for a certain set of elements.

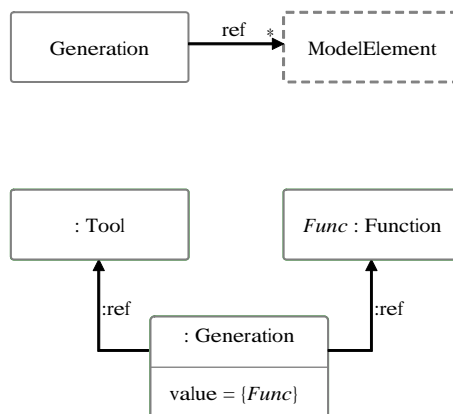


Figure 5.8: The code generation state metamodel

This simple metamodel (see Fig 5.8) contains a basic node (“generation”) and a relation with many-to-many multiplicity (“generation.ref”). The metamodel can be applied as shown in the bottom of Figure 5.8 to establish a logical relationship with an arbitrary number of model elements. In this particular case, this node is created to establish a *generation* relation between a *Tool* instance and its *Function*, storing the information that the code output for the *Function* element has already been created with the *Func* string as the function’s name.

This relationship can be generalized in a pattern description in VTCL:

```

/*
 * This pattern expresses the fact that the Function instance
 * has been processed by the code generator by the presence of
 * the generator reference model.
 * The data which is relevant in the context of the atomic code

```

```

* generation operation is stored in the reference model.
*/
pattern processed_tool_function(Tool, Function) =
{
    'SensoriaWorkflow'.'Tool'(Tool);
    'SensoriaWorkflow'.'Tool'.'Function'(Function);
    'DSM'.'codegen'.'generation'(CG);
    'DSM'.'codegen'.'generation'.'ref'(R1,CG,Tool);
    'DSM'.'codegen'.'generation'.'ref'(R2,CG,Function);
    check(value(CG) == name(Function))
}

```

Generally, the data which is relevant in the context of the atomic code generation operation (that is, the name of the Function element), is stored in the reference model. When this information changes, the code generation has to be invoked again. This is done by a trigger, which makes (partial) use of the above pattern to detect that the model has been modified and it is not in accordance with the generation state represented in the generation correspondence models.

The trigger code below describes a “fall” trigger which is fired when the assertion “value(CG) == name(Function)” becomes invalid (note: generally speaking, it also fires whenever a match of the precondition pattern is lost, but that action cannot be invoked from the simplified user interface). The corresponding trigger action will update the reference model and invoke the atomic code generation rule.

```

/**
* @trigger(0, "always", "fall", "passive")
*/
gtrule function_changed(inout Function) =
{
    precondition pattern lhs(Function, CG) =
    {
        'SensoriaWorkflow'.'Tool'.'Function'(Function);
        'DSM'.'codegen'.'generation'(CG);
        'DSM'.'codegen'.'generation'.'ref'(R1,CG,Function);
        check (value(CG) == name(Function))
    }
    action
    {
        // update reference model
        setValue(CG, name(Function));
        // generate output
        call generate_function_description(Function);
    }
}

```

In order to enable such operations, the framework has to ensure that the initial state of the model is set properly, i.e. the generation reference model is created when a new element is added by the user. This is done by the following trigger:

```

/**
* @trigger(100, always, rise, passive)
*/
gtrule add_function_to_tool(inout Function, inout Tool) =
{
    precondition pattern lhs(Tool, Function) =
    {
        find tool_function(Tool, Function);
        // this function has not been processed yet
        neg find processed_tool_function(Tool, Function);
    }
}

```

```

action
{
  // create reference model
  let R1 = undef in
  let R2 = undef in
  let CG = undef in seq
  {
    new ( 'DSM'..'codegen'..'generation' (CG) in Tool );
    new ( 'DSM'..'codegen'..'generation'..'ref' (R1, CG, Tool) );
    new ( 'DSM'..'codegen'..'generation'..'ref' (R2, CG, Function) );
    setValue (CG, name (Function) );
  }
  // create output
  call generate_function_description (Function);
}

```

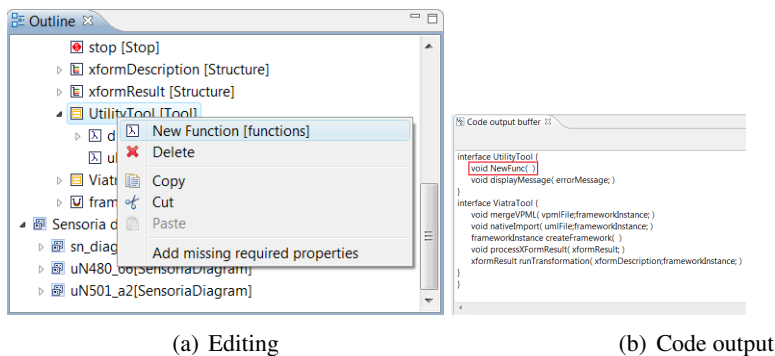


Figure 5.9: User editing action: adding a new function and the generated code output snippet

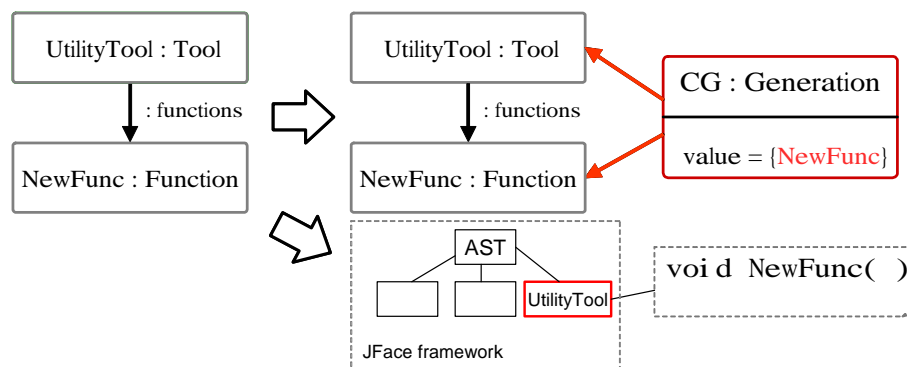


Figure 5.10: Code generation trigger execution

As it can be seen on Figure 5.10, the trigger is fired when the user adds a new Function element to a Tool instance (see Fig. 5.9/a). The negative application condition in the precondition pattern expresses that the Tool-Function pair has not been processed yet by our demo code generator. The action sequence adds the proper correspondence model elements and sets the initial value of the starting name of the Function instance, after which the initial textual output is generated (see Fig. 5.9/b).

After that, whenever the user changes the name of the Function instance, the first trigger will

fire and regenerate the corresponding textual output snippet.

In the example above, it is important to note the following. While the `function_changed` trigger is intended to listen to changes of the name of the `Function` element, each execution calls the rule to generate the whole function description – while, for instance, the input signature of the function remains unaffected and thus, it seems inefficient to regenerate the entire output. This seems to violate the first requirement laid out in 5.5.2, but it also points out a very important consideration: since the execution of that generation rule in our approach is much faster than with a “conventional” pattern matcher implementation – due to the fact that all the (partial) matches are cached in the RETE network –, the most time consuming operation (finding the matches) is reduced to constant time in these cases.

Therefore, the incrementality of the code generation application not only manifests itself in the trigger-based execution of the templates, but also in the fact that the output generation for unchanged elements is an order of magnitude faster because of the nature of the underlying RETE network. This means, that the implementation is not required to use extremely fine-grained ASTs for the sake of efficiency; the designer should select the granularity based on the possible user editing actions rather than the actual “syntax” of the generated output.

Using ASM functions as reference

In certain cases, an alternative technique may be used to store the generation state. Instead of correspondence models, persistent ASM functions can be utilized to store information about what model elements (and in what state) have already been processed. For instance, a much simpler implementation of the example above could be coded in VTCL like this:

```
/**
 * Associative array to cache Function names.
 */
asmfunction func_names / 1;

/**
 * @trigger(0, always, fall, passive)
 */
gtrule function_name_change(inout Function) =
{
    precondition pattern lhs(Function) =
    {
        'SensoriaWorkflow'.Tool.'Function'(Function);
        check (name(Function) == func_names(Function))
    }
    action
    {
        update func_names(Function) = name(Function);
        call generate_function_description(Function);
    }
}

rule main() = seq
{
    // start add_function_to_tool trigger
    print(startTrigger("add_function_to_tool", 0, "always", "rise", "passive"));
    // initialize function name cache
    forall F with find function(F) do update func_names(F) = name(F);
    // start trigger function_name_change
    print(startTrigger("function_name_change", 0, "always", "fall", "passive"));
}
```

```
}
```

In this example, the `func_names/1` ASM function is used as an associative array to store the names of the `function` elements. Since the RETE network can listen to ASM function value changes, as soon as the trigger's action sequence sets the appropriate value, the matches are updated.

5.5.4 Conclusion

In this case study, we have shown a technique to implement incremental code generation with the technology described in this paper. Incremental code generators enable the toolsmith to create a textual representation of the visual models the user is working with, and reflect the changes made by the various editing operations in real time. Such a facility is a very important feature for visual editing environments (such as user interface designers, or model-based frontends for textual programming environments), and it has been lacking in domain-specific language engineering tools up to now.

With our novel approach, an incremental code generator can be easily implemented using the extended high-level transformation language of VIATRA2. Making use of the underlying incremental pattern matching technology, the system can detect the changes in the domain model and invoke atomic generation operations which adjust the textual output incrementally. Moreover, since all partial matching are cached in the RETE network, the generation process itself is executed an order of magnitude faster than conventional batch-mode model traversing generators.

5.6 Summary

In Section 5.2 a detailed presentation about the current challenges in the domain-specific modeling world is presented. In the following three sections, we presented three detailed examples. Furthermore, alongside the examples, we proposed important conceptual results.

In the first example, in Section 5.3, we proposed a new diagram mapping mechanism, which enables arbitrary abstract-concrete syntax mapping. Our event-driven transformation engine was used to incrementally synchronise the abstract-concrete syntax.

In Section 5.4, we 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. On-the-fly incremental checking of complex constraint is not present in any of them.

In Section 5.5, we demonstrated, that with the use of triggers, incremental code generators can be written for the ViatraDSM framework. No other DSM tool provides the possibility of incremental code generation.

Chapter 6

Summary

6.1 Overview

In this paper we proposed an effective **incremental model transformation** technology, extended the VIATRA2 model transformation framework with this functionality, and applied it in the Via-traDSM domain specific modeling environment.

Despite the significant demand, currently there is no satisfactory support for efficient incremental synchronisation of models in any state-of-the-art model transformation systems; as we have shown, our event-driven transformation engine provides a solution for this problem. Graph rewriting tools are known to have difficulties with ALAP-type transformations; our incremental pattern matcher eliminates this obstacle.

Evaluating complex domain-specific well-formedness constraints while interactively editing a model poses a serious problem to current domain specific modeling frameworks; our event-driven transformation technique, however, achieves this goal. The synchronisation of the abstract and concrete syntax representations of the domain models is also greatly enhanced by our work.

6.2 Scientific contributions

- We proposed an efficient **incremental pattern matcher** that stores partial and complete pattern occurrences and updates them incrementally on modifications to the model, based on the **RETE algorithm**.
- Based on the incremental pattern matcher, we designed a methodology for **event-driven model transformations**, where actions are **triggered** when conditions and constraints become satisfied or violated in the model.
- We proposed a method for incremental, **on-the-fly checking** of complex well-formedness constraints during modeling.
- We proposed a method for the **incremental synchronisation** of logical models and their graphical representations in a domain specific modeling environment.
- We proposed a method for incremental **code generation** with the use of triggers.

6.3 Practical accomplishments

- We **implemented** the proposed components of incremental model transformation technology in the VIATRA2 model transformation framework, to provide support for truly incremental change propagation. In total **3700** lines of java code was written.
- We **conducted measurements** and concluded that the implemental pattern matching engine was **efficient**.
- We **applied** the proposed domain modeling improvements in the ViatraDSM domain specific modeling framework to address the outlined issues with domain specific modeling.

6.4 Future work

There are still no QVT-compliant transformation engines that can perform incremental updates truly efficiently; we plan to extend our engine to be able to execute QVT transformations.

One obvious point where there is plenty of room for improvement is the incremental pattern matcher. We plan to devise and apply various optimizations to the incremental pattern matcher and measure the benefits, as well as implement an alternative incremental pattern matcher based on a different principle (e.g. LEAPS), and compare the results. We also plan to extend our benchmark suite to further explore the problem space where an incremental engine is more efficient than an ordinary pattern matcher.

To ease the formulation of triggers, we plan to extend our event-driven model transformation engine to support triggers that were defined by the common left hand side - right hand side graph transformation formalism.

We plan to improve the ViatraDSM framework, by extending it with automatic trigger generation for special constraint types. We also plan to analyze which OCL constraints can be expressed by triggers.

Bibliography

- [1] The Eclipse project. 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] Don Batory. The LEAPS algorithm. Technical Report CS-TR-94-28, 1, 1994.
- [4] Bernd Kolb, Markus Völter. openArchitectureWare and Eclipse. <http://architekturware.sourceforge.net/data/oawEclipse.pdf>.
- [5] E. Börger and R. Särk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [6] Horst Bunke, Thomas Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 1990.
- [7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 22: Data Structures for Disjoint Sets. MIT Press/McGraw-Hill, 1990.
- [8] Jósvai E. and Tóth D. Grafikus felhasználói felületek modell-alapú fejlesztése. Scientific students' associations report, Budapest University of Technology and Economics, 2005.
- [9] 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.
- [10] 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.
- [11] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*.
- [12] Mark Proctor et al. *Drools Documentation*. JBoss. <http://labs.jboss.com/drools/documentation.html>.

- [13] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.
- [14] Rubino Geiß, Christoph Mallon, and Moritz Kroll. Sierpinski triangle for the agtive 2007 tool contest, July 2007. <http://www.informatik.uni-marburg.de/swt/active-contest/>.
- [15] 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.
- [16] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2006.
- [17] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios . Technical Report TR-RI-07-428, 2007.
- [18] 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.
- [19] Metacase. MetaEdit+. <http://www.metacase.com/mep/>.
- [20] Microsoft. DSL Tools. <http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx>.
- [21] D. P. Miranker and B. J. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, 1991.
- [22] P. Pandurang Nayak, Anoop Gupta, and Paul S. Rosenbloom. Comparison of the Rete and Treat production matchers for Soar. In *National Conference on Artificial Intelligence*, pages 693–698, 1988.
- [23] 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.
- [24] Object Management Group. *Action Semantics for the UML*, August 2001. <http://www.omg.org>.
- [25] Object Management Group. *Model Driven Architecture — A Technical Perspective*, September 2001. <http://www.omg.org>.
- [26] Object Management Group. *Object Constraint Language Specification (in UML 1.4)*, 2001. <http://www.omg.org>.
- [27] Object Management Group. *Meta Object Facility Version 2.0*, 2003. <http://www.omg.org>.
- [28] Object Management Group. *UML Semantics Version 2.0*, May 2003. <http://www.omg.org>.
- [29] The Eclipse Model Development Tools project. Official homepage. <http://www.eclipse.org/modeling/mdt/?project=ocl#ocl>.

- [30] The SENSORIA Project. The sensoria case tool tracker site. <http://svn.pst.ifi.lmu.de/trac/sct>.
- [31] The SENSORIA Project. The SENSORIA website. <http://www.sensoria-ist.eu>.
- [32] The SENSORIA Project. The sensoria case tool brochure, 2007. <http://www.sensoria-ist.eu/files/BI-sheet-english-7w.pdf>.
- [33] István Ráth. Declarative specification of domain specific visual languages. Master's thesis, Budapest University of Technology and Economics, 2006.
- [34] 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.
- [35] 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.
- [36] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
- [37] 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.
- [38] The Eclipse Project. Graphical Modeling Framework. <http://www.eclipse.org/gmf>.
- [39] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
- [40] 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.
- [41] 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.
- [42] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. Technical report, Budapest University of Technology and Economics, 2005. <http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf>.
- [43] Gergely Varró, Dániel Varró, and Andy Schürr. Incremental graph pattern matching: Data structures and initial experiments. In Gabor Karsai and Gabi Taentzer, editors, *Graph and Model Transformation (GraMoT 2006)*, volume 4 of *Electronic Communications of the EASST*. EASST, 2006.
- [44] Ian Wright and James Marshall. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation*, 2(1):36–48, February 2003.
- [45] Xifeng Yan and Jiawei Han. gSpan: graph-based substructure pattern mining. In *ICDM*, pages 721–724. IEEE Computer Society, 2002.

Index

- abstract syntax, 10
- action sequence, 57
- actual contents, 37
- alpha node, 30
- ASM function, 19

- beta node, 30

- cancellation, 59
- check condition, 15
- concrete syntax, 10
- conflict set, 29
- ConstantNode, 37
- constraint pool, 45
- control options, 57
- current stub, 45
- current variable tuple, 45

- delta monitor, 64
- delta set, 59
- Diagram, 27
- DiagramElement, 27
- direct containment root, 36
- Domain-Specific Modeling, 9

- Edge, 25
- EdgeFigures, 27
- Editor, 24
- editor generation, 24
- entity, 12
- entity root, 35
- EqualityNode, 39
- event, 57

- footprint, 45

- graph pattern matching, 15

- important update, 38
- Indexer, 38
- InequalityNode, 39
- input node, 29
- instanceOf, 13

- JoinNode, 39

- left-hand side, 16

- memory, 30
- Mode, 57
- Model Driven Architecture, 6
- multiplicity, 13

- native function, 19
- negative update, 30
- Node, 24
- node sharing, 52
- NodeFigures, 27
- NotNode, 39

- pattern mask, 35
- Plugin View Classes, 24
- positive update, 30
- postcondition, 17
- precondition, 17
- Priority, 57
- production node, 30
- Properties, 25
- pull operation, 37

- recursive pattern, 16
- relation, 12
- relation root, 35
- right-hand side, 16
- runtime framework, 24

- semantical contents, 37
- Sensitivity, 57
- signature, 35
- Startup, 58
- supertypeOf, 13

- term-substitute, 43
- TermEvaluatorNode, 41
- transaction handling, 64
- transitive containment root, 36
- trigger, 57

TrimmerNode, 41

tuple, 35

tuple inheritance, 52

uniqueness principle, 35

UniquenessEnforcerNode, 41

variable assignment, 45

VIATRA, 12

ViatraDSM, 22

virtual variables, 45

WME, 29