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

Incremental graph pattern matching and applications

Master's Thesis

Bergmann Gábor

Supervisors:

Dr. Dániel Varró, assistant professor
István Ráth, PhD student

Budapest, 16 May 2008

I would like to thank the supervisors István Ráth and Dr. Dániel Varró for their continued support, friendly advice, and enthusiasm. I would also like to thank Gergely Varró for his helpful ideas and extensive research; Ákos Horváth for helping me with many VIATRA2 *related suggestions; and Dániel Tóth for his numerous valuable observations.*

Nyilatkozat

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

Bergmann Gábor

Kivonat

Napjainkban a modellbázisú rendszerfejlesztési paradigma térhódítása figyelhető meg. Gyakorlati megvalósításainak legfontosabb eleme a tervezett rendszer modelljeinek különböző absztrakciós szintű megjelenítésére és azok közötti leképezésre alkalmas modelltranszformációs rendszer. A modelltranszformációs feladatra többféle technológiát alkalmazhatunk; a gráftranszformációra alapuló módszerek széleskörű alkalmazhatósága és vizuális jellege mára ipari és akadémiai eszközök sokaságát hívta életre, melyek közé tartozik a Méréstechnika és Információs Rendszerek tanszéken fejlesztett és több kutatási projektben alkalmazott VIATRA2 modelltranszformációs keretrendszer.

E gráftranszformációs eszközökben azonban a hagyományos megközelítés szerint a transzformációs szabályok feltétel részének (és általában minden gráfmintának) az illeszkedéseit minden alkalommal meg kell keresni, amikor szükség van rájuk, amely költséges művelet. A diplomaterv annak a kérdését vizsgálja, hogy miként lehet inkrementális mintaillesztéssel javítani a hatékonyságot, amikor a minták illeszkedéshalmazát a modelltér minden megváltoztatásakor lépésről lépésre karbantartjuk.

A dolgozat bemutatja a szabály alapú szakértői rendszerek területén sikerrel alkalmazott RETE inkrementális mintaillesztő algoritmust. Ezek után megvizsgálja, milyen elméleti és gyakorlati megfontolásokkal alkalmazható ez a megközelítés gráftranszformációs környezetben, és adaptálja a RETE algoritmust a VIATRA2 gazdag gráfminta-nyelvének feldolgozására. Az így elkészült inkrementális mintaillesztő modul beépült a transzformációs keretrendszerbe, és a VIATRA2 közelgő harmadik kiadásának része lett.

A fentieken túl a dolgozat megvizsgálja azokat az elméleti kérdéseket, amelyeket a mintaillesztő párhuzamosíthatósága felvet, a terjedő többmagú processzorok jobb kihasználásának érdekében. A félév során elkészült egy ennek megfelelő többszálú implementáció is. A kidolgozott rendszer helyes működését formális bizonyítás szavatolja.

A dolgozat mérésekkel igazolja a megvalósított mintaillesztő hatékonyságát, összehasonlítva azt a VIATRA2 rendszer eredeti mintaillesztő moduljával, valamint az egyik leggyorsabbnak tartott modelltranszformációs eszközzel (GrGen.NET). A mérések rávilágítanak, hogy bizonyos problémaosztályok kezelése nagyságrendileg hatékonyabb lehet inkrementális mintaillesztéssel.

A diplomaterv a megszületett elméleti és gyakorlati eredmények értékelésével zárul.

Abstract

Nowadays the model-based software development paradigm has gained significant popularity. The most important component of its realisation is a model transformation system capable of mapping between different abstraction levels of the system under design. Several technologies are available for model transformation, due to their widespread applicability and visual nature, methods based on graph transformation (graph rewriting) have given birth to a variety of academic and industrial tools; one of them is the VIATRA2 transformation framework which is developed at the Department of Measurement and Information Systems and is used in various research projects.

According to traditional graph transformation approaches, the matchings of the condition parts of transformation rules (and, more generally, any graph pattern) have to be searched for whenever they are needed, which is a costly operation. The thesis seeks to improve performance with incremental pattern matching, where occurrence sets of graph patterns are continuously maintained and updated after each change to the model space.

For the task of incremental pattern matching, the paper introduces the RETE algorithm, that is well-known in the field of rule-based expert systems. The paper examines conceptual and practical considerations in applying this approach to a graph transformation environment, and adapts the RETE algorithm to processing the rich graph pattern language of VIATRA2. The Incremental Pattern Matcher module has been developed and integrated into VIATRA2, and became one of the features of the upcoming third release of VIATRA2.

To harness the power of the increasingly popular multi-core processors, the thesis investigates the conceptual questions of exploiting parallelism in model transformation and graph pattern matching. A multi-threaded implementation conforming to these principles was also developed this semester. A formal proof is presented for the correctness of the design.

Measurements confirm the efficiency of the implemented pattern matcher, comparing it to the original pattern matcher module of VIATRA2, as well as one of the fastest model transformation systems (GrGen.NET). These measurements show that certain problem classes of graph transformations can be executed several orders of magnitude faster using an incremental pattern matcher.

The thesis concludes with the evaluation of conceptual and practical results.

Contents

Chapter 1

Introduction

This chapter provides an introduction on model based software engineering, the need for model transformation systems like VIATRA2, and challenges in this field that serve as a motivation for this work, along with its objectives and structure. The wording of this chapter is largely based on several previous publications of the VIATRA2 team.

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* [28] to provide a visionary paradigm of software development. MDA relies OMG's other flagship technologies: Unified Modeling Language (UML) [31], the Meta Object Facility (MOF) [30], 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.

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

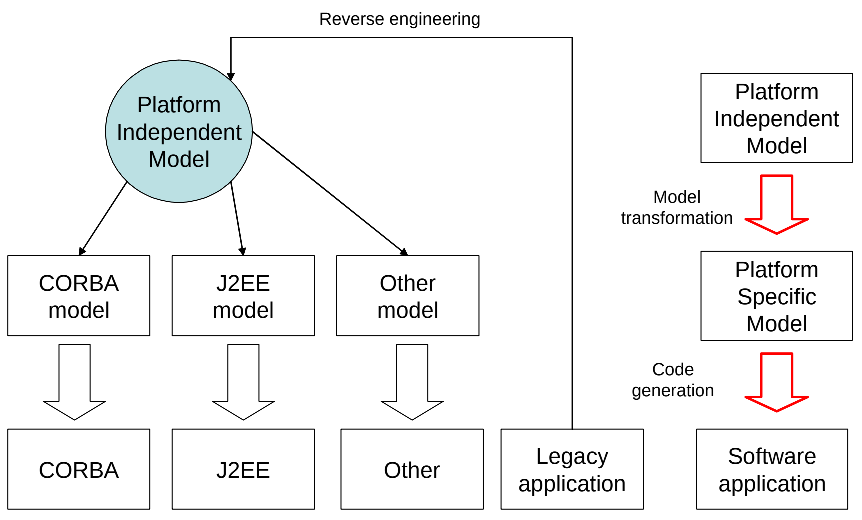


Figure 1.1: Model Driven Architecture

modeling concepts which are not platform specific. The PIM is a pure UML model, with constraints specified in the Object Constraint Language (OCL) [29], and behavioral semantics described in Action Semantics (AS) [27] 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 considerable 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 [39]:

- 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.

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 [34] 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.). [37]

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

1.3 Problems and challenges in model transformations

As 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.

Another scenario, the simulation of executable models is also introduced. Both tasks suffer from performance problems, which can be solved by taking a different approach.

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 [41] for measurements.

To be precise, two aspects of the synchronisation procedure can be incremental:

incremental target update refers to the practice that changing the source model does not involve recalculating the whole target model; only the changes will have to be transformed and applied to the target. If something is already transformed, it will not be transformed again.

incremental computation refers to the principle that calculated results should not be abandoned and recalculated later. In particular, it refers to doing the computation of the transformation itself incrementally, i.e. after changing the source model, the necessary alterations to the target model become easily apparent without having to search through the entire source and/or target model, because that searching has been performed before and the relevant results are cached.

Of the above two, the first aspect is already achievable with most transformation environments, but the second one is not. 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. This is one reason why incremental pattern matching can be a valuable contribution, as it is roughly an efficient cache mechanism that can be utilised to provide the set of precondition occurrences efficiently.

1.3.2 Model simulation

The dynamic behaviour of models can be studied with model-based simulations. This approach has heavy industrial applications, used in various engineering disciplines, supported by commercial software systems [2]. The same principle can also be applied to software engineering; for instance, BPEL4WS[19] is an executable model language for describing business processes interacting with and enclosed by web services. On the other hand, the success of formalisms such as UML is due to the fact that models used during software engineering are often easily represented as graph models, and state transitions as graph rewriting. Therefore, model-based simulation of graph models is an important task.

The general characteristics of graph model simulations (e.g. Petri-net firings) are large models, discrete time execution and small portions of the graph being altered during each step. This problem class involves a performance difficulty: after each step, the set of possible successor states have to be re-evaluated, which can be costly if the model graph is large. Due to this, [6] proposes Petri-net simulations as a benchmark problem for graph transformation systems.

The performance can be greatly improved if the re-evaluation of possible steps can be made cheaper. A proposed solution is continuously maintaining the occurrence sets of the sought patterns and subpatterns incremental. After each step, these occurrence will have to be updated according to the changes since the previous state of the model. This can be considerably more efficient than repeatedly searching these patterns in the model, which is another argument in favor of incremental pattern matching.

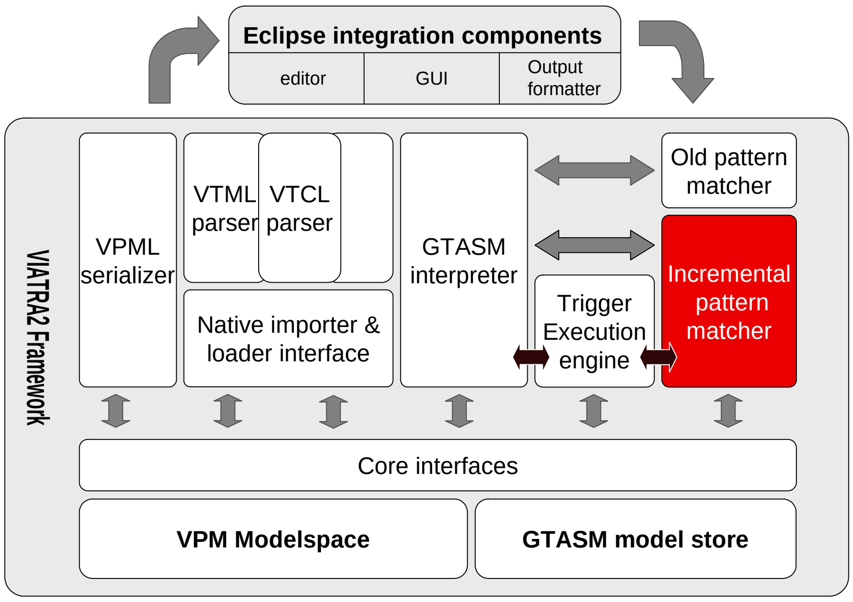


Figure 1.2: VIATRA2 R3 Framework extended with the Incremental Pattern Matcher

1.4 Objectives

In this thesis I propose an effective pattern matching approach and apply it gain performance benefits in model transformation and model simulation.

The detailed objectives are the following:

- I propose an efficient **incremental pattern matcher** that stores partial and complete pattern occurrences and updates them incrementally on modifications to the model.
- I **adapt** the theoretical RETE algorithm to the problem domain of model transformations.
- I **extend** the basic feature set of RETE to accommodate the rich pattern language of VIATRA2.

- I **implement** the proposed components of incremental model transformation technology in the VIATRA2 model transformation framework, as shown on Figure 1.2.
- I **investigate** the idea of a concurrent and multi-threaded pattern matcher, taking steps to a parallelised model transformation system, and implement the updated design.
- I **measure** the efficiency of the implemented pattern matcher.

1.5 Structure of the Thesis

In Chapter 2, I introduce the technical and theoretical background of my thesis. The first three sections are dedicated to VIATRA2, the model transformation framework utilized by my work. 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. The next part gives a brief overview of related work, surveying the state-of-the-art of pattern matching in graph transformation.

Chapter 4, elaborates this incremental approach further by designing a parallelism-enabled version of the pattern matcher that can benefit from multi-core processors found in modern desktop computers. The sections of the chapter deal with concurrent transformation execution and pattern matching; designing a multi-threaded pattern matcher and formally proving its correctness; discussing the possibilities of distributed pattern matching; and preparing the pattern matcher to provide service for multiple concurrent transformations.

In Chapter 5, the quantitative benefits of the incremental pattern matching approach are measured, evaluated and compared against other solutions. First, the benchmark problems are introduced. Then, using these benchmarks, the performance of the Incremental Pattern Matcher module is measured and compared to the original pattern matcher of VIATRA2 and also a different pattern matching system. The benefits of certain optimization techniques applied to RETE are confirmed using a synthetic benchmark.

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

Chapter 2

Context

This chapter introduces the technical and theoretical context of this thesis. VIATRA2, a model transformation framework, serves as the basis for the pattern matcher proposed in this paper. Section 2.1 introduces the modeling paradigm associated with VIATRA2. The pattern and transformation language of the framework is described in Section 2.2. Section 2.3 outlines the structure and the most important features of VIATRA2. Finally, Section 2.4 gives an introduction on the RETE algorithm, which is the theoretical background of my work.

2.1 Metamodeling in VIATRA2

VIATRA (VIsual Automated TRAnsformations) 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. This Section conceptually follows [3], and is not a work of the author.

2.1.1 Visual and Precise Metamodeling

Currently, most widely used metamodeling languages (e.g. ECore) are derived (with slight variations) from the Meta Object Facility (MOF) [30] metamodeling standard issued by the OMG. However, as stated in [39], 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) [39] 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

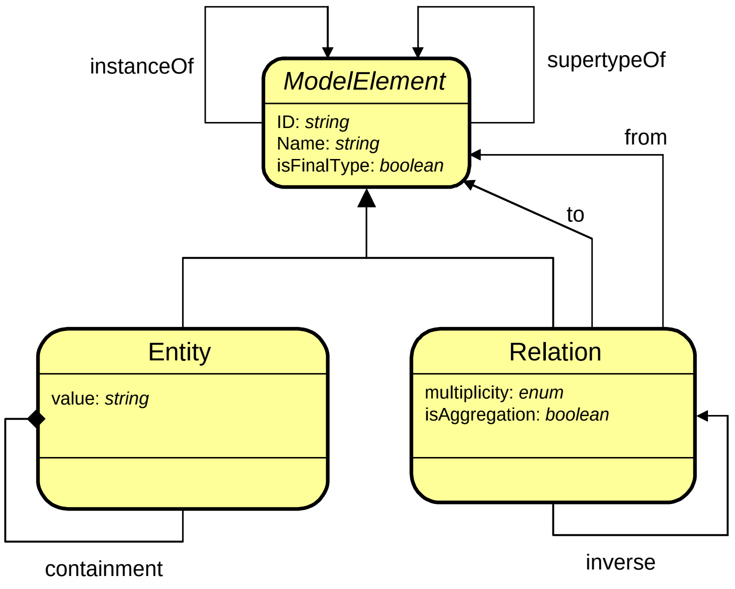


Figure 2.1: The VPM Metamodel

between other model elements[1]. Furthermore, entities may also have an associated value which is a string that contains application-specific data.

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

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

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

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

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

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

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:

$$instanceOf(a,b) \wedge subtypeOf(b,c) \Rightarrow instanceOf(a,c)$$
$$subtypeOf(a,b) \wedge subtypeOf(b,c) \Rightarrow subtypeOf(a,c)$$
$$instanceOf(a,b) \wedge instanceOf(b,c) \not\Rightarrow instanceOf(a,c)$$

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 [12].

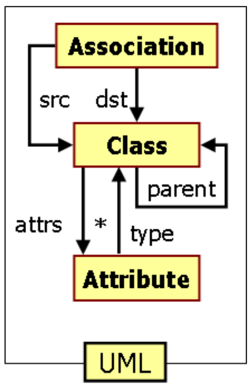


Figure 2.2: Sample UML metamodel

The VTML equivalent of the metamodel is as follows.

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

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

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

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

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

2.2 Transformations in VIATRA2, 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) [11] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [8] 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.

This Section conceptually follows [3], and is not a work of the author.

2.2.1 Graph patterns

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

A model (i.e. part of the model space) 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 [38]. 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 condition*s that are Boolean formulae which must be satisfied in order to make the pattern true. The example pattern checks 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 [33].

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

2.2.2 Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [11], which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its *left-hand side* (LHS) pattern with an image of its *right-hand side* (RHS) pattern.

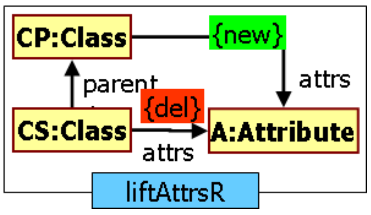


Figure 2.3: Sample graph transformation rule

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 *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 [26]) 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, the rule defines 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 the above sample rule.

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

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

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

2.2.3 Control Structure

To control the execution order and mode of graph transformation the VTCL language includes some concepts that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, the basic set of Abstract State Machine (ASM) language constructs [8] were included; they 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 function*s. ASM functions are special mathematical functions, which store values in arrays. These values can be updated from the ASM program. These functions are called *dynamic*. There are also *static* functions, which means that they cannot change their values. For example, the basic mathematical functions (+,-,*,/) are static.

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

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

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

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

2.3 VIATRA2 Architectural overview

This Section conceptually follows [3], and is not a work of the author. The VIATRA2 system is a standalone model container and transformation framework, which can be integrated into the Eclipse IDE as a plug-in. In stand-alone mode, the VIATRA2 system runs as a console application with a command-line console.

Within the Eclipse environment, additional integration components are available:

- a tree-view model space editor component, supporting the standard *Properties* view and undo-redo functionality;

- an Eclipse *view* which provides an interface to the import/export/parser facilities;
- a Code output view component to visualize the textual output generated by code generators.

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

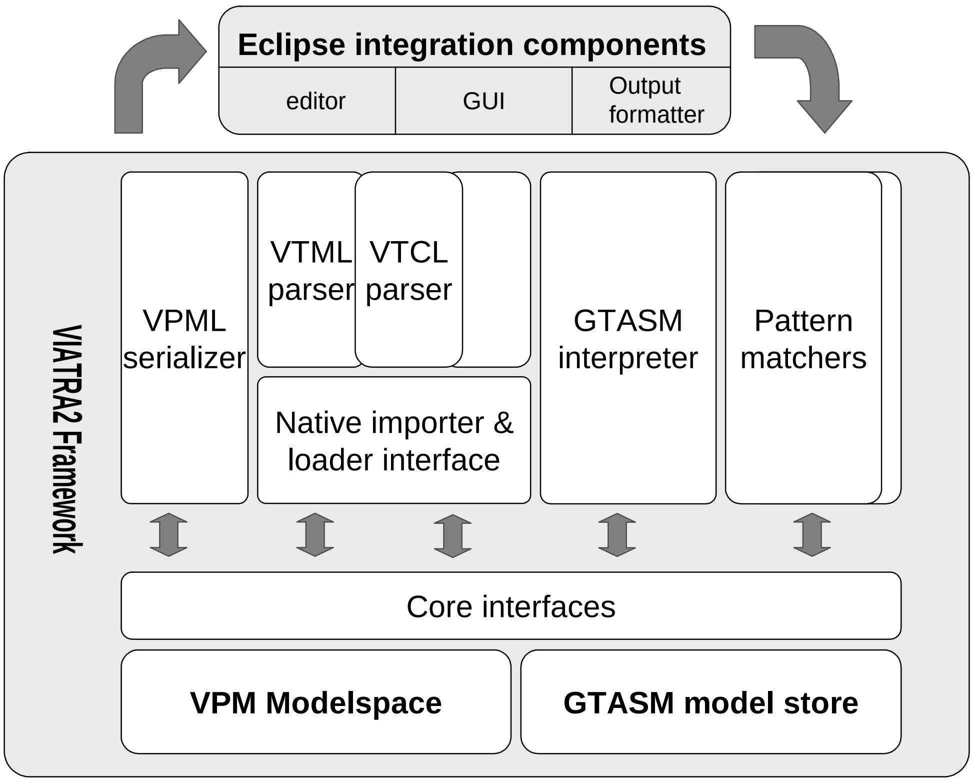


Figure 2.4: The architecture of the VIATRA2 framework

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

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.

2.4 RETE networks

2.4.1 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 [14]) 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, [13]), RC++ [44], Jess.

2.4.2 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.

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

Although in practice, there are several variations of the basic idea, an overview of commonly encountered features of RETE can be given here.

RETE networks contain nodes of various types. There is a distinguished set of nodes (usually one single node) called *input node*s that contain the asserted facts of the knowledge base. So-called *alpha node*s 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 node*s, that have two separate input slots, each connected to a node[3] 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 edges[4] 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.

2.4.3 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

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

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

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.

2.4.4 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 bilabial[5] 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 2.5. 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 2.5 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.

[5]bilabial consonants are articulated with both lips

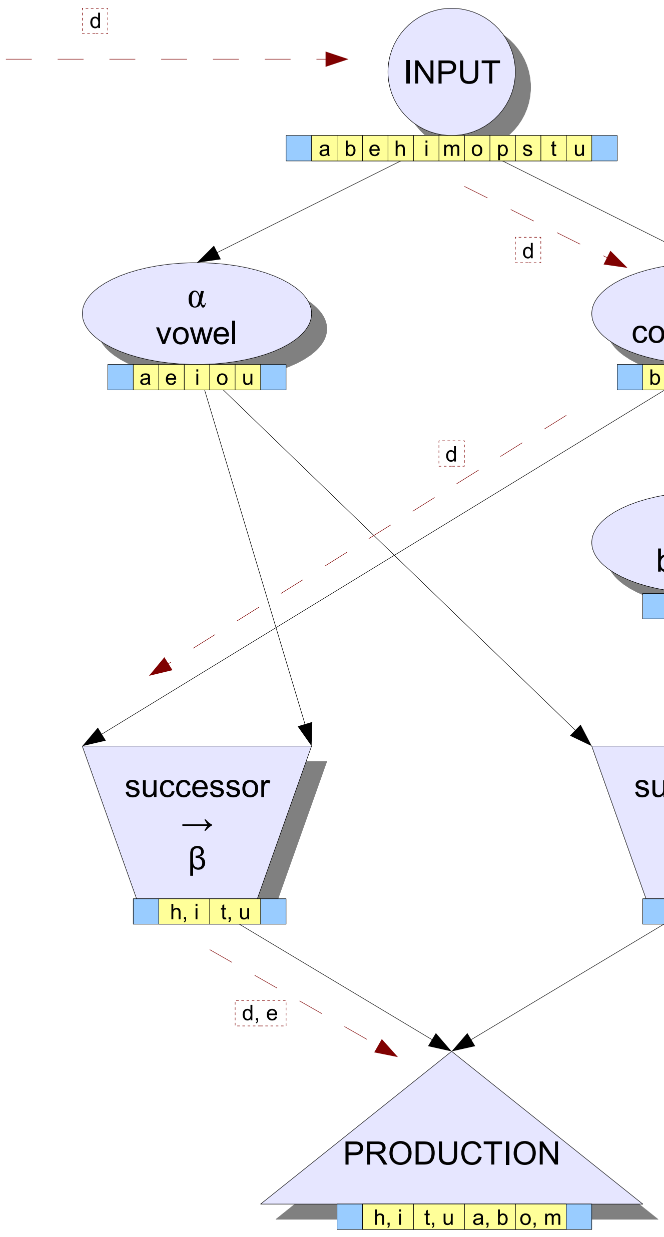


Figure 2.5: Consonants and vowels - a sample network illustrating the basic RETE concept

2.4.5 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 [23] 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 ([25] 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 occurences 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.

[17] presents a pattern matcher that builds and maintains Prolog-like SLD 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. Changes to the model are reflected by asserted or retracted clauses; they effects are incrementally applied on the resolution tree. In order to increase performance, some heuristical considerations on the ordering of predicates have also been presented. While the approach does not seem very efficient at an initial glance, this uniform treatment of facts and patterns can be considered a unique, advanced feature of the approach, giving rise to an interesting feature: the patterns can also be changed at runtime, resulting in incremental updates to the conflict set. This approach is utilised by the TefKat model transformation tool.

[4] 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 an inserted 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.

While there are many interesting candidate solutions, RETE was tried and well-known, flexible, easy to implement, and reliable enough to be chosen as the basis for the research. Experimenting with alternatives is still among future plans.

Chapter 3

Incremental pattern matching with the RETE algorithm

3.1 Goal of this chapter

In order to provide an alternative way of pattern matching in the VIATRA2 framework, the author has developed an incremental pattern matching engine. This chapter describes the conceptional results in applying the RETE network principle in a VIATRA2 environment, the practical results in building RETE networks for the incremental matching of GTASM patterns, and the efforts at improving the performance of the new pattern matcher.

The greatest benefits of incremental pattern matching are expected when pattern matching queries are frequent and only minor changes are made to the model inbetween. The iterate-choose loop of the VIATRA2 transformation language conforms well to this is usage pattern, the forall construct less so.

3.2 Applying the RETE concept on VPM model spaces

I have introduced the basics of RETE networks in Section 2.4. 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 model space contains relationships and properties of model elements, the contents of the RETE network must reflect this information. Therefore my RETE implementation employs *tuple*s of model elements[1] as working memory elements. The graph-like semantics required to describe the contents of the VIATRA2 model space can be represented this way; the idea stems from [9] where a similar approach is used for graph pattern matching.

[1] actually, references to model elements

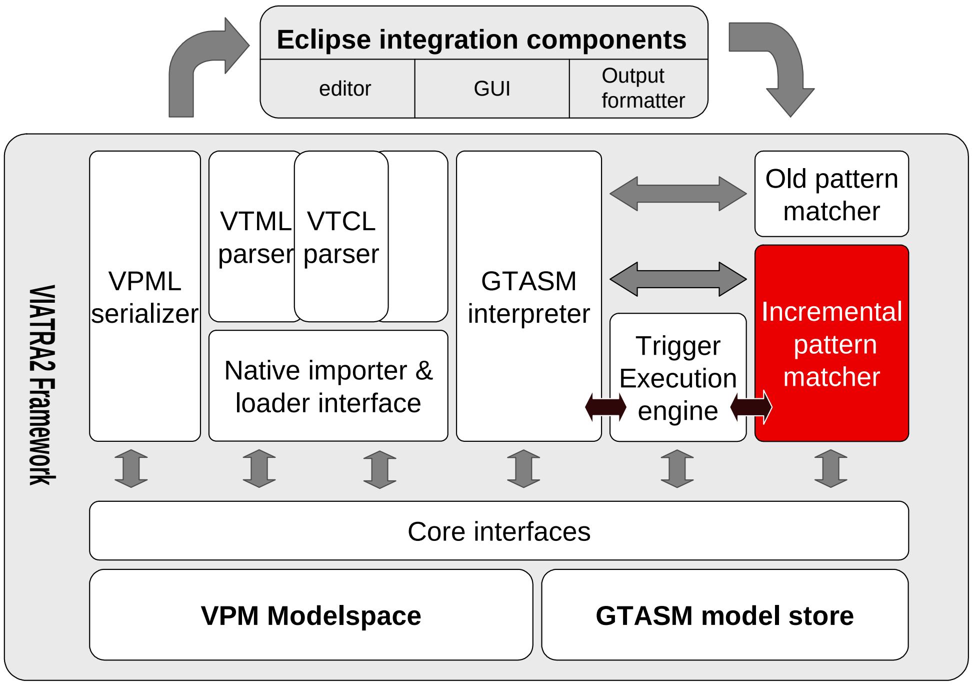


Figure 3.1: VIATRA2 R3 Framework extended with the Incremental Pattern Matcher

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, the *pattern mask* is defined as an array of indices, suitable for remapping (transforming) tuples, as described below. If a tuple ϕ is transformed by a mask μ, the result is defined as 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 network[2]; all information regarding the model space 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 model space broadcasts notifications of events such as creation or deletion of entities or relations. Input nodes receive update tokens by subscribing to these VIATRA2 model space 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 root*s belong to relation types, their tuples have three elements (the relation itself as a model element, source entity, destination entity).

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 and relationship roots

A single node called *direct containment root* is responsible for the containment hierarchy of the model space; 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, a *transitive containment root* is defined 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 Section 3.3) for a recursive pattern describing the transitive closure of direct containment.

The type-generic pattern features of VIATRA2, namely in-pattern supertype-of and instance-of relationships, is also supported by the Incremental Pattern Matcher. A very similar structure is used to derive the *transitive supertype-of root* from a root node containing elementary supertype relationship edges; the latter root node is hidden as only transitive supertyping is accessible from generic patterns. Instance-of checks also require an internal root node for elementary insatnce-of relationship edges, and the derived *instance-of root*, that is not exactly a transitive closure: it contains elementary instance-of relations prolonged by a supertype-of chain of arbitrary length.

[2] with the exception of term evaluators, see Section 3.2.4

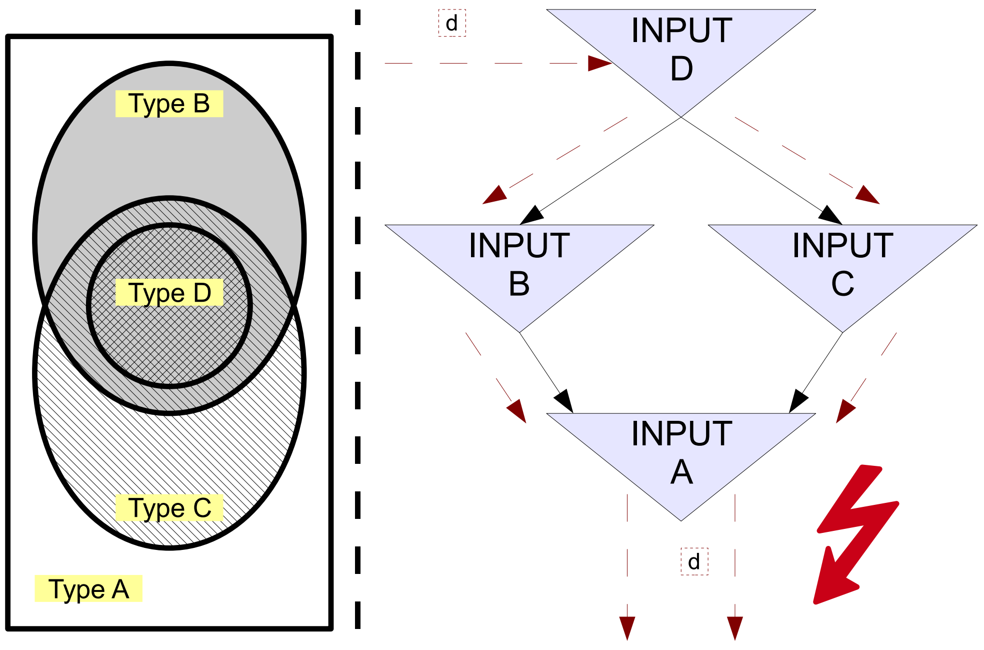


Figure 3.2: a problematic type hierarchy and the corresponding input nodes; the root node for type A must suppress duplicate updates

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 33) would be violated. Therefore these root nodes are UniquenessEnforcerNodes 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 estabilished procedure to get rid of the type associated with it.

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 Section 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, *ConstantNode*s 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.

*Indexer*s are special nodes associated with a pattern mask (see 3.2.1, page 33). 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

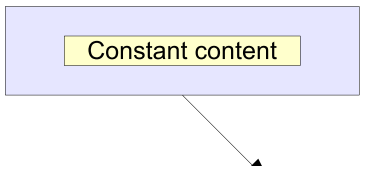


Figure 3.3: Graphical notation of ConstantNode

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, then it is considered 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.

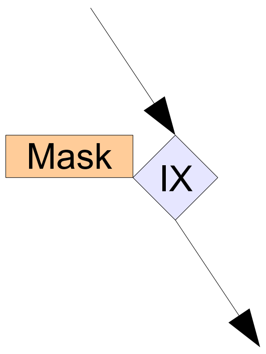


Figure 3.4: Graphical notation of Indexer

Dual input (beta) nodes

As an analogy to the beta node (see 2.4.2, page 27) 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.

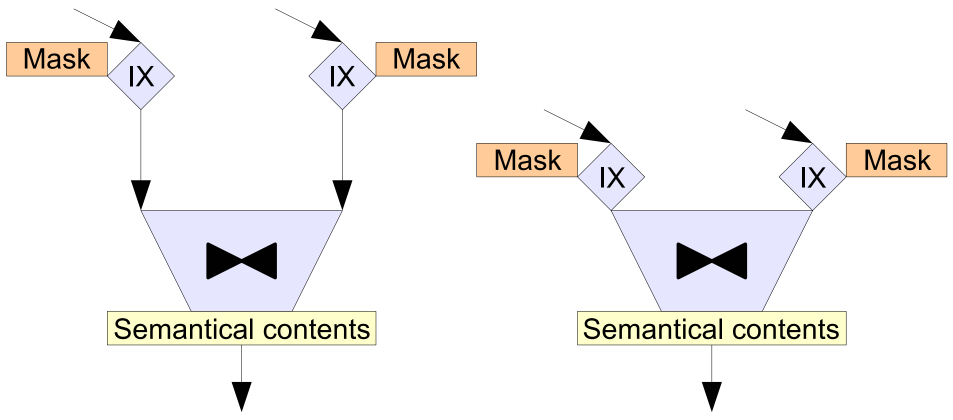


Figure 3.5: Graphical notation of JoinNode with Indexers as slots - concise form on the right

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 $\alpha \wedge \neg\beta$); the node actually conforms to the anti-join operation in relational algebra. 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.

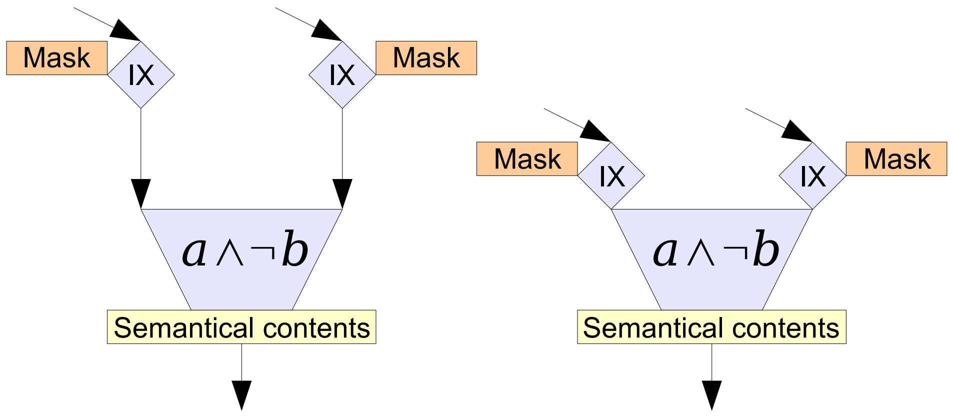


Figure 3.6: Graphical notation of NotNode with Indexers as slots - concise form on the right

Single-input filtering and transformation nodes

The *EqualityNode* and *InequalityNode* are types of alpha node (see 2.4.2, page 27). They check whether certain elements in the tuple, selected by a pattern mask associated with the node, are all equal (Equality-Node) 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 Section 3.3.3. See figures 3.7 and 3.8 for graphical notation.

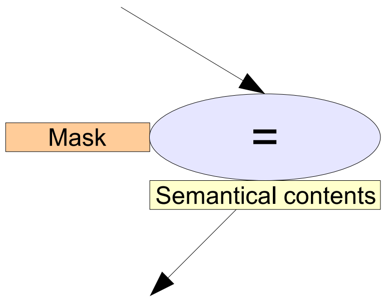


Figure 3.7: Graphical notation of EqualityNode

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

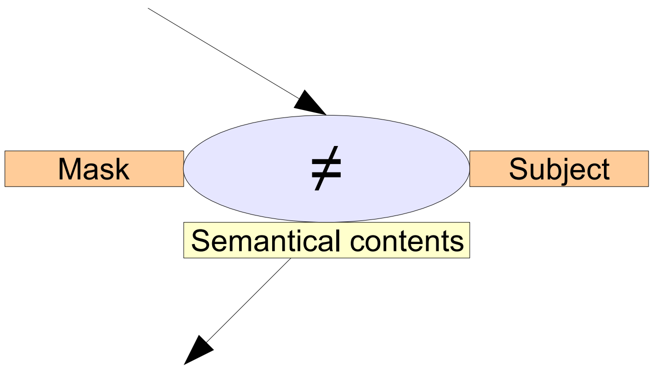


Figure 3.8: Graphical notation of InequalityNode

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 33). See Figure 3.9 for graphical notation.

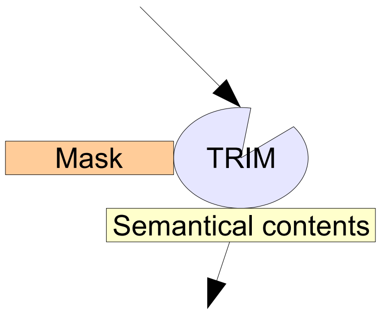


Figure 3.9: Graphical notation of TrimmerNode

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.

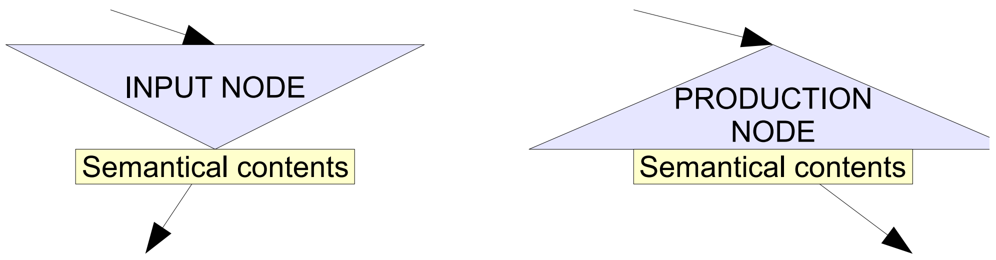


Figure 3.10: Graphical notation of root nodes; input node on the left, production node on the right

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 evaluates 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 model space containment hierarchy; or the values of referenced global permanent associative storages (ASM functions, see 2.2.3 on page 23). See Figure 3.11 for graphical notation.

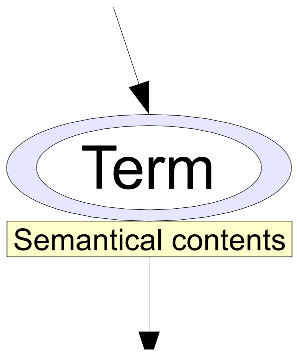


Figure 3.11: Graphical notation of TermEvaluatorNode

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 47) 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.

- 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 model space 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 form 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 model space change notifications.

The node is notified whenever the value of an invoked ASM function is changed, or an element in its subsciption 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 Asynchronous update propagation

The synchronous method and its disadvantages

My first RETE implementation used simple Java method calls to propagate update tuples between nodes; incoming updates induced outgoing updates in a depth-first fashion. This approach can be thought of as *synchronous* operation, because the processing of all updates leaving a particular node must be finished before the update that caused the activation of the node can be considered finished, transferring the control back to the parent node.

This initial approach has the following serious drawbacks:

- When changes to the model space are admitted into the RETE net as update tuples, the execution of the transformation must halt while the update is propagating in the network, obstructing parallelisation efforts (see Chapter 4).

- In some situations, the ordering of update messages are not preserved; if the deletion of an element is signaled before its creation, it could lead to incorrect results. This phenomenon will be discussed in more detail later.

Solution

The solution involves a *message queue* attached to the network, containing update messages manifested as objects. Each message object specifies a recipient node, the tuple representing the update, and the sign (positive/negative update). The message consumption cycle fetches the first message from the queue and delivers it to the appropriate node; the node will place any propagated output messages to the end of the queue, thereby achieving *asynchronous* messaging. Changes to the model are simply put at the end of the (then-empty) message queue; then the update propagation phase consists of looping the message consumption cycle until the queue becomes empty. This way, the consistent ordering of messages is preserved; furthermore, changes to the model can be reported quickly, thus the road is paved for the concurrent pattern matcher (see Section 4.2).

The pebbles game: a case study

Consider the following game: two players are given a common heap of pebbles. They take turns at removing some pebbles from the heap; detailed rules may govern how many pebbles can be removed, but that is uninteresting now, beyond the basic assumptions that at least one pebble is removed each turn and, of course, the number of pebbles cannot drop below zero. A player loses the game if they cannot make a move when it's their turn; with most rule variations, that means that there are no more pebbles left. This game can be formulated as a graph pattern matching problem: there is a separate 'situation' entity for every possible number of pebbles from zero to the initial size of the heap; an edge leads from one situation to another if it is a valid move to make (e.g. situation 10 is connected to situation 7 if it is permitted to take 3 pebbles from 10); finally, a situation is considered losing if no valid moves lead to another losing situation (for the other player), including the case when there are no valid moves at all.

The above definition specifies a well-founded negative recursion; this graph pattern can be expressed using VTCL notation the following way:

```
pattern losing(S) =
{
        situation(S);
        neg pattern cannotwin(S) = {
                situation(S);
                situation.succ(X, S, S2);
                situation(S2);
                find losing(S2);
        }
}
```

The recursion implies a cyclic RETE network; indeed, the production node of the 'losing' pattern is connected (through an Indexer) to one of the JoinNodes of the internal 'cannotwin' pattern, whose production node is connected to the NotNode of the enclosing pattern in turn. While the asynchronous solution experiences no problems with this configuration, the original synchronous version would deliver incorrect results, due to its depth-first nature.

3.2.6 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 Section 3.3.

Pattern queries

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 33), 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.

Live transformation

There is another possible application of the RETE network: if one registers an object as a child node to the production node, one can receive notifications whenever a new occurrence of the pattern is found or an old one is lost. Finding or losing a pattern occurrence can be considered as an event (defined by an arbitrary VIATRA2 pattern). Using the mechanism described above, event-driven reactions can be triggered. This idea is the basis of live model transformations, see [32], opening up many possibilities.

An obvious application would be continuously maintained synchronisation between source and target models. Whenever the source model is modified, the necessary changes are propagated to the target model (or vica versa, if needed). This does not only apply from manipulations through the user interface; changes performed by transformations or background processes (e.g. network discovery) are treated the same, as the key concept is the change of the model, not the performed action.

The same principle can enhance diagram editing by providing a live synchronisation between abstract and concrete syntax representations, consistently maintaining model and views, etc. Furthermore, complex, pattern-based well-formedness constraints can be checked on-the-fly and indicated during diagram editing. It is even possible to employ incremental code generation. The author has participated in writing [32], where these ideas are outlined in greater detail.

3.3 Building a RETE net from GTASM patterns

The VIATRA2 Incremental Pattern Matcher component utilises 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 2.2.1 on page 19) 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, 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. Elements of Γ are also called bound variables. The *footprint*[3] $F(\phi)$ of a constraint ϕ is defined 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 [10] 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.

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

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.
4. $\Gamma' := \Gamma \bigcup F(\alpha)$, 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.

It is easy to see that 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 common 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 36) 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 38) 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 36) 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 some of the actual parameters of the production or root node are supposed to equal each other. In this case, optional EqualityNode (see 3.2.3, page 39) instances are applied to the root node or production node before the joining to enforce that the specified 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:

```
        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.

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 38) 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: InequalityNodes are needed; the negative pattern, however, can only be checked when all elements in the footprint are already bound. Some actual parameters of the negative pattern call, however, are allowed to be free variables; they are not included in the footprint, since they need not and can not be bound; these variables do not appear anywhere else, they are unconstrained, and carry the $\nexists$ quantor of the negative pattern call. Figure 3.13 illustrates this configuration.

Checking GTASM terms is as simple as appending a TermEvaluatorNode (see 3.2.4, page 41) to

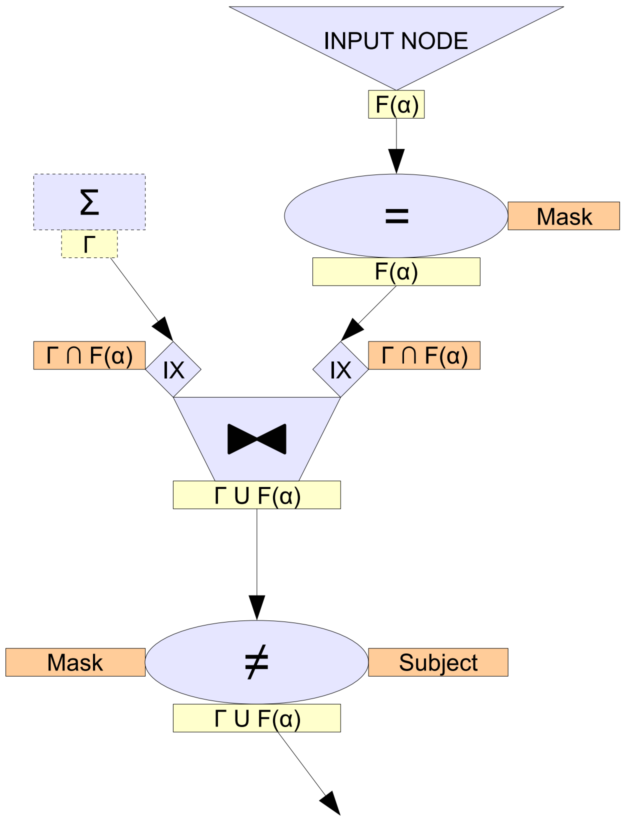


Figure 3.12: Node configuration for checking entity type, relation type, containment constraints and positive pattern calls

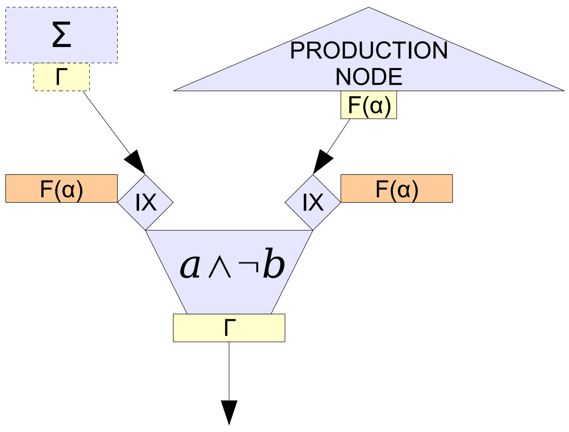


Figure 3.13: Node configuration for checking negative pattern calls

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.

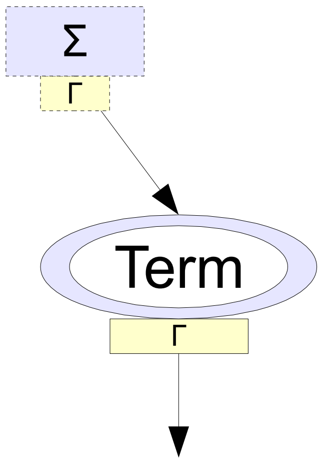


Figure 3.14: Node configuration for checking GTASM terms

The checkers described above have a limitation; its importance is dubious, it could even be called obscure, as the author and coworkers 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 "sandwitched" 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 39), 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 40) that is a child of the terminating TrimmerNodes of the pattern bodies.

3.3.4 Constraint ordering

I 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 combinatorical 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 combinatorical 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, the (now empty) ConstantNode that should have been created in step 1 has been omitted, and the figure starts with the first entity checker instead; the InequalityNodes that are supposed to follow the JoinNodes have also been omitted. 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 Improvements and optimization ideas

In this section, I discuss various techniques to improve the performance of the pattern matcher component described in the previous sections.

3.4.1 Node sharing

General idea

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 idea, however, is actually quite difficult. Identifying large similar subpatterns that

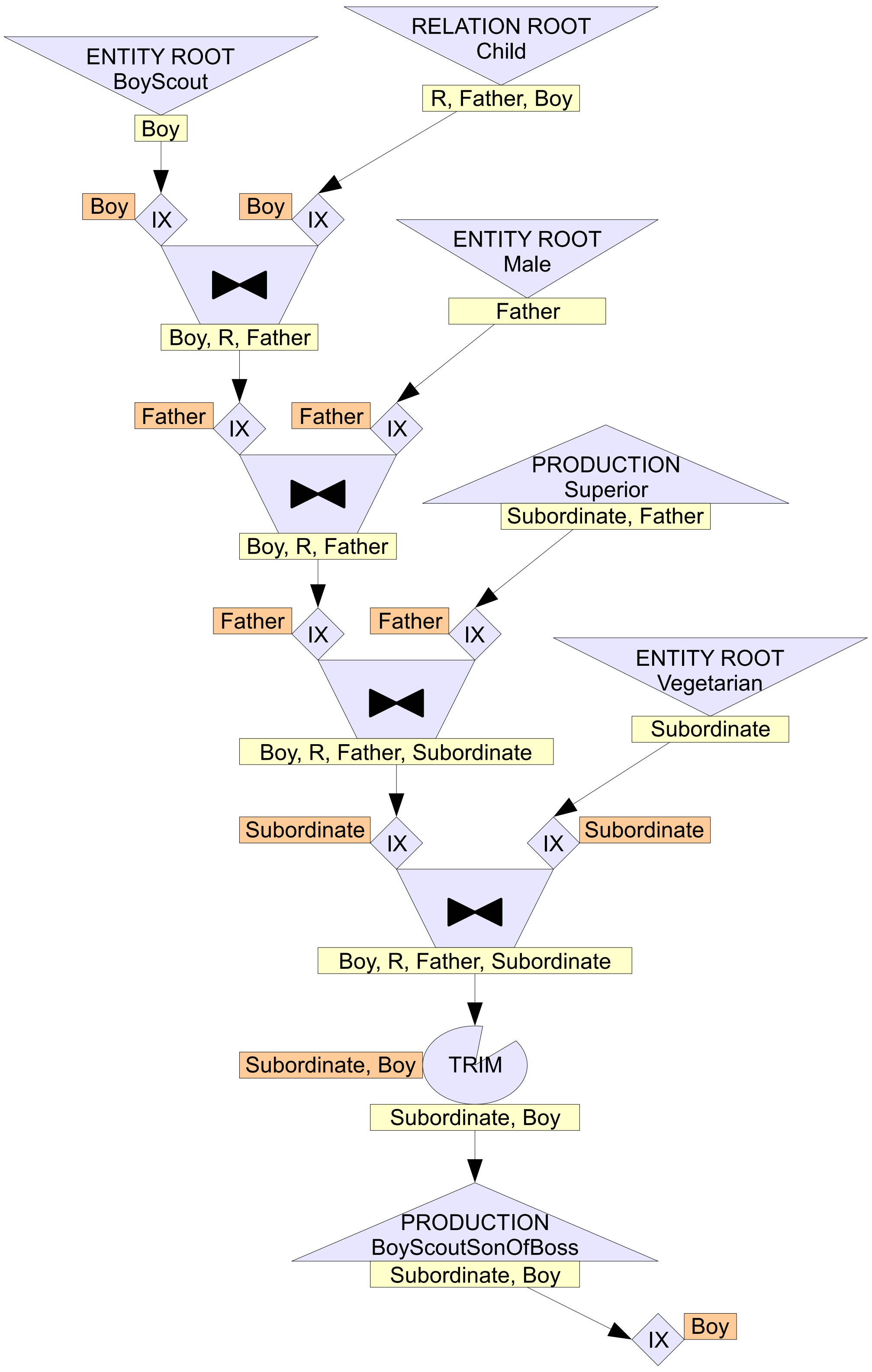


Figure 3.15: A RETE pattern matcher for the example pattern in Section 3.3.5; omitted details: the leading ConstantNode and InequalityNodes

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 calling patterns, as the production node is joined into them.

In contrast with the complexity of the general problem, two limited approaches were simple enough to be worthy to implement.

Indexer reusing

Indexers have their own tuple memory, consuming valuable space; in fact, apart from root nodes, Indexers are the primary source of memory consumption in the RETE network. Due to the nature of my adaptation of the beta node and its slots, many Indexer (see 3.2.3, page 36) 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. For these reasons, the RETE network implementation has the ability to reuse Indexers that are attached to the same node and operate according to the same pattern mask. This eliminates a major source of redundancy.

Greedy (full) node sharing

Beta and alpha nodes need not (and therefore do not) contain a pattern memory, so reusing them might not improve performance as much as reusing Indexers. However, reusability can have a ripple effect, as identical child nodes connecting to the same reused parent nodes can also be reused, leading to more and more reusable nodes in the ideal case, eventually helping to reuse memory-consuming Indexers. For this purpose, all RETE nodes[4] have been prepared to be reusable if the parent node(s) and other parameters match. This way, the pattern matcher now has a greedy approximation of the original node sharing idea.

3.4.2 Tuple inheritance

Basic concept

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 38) 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 one or both of it parent tuples without actually copying their contents. This is *tuple inheritance*.

[4]except, for technical reasons, TermEvaluatorNodes and production nodes

Left inheritance

The simpler version of this improvement is left inheritance, whose basic idea is borrowed from [43] (see Section **??**, page 30). 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 nor 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). The approach is particularly fitting for the naive construction algorithm outlined in Section 3.3, as it tends to build long lines of successive JoinNodes, each connecting to the previous on the primary side; large primary tuples are joined against small secondary tuples.

A more subtle advantage is easing hash calculation. Tuples cache their hash values for various hash table operations; as the used hash function is a simple rolling hash, the hash value from the primary side can be reused.

Two-way inheritance

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. Although a complete analogy would require a reference to a pattern mask describing the mapping from the secondary tuple to the rest of the positions of the result tuple, this new paradigm lets this projection operation be spared, as referencing the whole secondary tuple does not take up more space. This is another way to save CPU time besides memory usage with tuple inheritance.

Two-way inheritance could be beneficial if the secondary ancestor is, for example, the production node of a pattern with many symbolic parameters. In the future, experiments should be made to determine the conditions when a given JoinNode should use left or two-way inheritance.

3.4.3 Miscellaneous optimizations

The following potential improvements are currently at the planning stages; the author has not implemented any of them yet.

Type inference

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, provided that type safety of the model space is enforced by the transformation system. 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 and cheaper join operations; this latter argument favors the current approach.

Trivial indexers

If an indexer uses an empty pattern mask, its internal hash table would consist of a single bucket containing all the tuples. In other words, it simply recreates the contents of its ancestor node. If that ancestor node has built-in tuple memory, this replication is redundant; the indexer can be substituted with a dummy that does not actually store tuples. Another trivial case would be an identity pattern mask, returning all tuple elements in order.

Lazy evaluation

Borrowing an idea from LEAPS [4], the lazy evaluation of update messages can help to keep the size of the message queue low. The author could not find any related work improving RETE this way. It is possible that implementing LEAPS instead of RETE would be more profitable. Nevertheless, the ideas are described below.

Instead of one ancestor node sending one separate copy of the same message to each of its numerous child nodes, only one message item has to be placed into the message queue, containing an iterator of the set of child nodes. When the first message is delivered, the compound message does not get popped from the queue, only the iterator will advance to the next child node.

JoinNodes can produce messages lazily in one more way: if the received tuple matches several tuples from the other side, there is no need to send separate message items for each tuple combination, an iterator can be embedded into a single compound message. Combined with the previous trick, this compound message will have two nested iterator loops. Care should be taken to handle changes to the other side (and thus, the lazy tuple collection) while the unrolling of the lazy message is in progress; the timestamping idea of LEAPS seems appropriate for the task.

3.5 Implementation

The Incremental Pattern Matcher module was implemented in Java 6, as an Eclipse plugin, for the reason that VIATRA2 itself is a set of Eclipse plugins. The version described in this chapter consists of approximately 6500 lines of Java code; the parallelised version that will be introduced in Chapter 4 will extend the code to more than 7000 lines.

The pattern matcher successfully passes all unit tests verifying elementary pattern matcher capabilities, correct semantics of negative pattern calls, etc. Additionally, it passes all integration test suites of the Viatra Testing Framework, with the limitations described in Section 3.2.4, and Section 3.3.3 page 51; out of all integration test suites, 7 test suites specifically target pattern matching, with 17 individual test transformations in total.

The Incremental Pattern Matcher module has become a part of VIATRA2 Release 3.

3.6 Related work: state-of-the-art of pattern matching in graph transformation systems

Some parts of this text follow contribution of other authors (mainly Gergely Varró) to the overview in [7].

3.6.1 Non-incremental approaches

Most graph transformation systems employ the so-called *local search* strategy for pattern matching. When the occurrence set of a graph pattern is queried, the local search based pattern matcher starts with substituting one or ore variables of the pattern, and then gradually searches for other model elements related to the previous ones that can be substituted for other variables of the pattern. The order in which these search steps are executed is called the *search plan*. As choosing the search plan can have a great impact on performance, several strategies have been developed to optimize the speed of the search; for example, they can assign weights to different operations to assess the cost of each search plan candidate.

Many tools use local search based pattern matching, including: FUJABA[16], GrGen and GrGen.NET [5], Groove, VMTS and even the original pattern matcher of VIATRA2; see [40] for more examples and source citations. Still, they differ greatly on many aspects, including the exact method of generating the search plan; GrGen and can even redesign the search plan run-time to accommodate the characteristics of the model.

A notable exception is AGG [35], which treats graph pattern matching as a constraint satisfaction problem (CSP) and applies CSP algorithms and techniques (e.g. constraint propagation) in solving the problem.

3.6.2 Incremental and partially incremental approaches

All of the above systems need to start a search operation each time the occurrence set of a pattern is needed. This may degrade performance in certain scenarios; if information from previous searches can be salvaged, the pattern matcher may become faster. For this reason, some systems use a partially incremental approach. The PROGRES [36] graph transformation tool supports an incremental technique called attribute updates [18]. At compile-time, an evaluation order of pattern variables is fixed by a dependency graph. At run-time, a bit vector is maintained for each model node expressing whether it can be bound to the nodes of the left-hand side. When model nodes are deleted, some validity bits are set to false, which might invalidate partial matchings immediately. On the other hand, new partial matchings are only lazily computed.

The transformation engine of TefKat [21] builds and incrementally maintains an SLD resolution tree, see Section 2.4.5 for detailed discussion.

Apart from TefKat, no fully incremental approach could be found by the author, revealing the opportunity for a new incremental system to show that some problem classes are not tackled well by current transformation systems.

3.7 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.

Chapter 4

Exploiting parallelism in RETE-based graph pattern matching

4.1 Goal of this chapter

Modern desktop computers are often equipped with multi-core processors, and single-threaded execution does not take advantage of this increased computational capacity. The goal of this chapter is to examine ways in which parallelism could benefit RETE-based pattern matching.

Section 4.2 shows how the asynchronous approach allows the pattern matching process (or, more precisely, the update propagation process, as that is what consumes processing power in case of RETE) to be executed in the background, while the transformation continues uninterrupted. Section 4.3 generalises this approach to multiple RETE threads for systems with more than two CPU cores. Section 4.4 examines the possibility of building distributed RETE networks spanning over several computers. Finally, Section 4.5 briefly discusses how to deal with multiple simultaneous transformation threads.

4.2 Concurrent pattern matching

4.2.1 Concept

Using the asynchronous design described above, the load on the main thread of the transformation can be alleviated by executing the message queue consumption cycle on a separate thread. When the transformation changes the model, it is only required to post the new update message to the message queue, and can go on with its duties. The thread of the pattern matcher will execute the update propagation in the background, ideally without imposing a performance penalty on the transformation thread. When the message queue becomes empty, the RETE network has reached a *fixpoint*; the pattern matcher thread then goes to sleep and will not resume until a new update message is posted.

When the transformation needs to match a pattern, it has to make sure that update propagations in the background have settled and the matching tuples stored at the production nodes are up-to-date. If the network has not yet reached its fixpoint, the transformation thread will have to sleep until that happens. The Java methods wait() and notify() of java.lang.Object can easily solve this synchronisation problem.

4.2.2 Considerations

The TermEvaluatorNode (see 3.2.4, page 41) is a special case. It does operate normally in the thread of the pattern matcher, receiving and emitting update messages. Additionally, it may also be activated from the transformation thread, to re-evaluate a term that has just been influenced (e.g. by renaming an entity). The node will need to access its inner structures and emit updates in both cases, which leads to thread-safety issues. One way to resolve this is to define critical sections on the involved methods. This would have several disadvantages. First, it would make threads wait unnecessarily, which may lead to a loss in performance. Second, it would lead to messages being posted in a transformation thread, but by routines used for normal node-to-node communication, which is a bad design. While it would work with this version, it would break features introduced later; for example, in a multi-threaded matcher (see Section 4.3), this would lead to bad treatment of logical clocks. For these reasons, the author has chosen a different path: TermEvaluatorNodes now have listener nodes attached, that receive change notifications from the transformation thread in the same way as input nodes do. In a way, they can be considered input nodes. These notifications are delivered as change notification messages from the transformation to the RETE net. When the special listener nodes receive the notifications, already in the pattern matcher thread, they alarm the TermEvaluatorNode to act on the notification and re-evaluate the necessary tuples.

Synchronisation operations, critical sections, etc. have their own non-negligible costs. Message processing (inserting onto the queue, retrieving) is one of the performance-critical parts of the pattern matcher; synchronisations are performed to ensure safe access of the message queue for both the transformation and the pattern matcher thread, and they impose a significant overhead on performance. To combat this effect, the message queue was split into two queues: an *incoming queue* appended at the end of an *internal queue*; this technique will be referred to as *dual queueing*. The network uses the internal queue just like the original message queue concept described in Section 3.2.5; this is where all nodes output their propagated update messages. If the internal queue becomes empty, the messages will be fetched from the incoming queue; this is where the transformation sends its change notifications. If there are no messages there either, the pattern matcher thread waits for messages arriving in the incoming queue; there is no need to wait for the internal queue, as messages could only be put there by the pattern matcher thread itself. Only the incoming queue is accessed by both threads, therefore the internal queue need not be protected by critical sections. As the internal updates can be a majority of all update messages, this optimization may save a significant amount of the synchronisation overhead; it saves up to 50% of the total time in the benchmark described in Section 5.2.4. The technique of dual queueing may violate the strict global ordering of update messages, as change notifications may take long to deliver. Still, it preserves the ordering on any single RETE edge (and by extension, the communication channels between the transformation and the network), and thus will not lead to inconsistent results.

4.2.3 Performance expectations

While the original local search based pattern matcher operated with cheap model changes and costly pattern queries, the first version of the RETE-based matcher introduced a new situation with a moderate overhead on model change balanced by instant pattern queries. The newly introduced concurrent RETE approach combines the advantages of the former two: it has cheap model change costs, and potentially instant pattern queries. Although the transformation might have to wait if the background pattern matcher thread is not ready yet, the worst case of this time loss is still comparable with the update overhead of the original RETE approach.

This concurrent approach can improve performance over non-concurrent implementation (as described in Chapter 3) if there are comparatively infrequent pattern matcher queries and continuous model changing inbetween. This would correspond to the forall construct of the VTCL language (get all matches of a pattern and process them one by one). This complements the advantage of incremental pattern matching over non-incremental techniques, manifesting especially on the iterate-choose construct (as long as possible, get one occurrence of a pattern and act on it).

4.3 Multi-threaded pattern matching

4.3.1 Dividing the RETE net

The concurrent patten matching approach can be improved further given that the hardware architecture is capable of running multiple threads efficiently. The basic idea is to operate multiple message consumption threads. However, if these threads used the same message queue and RETE nodes, and multiple threads could access the same node simultaneously, it would possibly lead to inconsistency problems that could not be easily averted by locks.

My proposed solution divides the network into separate RETE *container*s with their own distinct set of nodes, and equips each RETE container with a dedicated pattern matcher thread consuming a dedicated message queue. Each container is responsible for forwarding messages to its nodes using the dedicated message queue. This way, no two threads operate on any single node, thus mutual exclusions are not necessary. Relaying messages between two containers is accomplished by enqueueing the message in the target container. To be more precise, containers are actually dual queueing (see 4.2.2, page 61); change notifications from the transformation and update messages from other containers are placed into the incoming queue, while in-container messages use the internal queue.

If a container runs out of messages, it reaches a *local fixpoint*, otherwise it is *active*. The *global fixpoint* occurs when all containers are in a local fixpoint. In order to retrieve up-to-date and consistent matching sets, the transformation thread has to wait for a global fixpoint. This thread synchronisation goal is, however, not that easy to accomplish, since a container can leave its local fixpoint and become active again before a global fixpoint is reached due to incoming messages from other, still active containers. To address this issue, I have designed a termination algorithm that is able to determine global fixpoints.

4.3.2 Proposed termination protocol

Each container C_i has a logical *clock* (denoted $clock_i$) that is incremented whenever a local fixpoint is reached by the message consumption thread of the container (denoted $thread_i$). Each time container C_i sends an update message to container C_k, the message is appended to the message queue of C_k and the value of $clock_k$ is retrieved and stored in c_i as $criterion_i[k]$, all as one single atomic step[1]. The retrieved clock value indicates the *termination criterion* that the network cannot reach a global fixpoint until the $clock_k$ exceeds that value, meaning that the relayed message has been delivered to the node in C_k and all of the (local) consequences have been resolved, resulting in a new local fixpoint.

When C_i reaches its local fixpoint, it atomically increments its clock and reports the event to a *global RETE network object*; this report includes the incremented $clock_i$, along with the values $criterion_i[k]$ for each k [2]. Similarly, when the transformation changes the model and consequently sends a change notification (formulated as an update message) to an input node in container C_k, it hands over the message to the message queue of C_k, fetches the $clock_k$ from that container and reports it to the network object as a termination criterion, also performed as a single atomic step.

The global network object maintains an array $criterion_{global}[k]$ storing the *largest* reported criterion for each k, and $clock_{reported}[i]$ for the latest clock value reported by C_i. Upon receiving the report, the global network object evaluates whether a global fixpoint is reached and wakes the transformation thread when appropriate. Determining whether a global fixpoint holds is as simple as checking, for each container, whether the highest reported termination criterion value stemming from that container is exceeded by its the latest reported fixpoint-time clock value. This will be referred as the *Termination Condition*, and formulated as:

$$\forall_k : clock_{reported}[k] > criterion_{global}[k] \tag{4.1}$$

This method can be simplified if these checks are made upon receiving a report by or about C_k, and $criterion_{global}[k]$ values are simply deleted if they satisfy the above condition; global fixpoint holds if and only if $criterion_{global}$ is empty, i.e. there are no more termination criteria.

Note that for each container C_k, $clock_k = clock_{reported}[k]$ always holds, since they are both initialised with the same value (not mentioned above), and whenever $clock_k$ is updated, the new value is copied

[1] the outlined procedure is only necessary if $k \neq i$; messages sent and received within the same container can use the message queue the same way as described in Section 3.2.5

[2] actually, only the termination criteria retrieved since the last local fixpoint are truly needed; since the previously retrieved clock values have already been reported, there is no need for them anymore, and the implementation deletes them after having sent the report

to $clock_{reported}[k]$, which cannot change in any other way. Another observation is that relaxing the relational operator in the Termination Condition (4.1), it always holds that $clock_k = clock_{reported}[k] >= criterion_{global}[k]$, as any stored criterion is the value of the clock at some point in the past.

Atomicity can be guaranteed by taking some synchronisation measures, as described below.

- When $thread_i$ takes a message from its message queue, it enters a critical section of the message queue for the brief duration of the retrieval, but not during waiting for a new message if the queue is empty.
- When container C_i sends a message to C_k, $thread_i$ enters a critical section of the message queue C_k, enqueues the new message, copies the value of $clock_k$ into $criterion_i[k]$, and leaves the critical section.
- When C_i reaches a local fixpoint,$thread_i$ enters a critical section of the message queue of C_i, increments $clock_i$, enters a critical section of the global network object that updates $clock_{reported}[i]$ and possibly $criterion_{global}[k]$ for some value of k, and finally leaves both critical sections.
- When a transformation thread sends a message to C_k, its thread enters the critical section of the message queue of C_k, enqueues the new message, enters a critical section of the global network object, copies the value of $clock_k$, sends the report that possibly updates $criterion_{global}[k]$ and leaves both critical sections.

I will formally prove that the outlined termination algorithm is correct and deadlock-free. Correctness refers to the design goal that the Termination Condition (4.1) holds exactly when the system is in a global fixpoint. The deadlock-free property applies for the message consumption threads, as well as to the transformation thread that can possibly wait for a global fixpoint to match a pattern. For the sake of simplicity, the proof assumes that the optimization where $criterion_i[k]$ and $criterion_{global}[k]$ values get deleted is not employed.

4.3.3 Proof of correctness and liveness

Termination Condition $\Longrightarrow$ all containers are in a local fixpoint. Proof by contradiction: let's assume that some containers are not in a fixpoint. From those containers, select the one that has been active (not in a fixpoint) for the longest time (this time interval is finite, as all containers initialise in a fixpoint); name that container C_k. Termination Condition (4.1) implies that $clock_k = clock_{reported}[k] > criterion_{global}[k]$, meaning that a termination criterion of value $clock_k$ was never reported for C_k, since $criterion_{global}[k]$ can only increase.

Whichever message marked the end of the last fixpoint of C_k, its sender received the value $clock_k$ as a termination criterion; this criterion was never reported. Had the sender been a transformation, the criterion would have been atomically reported to the network according to the protocol; this means that the sender must have been a container C_i. Since C_k is asserted to have been active longer than

C_i, C_i must have reached a local fixpoint since sending that message, and then it should have reported $criterion_i[k] = clock_k$ to the global network object. This is a contradiction. □

Termination Condition $\Longleftarrow$ *all containers are in a local fixpoint.* Proof by contradiction: let's assume that some k violates the Termination Condition (4.1), i.e. $clock_k = clock_{reported}[k] = criterion_{global}[k]$. The criterion was reported either by a transformation or by another container. In the former case, the transformation was sending a message to C_k when delivering this report; in the latter case, a container C_i delivered a message to C_k before reporting. In either case, at some point in the past, C_k received a message when $clock_k$ already had its present value. Even if it had been in a local fixpoint, receiving the message made C_k active. Since according to the assumption, C_k is in a fixpoint at the present time, it must have reached a new fixpoint since the message was received. This means that $clock_k$ must have been incremented since the message was received, but $clock_k = clock_{reported}[k]$ is still the same, which is a contradiction. □

The protocol cannot halt in a deadlock. The critical section of the *global network object* is only entered for short reporting routines, during which there is no waiting, so the critical section will eventually be free again. This also means that waiting to enter this critical section will not cause a deadlock. On the contrary, a thread can suspend execution and wait within critical sections of a *message queue*, but the only way this can happen is while trying to enter the inner critical section. As the inner critical section belongs to the global network object and is already shown to be live, critical sections of a message queue can also be freed.

The only remaining way a message consumption thread can be forced to wait is when it is in a local fixpoint and waiting for a new message. In this case, if there are active containers remaining, they can operate further. Finally, if all message consumption threads are waiting, then all containers are in a local fixpoint, so global fixpoint is reached, and the transformation thread is not forced to wait. □

4.4 Distributed pattern matching

A way to scale the RETE approach further would be to build distributed RETE networks. The mechanisms lain out in Section 4.3 provide easy transition to a distributed environment: as RETE containers already use a message-based communication interface, they can be placed on any participant machine independently of each other. The termination algorithm outlined above is still applicable here, with the same proof of correctness. It should be noted, however, that some RETE nodes (namely: input nodes, production nodes and TermEvaluationNodes) are tied to the model transformation system, and it is not practical to accommodate them at remote machines; for their sake, a special *head container* must always be reserved at the computer hosting the model transformation system.

Nodes are addressable from throughout the cluster with their container ID and their local ID within the container. These serialisable addresses serve as the recipient field for inter-container messages and net construction orders. This eliminates the need that RETE nodes be aware of the remote communication; only the global network object and the containers have to provide a remote interface.

Apart from the increased processing power, the easily scalable memory pool also benefits the RETE-based matcher. Network latency, however, may put a considerable overhead on message passing between containers, seriously undermining the goal of speeding up the transformation process. For this reason, the implementation and evaluation of the distributed pattern matcher is still subject to future work. Distributing the pattern matcher along with the model space and the interpreter of VIATRA2 could still allow much larger models to be processed, and therefore remains a major future goal.

The author could not find any prior art of distributed RETE implementations in the literature, apart from a patent application[22] that focuses on expert systems and provides no clear description of certain aspects of the solution. In particular, the patent application does not present or claim any termination protocol similar to the one presented in Section 4.3.2; even if possibly not required by an expert system, detecting the global fixpoint is necessary in a model transformation context.

4.5 Thread-safe pattern matcher for multi-threaded transformations

The usefulness of optimizing the pattern matcher has its limits, as a large part of the CPU time is consumed by the transformation execution itself. Further performance gains can only be achieved by speeding up the transformation execution; the power of multi-core architectures can be harnessed by multi-threading the transformation execution component as well. While this topic is out of scope for this thesis, it is interesting to discuss how the RETE-based pattern matcher can handle multiple threads using it.

The basic operation of the RETE network is easily adopted for a multi-threaded environment. When transformation threads inflict changes on the model, they send update notifications atomically; this involves inserting an update message addressed to the appropriate input node into the message queue of the node's container, retrieving the current clock value of container, and storing it at the global network object as a termination criterion. When transformation threads need the tuples matching a pattern, the pattern matcher call returns them immediately if the network is in a global fixpoint, or suspend the thread until that state is reached; again, the intrinsic synchronisation features of Java allow the thread-safe execution of this scheme. As the proof in Section 4.3.3 is easily extended to the case of multiple transformation threads, the proposed termination protocol remains valid.

The construction phase deserves special mention. If a thread needs to match a pattern for which no matcher has been built yet, it must extend the existing network structure. While new nodes are being built and contents are pulled into them, propagating updates would cause serious issues, and multiple threads building the RETE network concurrently would cause further problems. Therefore, an assymmetric (R/W) lock has been added to the RETE network; while basic operations outlined above require only a compatible lock, threads attempting network construction must grab the exclusive lock (and also wait for global fixpoint) before commencing work.

Chapter 5

Performance evaluation

5.1 Goal of this chapter

The goal of this chapter is to measure and evaluate the quantitative benefits of the incremental pattern matching approach. Section 5.2 introduces the benchmark problems on which the measurements will be conducted along with the measurement conditions. Section 5.3 compares the Incremental Pattern Matcher against the original, local search based pattern matcher of the VIATRA2 framework, and against GrGen.NET, a high-performance graph transformation system, in order to determine whether incremental pattern matching can prove advantageous. Section 5.4 compares different versions of the Incremental Pattern Matcher to evaluate various optimizations and improvements.

5.2 Benchmark environment and test cases

5.2.1 Measurement environment

The measurements have been carried out on a standard desktop computer with a 2 GHz Intel Core2 processor, 2 gigabytes of system RAM available, the Windows Vista operating system, running version 1.6.0_05 of the 32-bit Sun Java SE Runtime (for VIATRA2) and version 3.0 of the .NET Framework on (for GrGEN.NET). In general, several test runs were executed, ranging from five to ten runs on different test cases. The transformation sequences were coded so that little or no output was generated; in the case of VIATRA2, the GUI was not disabled. Execution times were measured with millisecond precision as allowed by the operating system calls.

5.2.2 Mutual exclusion

The Varró benchmarks [42] define a set of transformations on which many transformation systems have been measured. The author has conducted measurements with the distributed mutual exclusion benchmark, in particular the long transformation sequence (*LTS*) and short transformation sequence (*STS*) versions; for the sake of brevity, only the results of the STS case will be discussed later in this paper.

The test case describes a process ring arbitrating control of resources. After growing the process ring to the required size, STS creates one resource that is owned by one process; in the next phase, every process issues a request on that resource; the resource will be passed along the ring until a full cycle is performed. LTS starts with a complete ring and a separate resource owned by every process; the whole set of resources is shifted along the process ring once in each cycle, involving waiting and blocked processes, as well as a deadlock-resolving algorithm. The ring size is the characteristic parameter in both cases; for LTS, the number of executed cycles can also be specified. For a detailed test specification, see [42].

5.2.3 Simulation Scenario based on Petri net firing

Petri-net firing is introduced in [6] as a model simulation task characterised by nontrivial graph patterns, small changes to large models in each step, and ALAP-style (as long as possible) execution of steps. The article defines a methodology to generate arbitrarily large bounded and live Petri-nets, with either *sparse* (i.e. has few tokens and few enabled transitions at any time) or dense token placement. A set of generated Petri-nets is also published, that form the basis of benchmarking model transformation systems. The key parameters of a test case are the identity of the actual Petri-net and the number of firings to be performed.

The article was co-authored by me, and following test case description is a copy of its relevant parts.

Description. The Petri net benchmark was chosen for the scenario of simulation of visual languages with dynamic operational semantics. This scenario summarizes typical domain specific language simulation with the following characteristics: (i) mostly static graph structure, (ii) relatively small and local model manipulations, and (iii) typical *as-long-as-possible* (ALAP) execution mode. This benchmark focuses on the effective reusability of already matched elements as typical firing of a transition only involves a small part of the net. While an incremental pattern matcher can track the changes of the Petri net and updates only the involved sub-matchings, non-incremental local search based approaches will have to restart the matching from scratch after the net changed.

Test case generation. In the Petri net test set, some “regular” Petri nets have been selected as *test cases*, which are generated automatically. Here regular means that the number of *places* and *transitions* are approximately equal (their exact ratio is around 1.1). Furthermore, the so-called sparse net has only a low number of tokens, and thus, there are few fireable transitions in each marking.

The elements of the test set have been generated by using six reduction operations (in the inverse direction to increase the size of the net) which are described in [24] as means to preserve safety and liveness properties of the net. These operations are combined with a weighted random operation selection. This allows fine parametrization of the number of transitions and places with an average fan-out of 3-5 incoming and outgoing edges. In all test cases, the generation started a simple, trivially live Petri net and the final test graphs are available in PNML [20] format at [15]. As the size of a Petri net cannot be described by a single parameter, the term "Petri net size" refers to the number of property preserving operations performed during the generation of the net.

Execution phases. A step in the iterative execution sequence contains two phases: (i) a fireable transition is non-deterministically selected by pattern *isTransitionFireable* (Fig. 5.1) and then (ii) the GT rules *addToken* and *removeToken* are applied to simulate the token flow (Fig. 5.2).

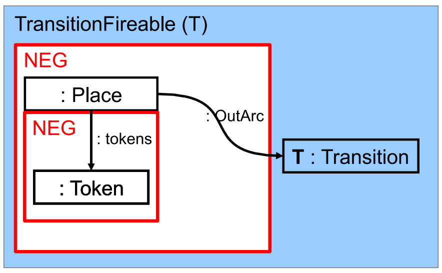
Despite its simple execution semantics, it is easy to derive additional Petri nets as new benchmark scenarios with significantly different run-time characteristics for the different graph transformation tools. For example, a Petri net with an equal number of transitions, places and tokens but with few fireable transitions can be used as a benchmark where type-based optimization strategies of pattern matcher algorithms are neutralized, which forces the pattern matchers to use other heuristics.

Note that the only assumption that was made on the Petri net test cases is to use *live* and *bounded* nets to have a potentially unbounded execution sequence. *Short* execution sequences consist of 1000 consecutive transition firings, while *Long* execution sequences consist of 1000000 transition firings.

For this benchmark, the total execution time of the simulation sequences were compared. As the actually firing transitions are non-deterministically selected by the tools, the pattern matchers were allowed to select their own execution paths, but this turned out to have only insignificant effects on execution times.

5.2.4 Combinatorical explosion benchmark

A third benchmark is defined here, with the purpose of comparing different current and future variations of the Incremental Pattern Matcher module. One of the design goals was to minimize the impact of transformation interpretation, model management and other aspects of the framework, and focus on



```
pattern isTransitionFireable(Transition) ={
  transition(Transition);
  neg pattern notFireable_fl(Transition) =
  {
   place(Place);
   outArc(OutArc, Place, Transition);
   neg pattern placeToken(Place) =
   {
     token(Token);
     tokens(X, Place, Token);
   }
  }
}
```

Figure 5.1: Petri-net firing condition

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Place, in Transition) = {
 precondition find sourcePlaceWithToken
    (Transition, Place, Token);
 postcondition find sourcePlaceWithoutToken
    (Transition, Place, Token);
}
// Adds a token from the place 'Place'.
gtrule addToken(in Place, in Transition) = {
 precondition find targetPlaceWithoutToken
    (Transition, Place, Token);
 postcondition find targetPlaceWithToken
    (Transition, Place, Token);
}
```

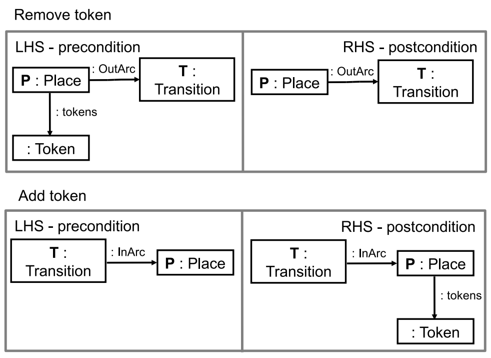


Figure 5.2: Graph transformation rules for firing a transition

the raw performance of the pattern matcher. Another design goal is making it possible to observe the behaviour of certain optimization techniques. The result is a synthetic benchmark described below.

The benchmark uses graph models with a single node type and no edges. The model is initially empty and the pattern matcher is allowed to prepare the RETE net in advance. The transformation first creates N nodes, reaching the so-called *commit point*, and then queries the occurrence sets of two patterns. Both patterns have the same definition: k separate[1] nodes of the same type (without any edges required). It is easy to see that each one of these patterns has $N!/(N-k)!$ occurrences after the transformation run.

In particular, with $N = 40$ and $k = 4$, the transformation performs 40 steps to create 40 entities, while the pattern matcher has to generate a result set of size $40!/36! = 2193360$; this emphasises the performance of the pattern matcher component. Having two identical patterns, this test case is a good opportunity to study the effects of node sharing.

5.3 The Incremental Pattern Matcher compared to other approaches

This section uses the mutual exclusion and Petri-net firing benchmarks to measure the performance of the Incremental Pattern Matcher module (*VIATRA/RETE*) against the original, local search based pattern matcher module (*VIATRA/LS*). A different graph transformation system, GrGen.NET has also been included in the tests, to provide reference.

GrGen.NET features a search-plan driven, host-graph sensitive pattern matcher[5] and various other features including compiled transformations. As a result, the highly optimized GrGen.NET is one of the fastest transformation engines as of today, as shown at [1]. This makes GrGen.NET the ideal reference for performance comparison.

[1]this is implicitly enforced by injectivity

The author has carried out the measurements in this section for an ICGT 2008 paper submission [6], written with Ákos Horváth and István Ráth and Dániel Varró. The wording of this section is partly taken from [6] as well, and as such, it should be considered joint work.

5.3.1 Distributed Mutual Exclusion Algorithm (STS)

In order to compare the pattern matcher algorithms using an already available benchmark, the performance of the VIATRA2 local search based (VIATRA/LS) and incremental (VIATRA/RETE) pattern matchers have been evaluated along with the GrGEN.NET with the short transformation sequence version of the *distributed mutual exclusion algorithm* test set (see Section 5.2.2) which is not a primary application filed for incremental pattern matching.

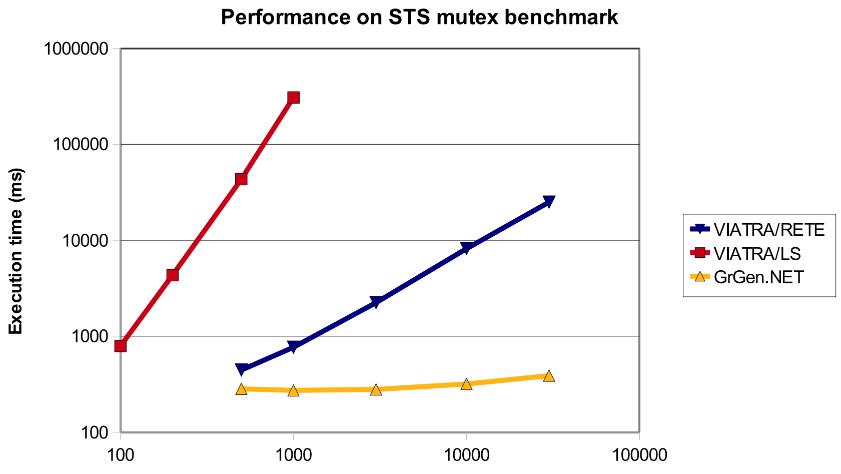


Figure 5.3: Results for the STS mutex benchmark

The results are shown in Fig. 5.3 with logarithmically scaled axes, where the *size of the process ring* represents the number of processes. The following observations can be made: (i) the scaling complexity is a high order polynomial for VIATRA/LS and close to linear for VIATRA/RETE and linear for GrGEN.NET[2]; (ii) this test set seems to be a better fit for optimized local search based approaches, or at least they have a smaller disadvantage, as incremental caching of non-reusable model elements produced in the second phase of STS increases the overhead of the cache synchronization. Additionally, by looking at memory consumption figures, it can be seen that the static graph structure imposes a linear memory overhead on RETE, which is the same complexity class as VIATRA/LS and GrGEN.NET; memory consumption will not significantly limit the model size in these kinds of problems.

[2]it would even seem sublinear, as constant warmup costs weigh in significantly for smaller model sizes

5.3.2 Simulation of Petri-nets

The Petri net synchronization benchmark was executed with *short* (1000) and *long* (1000000) execution sequences, on large, sparse Petri nets.

The size parameters of the nets used as test cases are depicted in Fig. 5.4. *Net size* represents the number of randomly applied net-growing operations used during their generation, while *Places*, *Transitions* and *Tokens* represent their actual number. The results are shown in Fig. 5.5 with logarithmically scaled axes, where model size indicates the *net size* of the test case.

Net Size	Places	Transitions	Tokens
10000	7497	7450	10
20000	14987	14970	11
50000	37581	37593	12
75000	56331	56053	13
100000	74924	75124	14

Figure 5.4: Size of test cases

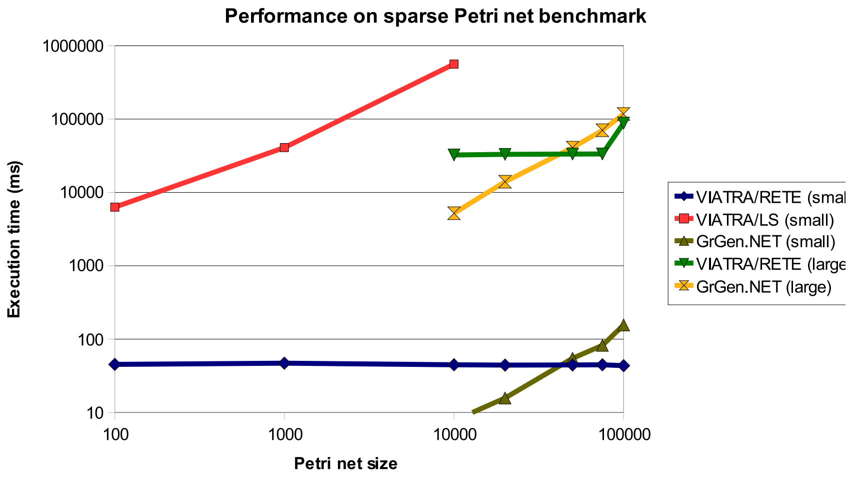


Figure 5.5: Results for the Petri net firing benchmark

As it can be seen from the graph, VIATRA/RETE has a predictable linear scaling up to model size of 10^5 with a speed of at least two orders of magnitude faster than VIATRA/LS. As expected, the incremental approach works well for large model sizes as long as there is enough memory (the spike in case of long transformation sequences occurred because of frequent garbage collection as the heap was nearly filled).

VIATRA/RETE matches and outperforms the GrGEN.NET tool for very large models in case of both short and long execution sequences. Moreover, with additional memory provided, the characteristics of VIATRA2 are expected to improve for even larger models with predictable execution time.

This result is a significant achievement considering the architectural and run-time differences between VIATRA2 and GrGEN.NET. Most notably, GrGEN.NET uses compile-time optimizations and an entirely different model persistence approach based on compile-time generated type information, whereas VIATRA2 uses a generic model storage supporting dynamic typing and support for interactive applications such as a notification and transaction management mechanism (note that the VIATRA2 GUI was not disabled for the measurement, while GrGEN.NET was used without GUI through GrShell). However, for fairness, it should be pointed out that (unlike the mutual exclusion case) this benchmark was prepared by ourselves (i.e. by GrGEN non-experts), thus additional language or tool-specific optimizations might be available.

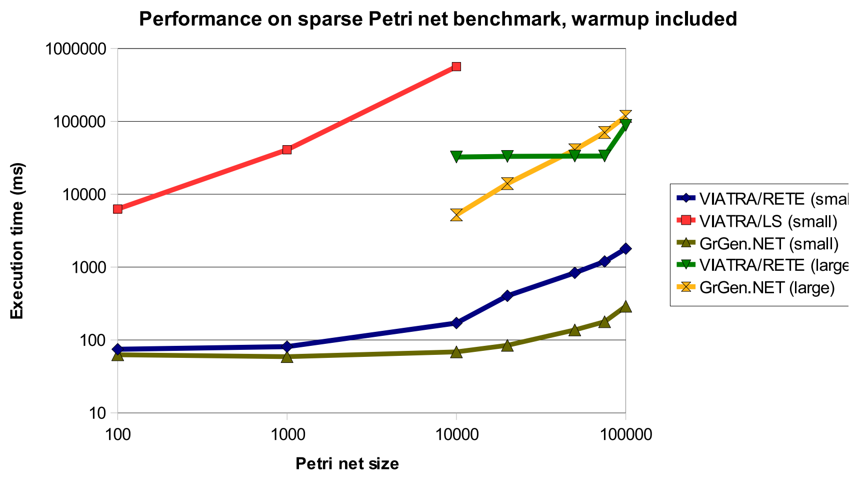


Figure 5.6: Results for the Petri net firing benchmark (warmup included)

Still, constructing the RETE network and filling it with the initial contents of the model is, naturally, not a constant time operation. For small firing sequences, the network construction phase can dominate time consumption, and make RETE less efficient. If the simulation length is long enough compared to the size of the model, building a RETE net pays off. Figure 5.6 shows the results with the time of the warmup run included in the average. This new comparison, however, is misleading and less accurate than Figure 5.5, as the time required to read the input file containing the model is still not taken into consideration in either systems; this comparison is therefore disadvantageous to VIATRA/RETE, as the RETE network is only built on demand.

The good performance of RETE comes at a cost: increased memory consumption. Figure 5.7 compares the memory usage of the two Viatra configurations[3]. They both use the same model storage facility, user interface, interpreter, etc. the only difference is the pattern matcher module. While the local search based pattern matcher takes up little space, the indexing and caching mechanism of the Incremental Pattern Matcher occupies almost as much memory as the model space itself. Still, doubling the amount of available memory is an accessible option, while the execution times of the local search based matcher become unacceptable already at relatively small model sizes.

[3]The length of the executed firing sequence was as small as 10, to allow even the local search based implementation to finish in a reasonable time on the larger problem instances.

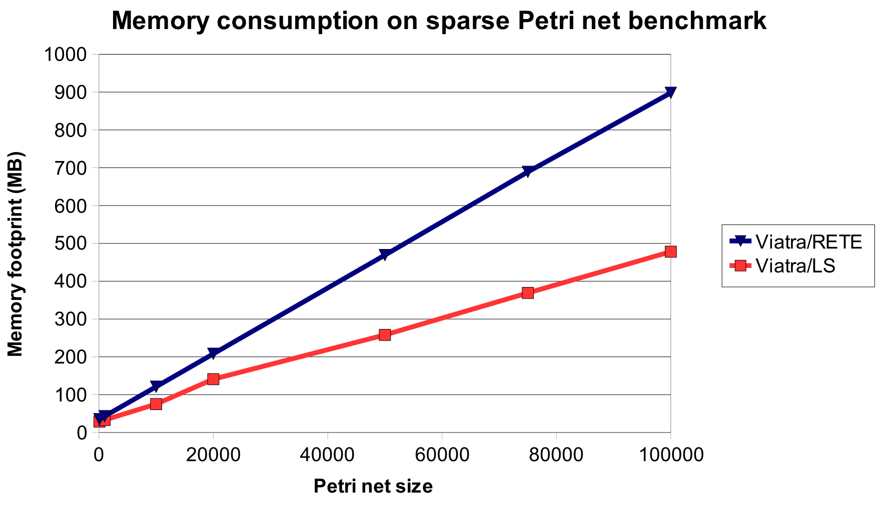


Figure 5.7: Memory consumption for the Petri net firing benchmark

5.3.3 Summary

Analyzing the obtained results, the following conclusions can be drawn:

(i) A major concern of any incremental pattern matching implementation is the increased memory consumption. While the presented implementation does indeed consume more memory than the standard local search-based VIATRA2 engine, this overhead, even for the extreme model sizes in the benchmark problems, is still within the bounds of RAM available in modern desktop computers making the approach feasible for a wide range of applications.

(ii) Within the memory boundaries, the RETE-based pattern matcher provides a *predictable, linear scaling* up to the 10^5 model size range in both scenarios. While even generic transformations experience a speed-up, the real potential of the implementation is revealed in the scenarios especially suited for incremental pattern matching where the execution speed matches, or even surpasses the speed of the fastest conventional graph transformation tool employing compile-time optimization.

(iii) By comparing the run-time characteristics of the given test cases, it seems evident that the best results could be achieved by employing different pattern matching strategies for different execution phases, or, even for different patterns in a model transformation program.

5.4 Measuring improvements and optimizations of the RETE network

These measurements were performed on the synthetic benchmark described in Section 5.2.4, with $N = 40$ model elements and $k = 4$ pattern variables. Besides recording the total execution time, the time in which the commit point was reached has also been measured, to be able to analyze the effects of concurrent matching. Memory consumption, or more precisely, the total heap usage of the VIATRA2 system, has also been recorded, allowing the study of memory optimization techniques. The benchmark was run with different node sharing (see 3.4.1, page 53), tuple inheritance (see 3.4.2, page 55) options; most of them where executed in a concurrent fashion (see Section 4.2) with a single container, but a single measurement series with the original non-concurrent module was also performed. The results are shown on Table 5.1. Warmup runs are not contained in the average, as the Java VM needs time to compile and optimize Viatra, and this has a noticable impact on measured times.

concurrent	node sharing	inheritance	mem(MB)	commit (ms)	runtime(ms)
no	full	left	397	10770	10770 (± 1.94%)
special	full	left	395	11229	11230 (± 1.25%)
yes	full	none	398	9	12595 (± 2.69%)
yes	full	left	396	8	14117 (± 2.76%)
yes	indexer	left	500	8	18684 (± 5.71%)
yes	none	left	501	10	18395 (± 1.61%)

Table 5.1: Memory footprint in megabytes; average commit times, execution times (and standard deviation of the latter) in milliseconds; for different versions of the Incremental Pattern Matcher module on the combinatorical explosion benchmark

As the two patterns have $40!/36! = 2193360$ occurrences each, the RETE net with its most memory-preserving settings consumes about 90 bytes per each pattern occurrence in this example. Note that in the general case the production node might not be the one with the most tuples contained; it is possible that later constraints narrow down the occurrence sets of subpatterns. Still, this measurement gives the impression that the memory footprint of RETE is in reasonable proportion to the combinatorical difficulty of the pattern.

Full (greedy) node sharing is shown to have a substantial impact both on memory and time consumption, as the two patterns can share their whole recognition network (apart from the respective production nodes). This result is, however, somewhat deceptive, as this synthetic benchmark was particularly tuned to be ideal for greedy sharing, to prove its worth. On the other hand, this test case somewhat downplays the benefits of simple indexer sharing, as input sets are exceptionally small.

Note that if even indexer sharing is turned off, the indexers used to extract the matches from the production nodes can be released, and the memory footprint eventually descends to 365M from the peak (501M). The downside is that for the next query, this one-shot indexer will have to be rebuilt, consuming an additional 3947 ms (± 11.07%) on average (if the model is unchanged). If at least indexer sharing is enabled, this cost virtually disappears (0ms and 1 ms values were observed).

It is surprising to see that tuple inheritance fails to make a significant difference regarding memory consumption, while it has a noticeable performance penalty. It is likely that tuple inheritance only pays off with patterns containing more variables. Future investigations are needed to determine the conditions where this optimization has substantial impact.

With the concurrent execution of the pattern matcher and the transformation, the transformation can finish changing the model and reach a commit point earlier - with this special benchmark, reaching the commit point (8-10 ms) was so fast it was hardly measurable. However, as the figures indicate, the update propagation itself lasts longer in the concurrent case. This phenomenon can be traced back to at least three causes.

- First, the concurrent version has to deal with synchronisation, mutual exclusion, waiting and other thread operations that have their own considerable overhead. Since message delivery relies on them, which is probably one of the most performance-critical elements in update propagation, it is to be expected that this has a performance impact. The implementation uses dual queueing (see 4.2.2, page 61) to reduce the magnitude of this effect; execution time can be reduced by as much as 50% on this benchmark test (not shown on Table 5.1).

- Second, the code maturity of the new, parallelism-enabled branch of the pattern matcher is not yet on the same level as the well-tested original RETE implementation. While the most recent versions of the parallel branch have made significant progress, this is still expected to change for the better in the near future.

- Third, this particular test case seems to favor non-parallel execution. In particular, it appears that somewhat less computation is performed if change notification messages are only delivered after all internal update messages are settled. To prove this, a special test has been performed, marked 'special' on Table 5.1. In this test, the concurrent matcher is set up to wait for a fixpoint before accepting change notifications, thereby mimicking the behaviour of the non-concurrent matcher. The results indicate that this special pattern matcher achieves better performance than the concurrent one in this special case (close to the speed of the non-parallel one), even though the actual implementation is virtually the same. It is an important lesson learned from these tests that this factor also has to be taken into consideration.

Chapter 6

Summary and conclusions

6.1 Overview

In this thesis, I proposed an effective **incremental pattern matcher** to be used in conjunction with model transformation technology, extended the VIATRA2 model transformation framework with this functionality, and measured its performance. I have also explored the possibilities of improving this concept with parallel execution.

6.2 Scientific contributions

- I have 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**.
- I have investigated **concurrent pattern matching** performed on a separate thread, to gain benefits from multi-core hardware architectures.
- I have proposed a technique of **multi-threaded pattern matching** happening on separate threads, to improve these benefits further.

6.3 Practical accomplishments

- I have **implemented** the proposed Incremental Pattern Matcher component in the VIATRA2 model transformation framework, to provide support for truly incremental change propagation. Over **7000** lines of Java code were written.
- From a proposed list of **optimizations**, I have implemented greedy node sharing (indexer reusing included) and left tuple inheritance.

- To boost performance on multi-core processors, I have implemented the proposed version of the pattern matcher that operates **concurrently** to the transformation, and may run on **multiple threads** itself.
- I **conducted measurements** and concluded that the incremental pattern matching engine was **efficient**, and in some circumstances, it even had higher performance than one of the fastest known model transformation systems.

6.4 Future work

There is plenty of room for improvement in the incremental pattern matcher. I 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.

The parallel version of RETE is not yet on the same level of maturity and effectiveness as the original one, and needs some care in the near future. Furthermore, the construction of RETE is not performed in a parallel fashion yet.

It is also desirable to extend the benchmark suite to further explore the problem space where an incremental engine is more efficient than an ordinary pattern matcher.

The Viatra team has long-running plans to make VIATRA2 distributed; this involves the pattern matcher, and also other components of the framework. The Incremental Pattern Matcher module is close to being ready for this leap, as the multi-threaded version was designed and implemented with this future goal in mind.

Appendix A

Benchmark source codes

A.1 STS Mutex benchmark

A.1.1 VTML metamodel for Viatra

```
entity(mutex)
{
  entity(metamodel) {
    entity(resource);
    entity(process);
    relation(next, process, process);
    relation(blocked, resource, process);
    relation(held_by, resource, process);
    relation(token, resource, process);
    relation(release, resource, process);
    relation(request, process, resource);
  }

}
```

A.1.2 VTML initial model for Viatra

```
namespace mutex;

  entity(model) {
    mutex.metamodel.process(p1);
    mutex.metamodel.process(p2);
    mutex.metamodel.process.next(n1, p1, p2);
    mutex.metamodel.process.next(n2, p2, p1);
  }
```

A.1.3 VTCL implementatation of all transformation rules for Viatra

```
namespace mutex;
import mutex.metamodel;

machine lib
{
  rule cleanModel() =
    let Model = undef, P1 = undef, P2 = undef, N1 = undef, N2 = undef in seq
  {
    println(clean(mutex.model));
    new (entity(Model) in mutex);
    rename(Model, "model");
    new (mutex.metamodel.process(P1) in Model);
    new (mutex.metamodel.process(P2) in Model);
    new (mutex.metamodel.process.next(N1, P1, P2));
    new (mutex.metamodel.process.next(N2, P2, P1));
  }

  pattern newRule_lhs(P1, P2, N) =
  {
    process(P1);
    process(P2);
    process.next(N, P1, P2);
  }
  rule newRule(in P1, in P2, in N, out P, out N1, out N2) = seq{
    new(process(P) in mutex.model);
    new(process.next(N1, P1, P));
    new(process.next(N2, P, P2));
    delete(N);
  }

  pattern killRule_lhs(P1, P2, P, N1, N2) =
  {
```

```
  process(P1);
  process(P2);
  process(P);
  process.next(N1, P1, P);
  process.next(N2, P, P2);
}
rule killRule(in P1, in P2, in P, in N1, in N2, out N) = seq
{
  new(process.next(N, P1, P2));
  delete(P);
  delete(N1);
  delete(N2);
}

pattern mountRule_lhs(P) =
{
  process(P);
}
rule mountRule(in P, out R, out T) = seq
{
  new (resource(R) in mutex.model);
  new (resource.token(T, R, P));
}

pattern unmountRule_lhs(P, R, T) =
{
  process(P);
  resource(R);
  resource.token(T, R, P);
}
rule unmountRule(/*in P, */in R, in T) = seq
{
  delete(R);
  delete(T);
}

pattern passRule_lhs(P1, P2, R, T) =
{
  process(P1);
  process(P2);
  process.next(N, P1, P2);
  resource(R);
  resource.token(T, R, P1);
  neg pattern req(P1, R) = {
    process(P1);
    resource(R);
    process.request(Req, P1, R);
  }
}
rule passRule(/*in P1, */in P2, in R, in T, out T2) = seq
{
  new(resource.token(T2, R, P2));
  delete(T);
}

pattern requestRule_lhs(P, R) =
{
  process(P);
  resource(R);
  neg pattern held(R, P) = {
    process(P);
    resource(R);
    resource.held_by(HB, R, P);
  }
  neg pattern req(P) = {
    process(P);
    resource(Rn);
    process.request(Reqn, P, Rn);
  }
}
rule requestRule(in P, in R, out Req) = seq
{
  new(process.request(Req, P, R));
}
```

```

pattern takeRule_lhs(P, R, T, Req) =
{
  process(P);
  resource(R);
  process.request(Req, P, R);
  resource.token(T, R, P);
}
rule takeRule(in P, in R, in T, in Req, out HB) = seq
{
  new(resource.held_by(HB, R, P));
  delete(T);
  delete(Req);
}

pattern releaseRule_lhs(P, R, HB) =
{
  process(P);
  resource(R);
  resource.held_by(HB, R, P);
  neg pattern req(P) = {
    process(P);
    resource(Rn);
    process.request(Reqn, P, Rn);
  }
}
rule releaseRule(in P, in R, in HB, out Rel) = seq
{
  delete(HB);
  new(resource.release(Rel, R, P));
}


pattern giveRule_lhs(P1, P2, R, Rel) =
{
  process(P1);
  process(P2);
  process.next(N, P1, P2);
  resource(R);
  resource.release(Rel, R, P1);
}
rule giveRule(in P2, in R, in Rel, out T) = seq
{
  delete(Rel);
  new(resource.token(T, R, P2));
}

pattern blockedRule_lhs(P1, P2, R, Req, HB) =
{
  process(P1);
  resource(R);
  process.request(Req, P1, R);
  process(P2);
  resource.held_by(HB, R, P2);
}
rule blockedRule(in P1, in R, out B) = seq
{
  new(resource.blocked(B, R, P1));
}

pattern waitingRule_lhs(P1, P2, R1, R2, Req, B) =
{
  process(P2);
  resource(R1);
  process.request(Req, P2, R1);
  process(P1);
  resource.held_by(HB, R1, P1);
  resource.blocked(B, R2, P1);
  resource(R2);
}
rule waitingRule(in P2, in R2, in B, out BN) = seq
{
  new(resource.blocked(BN, R2, P2));
  delete(B);
```

```
}

pattern ignoreRule_lhs(P, R, B) =
{
  process(P);
  resource(R);
  resource.blocked(B, R, P);

  neg pattern hold(P) = {
    process(P);
    resource(RN);
    resource.held_by(HBN, RN, P);
  }
}
rule ignoreRule(in B) = seq
{
  delete(B);
}

pattern unlockRule_lhs(P, R, HB, B) =
{
  process(P);
  resource(R);
  resource.held_by(HB, R, P);
  resource.blocked(B, R, P);
}
rule unlockRule(in P, in R, in HB, in B, out Rel) = seq
{
  new(resource.release(Rel, R, P));
  delete(B);
  delete(HB);
}



pattern mountStarRule_lhs(P) =
{
  process(P);
}
rule mountStarRule(in P, out R, out HB) = seq
{
  new(resource(R) in mutex.model);
  new(resource.held_by(HB, R, P));
}

pattern requestStarRule_lhs(P1, P2, R1, R2) =
{
  process(P1);
  process(P2);
  process.next(N, P2, P1);
  resource(R1);
  resource.held_by(H1, R1, P1);
  resource(R2);
  resource.held_by(H2, R2, P2);
  neg pattern req(P1, R2) = {
    process(P1);
    resource(R2);
    process.request(Rqn, P1, R2);
  }
}
rule requestStarRule(in P1, in R2, out Req) = seq
{
  new(process.request(Req, P1, R2));
}

pattern releaseStarRule_lhs(P1, P2, R1, R2, H1) =
{
  process(P1);
  resource(R1);
  process.request(RQ, P1, R1);
  process(P2);
  resource.held_by(H1, R1, P2);
  resource(R2);
  resource.held_by(H2, R2, P2);
}
rule releaseStarRule(in P2, in R1, in H1, out RL) = seq
{
  new(resource.release(RL, R1, P2));
  delete(H1);
}

pattern requestSimpleRule_lhs(P, R) =
{
  process(P);
  resource(R);
  resource.token(T, R, P);
  neg pattern req(P, R) = {
    process(P);
    resource(R);
    process.request(Reqn, P, R);
  }
}
rule requestSimpleRule(in P, in R, out Req) = seq
{
  new(process.request(Req, P, R));
}

}
```

A.1.4 VTCL machine running the STS case for Viatra

```
namespace mutex;
import mutex.metamodel;

machine sts
{
  rule main(in Size) = let FireCount = 0, StartTime = systime() in seq {
    // mutex benchmark, STS
    // see http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf

    // step1: new processes
    let StepCount = 0 in iterate
    if(StepCount < Size-2) seq{
      update StepCount = StepCount+1;

      choose P1, P2, N with find mutex.lib.newRule_lhs(P1, P2, N) do
        let P = undef, N1 = undef, N2 = undef in seq{
          call mutex.lib.newRule(P1, P2, N, P, N1, N2);
          update FireCount = FireCount +1;
      }
    } else fail;

    // step2: mount a single resource
    choose P with find mutex.lib.mountRule_lhs(P) do
      let R = undef, T = undef in seq {
        call mutex.lib.mountRule(P, R, T);
        update FireCount = FireCount +1;
    }


    // step3
    forall P, R with find mutex.lib.requestRule_lhs(P, R) do
      let Req = undef in seq {
        call mutex.lib.requestRule(P, R, Req);
        update FireCount = FireCount +1;
    }

    // step4
    iterate seq{
      choose P, R, T, Req with find mutex.lib.takeRule_lhs(P, R, T, Req) do
        let HB = undef in seq {
          call mutex.lib.takeRule(P, R, T, Req, HB);
          update FireCount = FireCount +1;
      }
      choose P, R, HB with find mutex.lib.releaseRule_lhs(P, R, HB) do
        let Rel = undef in seq {
          call mutex.lib.releaseRule(P, R, HB, Rel);
          update FireCount = FireCount +1;
        }
        choose P1, P2, R, Rel with find mutex.lib.giveRule_lhs(P1, P2, R, Rel) do
          let T = undef in seq {
            call mutex.lib.giveRule(P2, R, Rel, T);
            update FireCount = FireCount +1;
        }
      };

      //println("step 4 done");
      print("DONE: " +
        FireCount + " rules fired in " +
        (systime() - StartTime) + "ms; ");

      //cleaning
      call mutex.lib.cleanModel();
    }

}
```

A.1.5 GrGen.NET implementation

Delivered with the GrGen.NET distribution as an example.

A.2 Petri net simulation benchmark

A.2.1 Metamodel for Viatra

Delivered with the Viatra DSM framework.

A.2.2 Generated model for Viatra

See [6].

A.2.3 VTCL machine for Viatra

This machine is a common Viatra example, it was not implented by me.

```
namespace DSM.machines.PetriNet;

import DSM.metamodel.PetriNet.PetriNetEditor;
```

```

machine 'PetriNetSimulator'
{
  // 'Transition' is a transition of the petri net 'PN'.
  pattern petriTransition(PN, Transition) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Transition'(Transition);
    'PetriNet'.'transitions'(X, PN, Transition);
  }

  // 'Place' is a source place for transition 'Transition'.
  pattern sourcePlace(Transition, Place) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Transition'(Transition);
    'PetriNet'.'transitions'(X1, PN, Transition);
    'PetriNet'.'Place'(Place);
    'PetriNet'.'places'(X2, PN, Place);
    'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
  }

  // 'Place' is a target place for transition 'Transition'.
  pattern targetPlace(Transition, Place) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Transition'(Transition);
    'PetriNet'.'transitions'(X1, PN, Transition);
    'PetriNet'.'Place'(Place);
    'PetriNet'.'places'(X2, PN, Place);
    'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
  }

  // 'Place' contains a token 'Token' linked to it
  pattern placeWithToken(Place, Token) =
  {
    'PetriNet'.'Place'(Place);
    'PetriNet'.'Place'.'Token'(Token) in Place;
    'PetriNet'.'Place'.'tokens'(X, Place, Token);
  }

  // Transition is fireable
  pattern isTransitionFireable_flattened(Transition) =
  {
    'PetriNet'.'Transition'(Transition);
    neg pattern notFireable_flattened(Transition) =
    {
      'PetriNet'.'Place'(Place);
      'PetriNet'.'Transition'(Transition);
      'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
      neg pattern placeToken(Place) =
      {
        'PetriNet'.'Place'(Place);
        'PetriNet'.'Place'.'Token'(Token);
        'PetriNet'.'Place'.'tokens'(X, Place, Token);
      }
    }
    or
    {
      'PetriNet'.'Place'(Place);
      'PetriNet'.'Transition'(Transition);
      'PetriNet'.'Place'.'InhibitorArc'(OutArc, Place, Transition);
      'PetriNet'.'Place'.'Token'(Token);
      'PetriNet'.'Place'.'tokens'(X, Place, Token);
    }
  }

  // Transition is fireable in PetriNet
  pattern fireable(PetriNet, Transition) =
  {
    find petriTransition(PetriNet, Transition);
    find isTransitionFireable_flattened(Transition);
  }

```

```
rule addToken(in Place) = let Token = undef, X = undef in seq
{
  new('PetriNet'.'Place'.'Token'(Token) in Place);
  new('PetriNet'.'Place'.'tokens'(X, Place, Token));
}


rule fireTransition(in Transition) = seq
{
  forall Place with find sourcePlace(Transition, Place) do
    choose Token with find placeWithToken(Place,Token) do delete(Token);
  forall Place with find targetPlace(Transition, Place) do
    call addToken(Place);
  update counter("firings") = counter("firings") + 1;
}



asmfunction counter/1;

// entry point
rule main(in NetFQN, in Iterations) = let Start = 0 in seq
{
  update counter("iterations") = 0;
  update counter("firings") = 0;
  update Start = systime();
  let PN = ref(NetFQN) in iterate seq
  {
    update counter("iterations") = counter("iterations") + 1;
    if (counter("iterations") > Iterations) fail;
    choose  T with find fireable(PN,T) do seq
    {
      call fireTransition(T);
    }
  }
  println("Simulation ended, fired " +
    counter("firings") + " transitions in " +
    (counter("iterations")-1) + " iterations in "+
    (systime()-Start)+ " msec.");
}


}

```

A.2.4 GrGen.NET graph model (PetriModel.gm)

```
node class PetriNet;

node class Place;

node class Transition;

node class Token;

edge class places
  connect PetriNet[*] -> Place[0:1];

edge class transitions
  connect PetriNet[*] -> Transition[0:1];

edge class tokens
  connect Place [*] -> Token[1];

edge class inArc
  connect Transition [*] -> Place [*];

edge class outArc
  connect Place [*] -> Transition [*];

edge class inhibitorArc
  connect Place [*] -> Transition [*];
```

A.2.5 GrGen.NET graph rules (Petri.grg)

```
using PetriModel;

rule fireRule {
  pattern {
    net:PetriNet -ts:transitions-> t:Transition;
    negative {
      t <-o:outArc- pEmpty:Place;
      negative {
        pEmpty -ksn:tokens-> ;
      }
    }
    negative {
      t <-i:inhibitorArc- pFilled:Place -ks:tokens-> ;
    }
  }
  modify {
    exec( [emitTokens(t)] && [consumeTokens(t)] );
  }
}

rule emitTokens(t:Transition) {
  pattern {
    t -i:inArc-> p:Place;
  }
  modify {
    p -ks:tokens-> k:Token;
  }
}
rule consumeTokens(t:Transition) {
  pattern {
    t <-o:outArc- p:Place;
  }
  modify {
    exec(deleteToken(p));
  }
}
```

```
rule deleteToken(p:Place) {
  pattern {
    p -ks:tokens-> k:Token;
  }
  replace{
    p;
  }
}
```

A.2.6 VTCL machine for generating GrShell scripts from Petri nets

```
namespace DSM.machines.PetriNet;

import DSM.metamodel.PetriNet.PetriNetEditor;

machine petrinet2grgen{
model space// 'Transition' is a transition of the petri net 'PN'.
model spacepattern petriTransition(PN, Transition) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Transition'(Transition);
    'PetriNet'.'transitions'(X, PN, Transition);
  }

  // 'Place' is a place of the petri net 'PN'.
  pattern petriPlace(PN,Place) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Place'(Place);
    'PetriNet'.'places'(X2, PN, Place);
  }

  // 'Place' is a source place for transition 'Transition'.
  pattern sourcePlace(Transition, Place, OutArc) =
  {
    'PetriNet'(PN);
    'PetriNet'.'Transition'(Transition);
  'PetriNet'.'transitions'(X1, PN, Transition);
  'PetriNet'.'Place'(Place);
  'PetriNet'.'places'(X2, PN, Place);
  'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
}

// 'Place' is a target place for transition 'Transition'.
pattern targetPlace(Transition, Place, InArc) =
{
  'PetriNet'(PN);
  'PetriNet'.'Transition'(Transition);
  'PetriNet'.'transitions'(X1, PN, Transition);
  'PetriNet'.'Place'(Place);
  'PetriNet'.'places'(X2, PN, Place);
  'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
}

// 'Place' contains a token 'Token' linked to it
pattern placeWithToken(Place, Token) =
{
  'PetriNet'.'Place'(Place);
  'PetriNet'.'Place'.'Token'(Token) in Place;
  'PetriNet'.'Place'.'tokens'(X, Place, Token);
}

asmfunction counter/1;

rule generatePlaces(in PN) =
forall P below PN with find petriPlace(PN,P) do seq  {
  println("new place_"+name(P)+":Place");
  println("new net -:places-> place_"+name(P));

  call generateTokens(P);
}

rule generateTokens(in P) =
forall Token below P with find  placeWithToken(P,Token) do seq {
  println("new token_"+name(Token)+":Token");
  println("new place_"+name(P)+" -:tokens-> token_"+name(Token));
}

rule generateTransitions(in PN) =
forall T below PN with find petriTransition(PN,T) do seq {
  println("new transition_"+name(T)+":Transition");
  println("new net -:transitions-> transition_"+name(T));
}

rule generateArcs(in PN) = seq {
  forall T below PN, Place below PN, OutArc below PN
    with find sourcePlace(T, Place, OutArc) do
      println("new place_" + name(Place) +
        " -outarc_" + name(OutArc) +
        ":outArc-> transition_" + name(T));
  forall T below PN, Place below PN, InArc below PN
    with find targetPlace(T, Place, InArc) do
      println("new transition_" + name(T) +
        " -inarc_" + name(InArc) +
        ":inArc-> place_" + name(Place));
}


rule main(in InputPetriNet, in Iterations) = let PN = ref(InputPetriNet) seq {
  println("new graph \"Petri.grg\"");
     println("new net:PetriNet");

  call generatePlaces(PN);
  call generateTransitions(PN);
  call generateArcs(counter(PN);

  println("custom graph analyze");
     println("custom actions " +
       "gen_searchplan fireRule emitTokens consumeTokens deleteToken");
  println("xgrs fireRule["+Iterations+"]");
  println("quit");
  }



}
```

A.3 Combinatorical explosion synthetic benchmark

A.3.1 Metamodel for Viatra

```
namespace pmintensive;

entity(metamodel) {
  entity(a);
}
entity(model);
```

A.3.2 VTCL machine for Viatra

```
import pmintensive.metamodel;

machine pmintensive_test2
{

  pattern a4(A1, A2, A3, A4) =
  {
    a(A1);
    a(A2);
    a(A3);
    a(A4);
  }
  pattern a4_b(A1, A2, A3, A4) =
  {
    a(A1);
    a(A2);
    a(A3);
    a(A4);
  }

  rule main() = let StartTime = undef, CommitTime = undef, StopTime = undef in seq
  {
    //warmup
    try choose A1, A2, A3, A4 with find a4(A1, A2, A3, A4) do skip;
    try choose A1, A2, A3, A4 with find a4_b(A1, A2, A3, A4) do skip;

    update StartTime = systime();

    let Iter = 40 in iterate if (Iter > 0) let A=undef in seq {
      update Iter = Iter - 1;

      new (a(A) in pmintensive.model);
    } else fail;

    update CommitTime = systime();
    print("Commit reached in  " + (CommitTime-StartTime)+ " msec.");

    try choose A1, A2, A3, A4 with find a4(A1, A2, A3, A4) do skip;
    try choose A1, A2, A3, A4 with find a4_b(A1, A2, A3, A4) do skip;

    update StopTime = systime();
    println(" finished in " + (StopTime-StartTime)+ " msec.");
  }
}
```

Bibliography

[1] GrGen.NET homepage, 5 2008. `http://www.info.uni-karlsruhe.de/software/grgen`.

[2] Simulink homepage, 5 2008. `http://www.mathworks.com/products/simulink`.

[3] 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.

[4] Don Batory. The LEAPS algorithm. Technical Report CS-TR-94-28, 1, 1994.

[5] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A first experimental evaluation of search plan driven graph pattern matching. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume NN of *LNCS*. Springer, 2008. http://www.springerlink.com/content/105633/.

[6] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *International Conference on Graph Transformation*, 2008. Accepted.

[7] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the VIATRA model transformation system. In Gabor Karsai and Gabi Taentzer, editors, *Graph and Model Transformation (GraMoT 2008)*. ACM, 2008.

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

[9] 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.

[10] 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.

[11] 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.

[12] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló 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)*.

[13] Mark Proctor et al. *Drools Documentation*. JBoss. `http://labs.jboss.com/drools/documentation.html`.

[14] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.

[15] The VIATRA2 framework. official website, 2008. `http://viatra.inf.mit.bme.hu`.

[16] L Geiger, C Schneider, and C Reckord. Template- and modelbased code generation for mda-tools, in. In *In: 3rd International Fujaba Days 2005*, pages 57–62, 2005.

[17] 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.

[18] Scott E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[19] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. *Business Process Execution Language for Web Services version 1.1*, 2002. `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`.

[20] M. Jungel, E. Kindler, and M. Weber. The petri net markup language. In *In S. Philipi, editor, Algorithmen und Werkzeuge fur Petrinetze (AWPN), Koblenz*, June 2002.

[21] Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt, editors, *Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005)*, October 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/`.

[22] Peter Lin. System and method to distribute reasoning and pattern matching in forward and backward chaining rule engines. US Patent application USPTO 20050246301, 02 2005.

[23] 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.

[24] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.

[25] 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.

[26] 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.

[27] Object Management Group. *Action Semantics for the UML*, August 2001. `http://www.omg.org`.

[28] Object Management Group. *Model Driven Architecture — A Technical Perspective*, September 2001. `http://www.omg.org`.

[29] Object Management Group. *Object Constraint Language Specification (in UML 1.4)*, 2001. `http://www.omg.org`.

[30] Object Management Group. *Meta Object Facility Version 2.0*, 2003. `http://www.omg.org`.

[31] Object Management Group. *UML Semantics Version 2.0*, May 2003. `http://www.omg.org`.

[32] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In *Proceedings of 1st International Conference on Model Transformation*, LNCS. Springer. In Press.

[33] 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.

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

[35] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *6th International Workshop on Theory and Application of Graph Transformations*, volume 1764, Berlin, 2000. Springer-Verlag.

[36] A. Schürr. Introduction to PROGRES, an attributed graph grammar based specification language. In M. Nagl, editor, *Graph–Theoretic Concepts in Computer Science*, volume 411 of *LNCS*, pages 151–165, Berlin, 1990. Springer.

[37] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.

[38] 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.

[39] 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.

[40] Gergely Varró. *Advanced Techniques for the Implementation of Model Transformation Systems*. PhD thesis, Budapest University of Technology and Economics, 2008.

[41] 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`.

[42] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 05)*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Press.

[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