Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Transitive reachability for efficient event-driven model transformations

MSc thesis

Candidate
Tamás Szabó

Advisor
Gábor Bergmann

December 3, 2012

FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

HALLGATÓI NYILATKOZAT

Budapest, December 3, 2012

Szabó Tamás
hallgató

Kivonat

Napjainkban a modellalapú szoftverfejlesztési módszerek egyre nagyobb szerepet kapnak, mert a hardver és szoftver rendszerek komplexitása sokszor túlmutat azon a szinten, amely még a hagyományos fejlesztési paradigmákkal kezelhető. Ez a megközelítés szakterület specifikus fejlesztési környezetek létrehozásával lehetővé teszi, hogy a szakterület ismerői - de nem feltétlenül csak szoftvermérnökök - is részt tudjanak venni a gyors prototipizálásban, felgyorsítva ezzel a fejlesztés ütemét. A modellalapú szoftverfejlesztéshez szorosan kapcsolódik a modelltranszformáció is, amellyel modellek közötti automatikus leképezést tudunk megvalósítani, például kódgenerálást megvalósítva ezzel. Az elmúlt években történt kutatások egyik fontos irányvonala volt ez a terület az informatikán belül, de a szakterületi modellező nyelvek kifejlesztése még mindig sokszor igényli több terület mélyebb ismeretét, úgy mint metamodellezés, transzformációk, nyelvtervezés.

A diplomaterv készítése során a modellezési környezetnek az Eclipse Modeling Framework-öt választottam, amely ipari standardnak tekinthető a (meta)modellezés területén. Erre épít az EMF-IncQuery keretrendszer, amely lehetővé teszi modellek fölötti deklaratív lekérdezések hatékony megvalósítását. A hatékonyságot az inkrementális gráfminta-illesztés segítségével éri el, amely azt jelenti, hogy a lekérdezések eredményeit egy folyamatosan (minden modellmódosítás után) karbantartott gyorsítótárból tudja megválaszolni.

A diplomaterv egyik fő eredménye az EMF-IncQuery-re építő eseményvezérelt szabályvégrehajtó motor elkészítése. Ezzel a kiegészítéssel lehetőség nyílik arra, hogy lekérdezések eredményhalmazának változásakor előre definiált akciókat lehessen automatikusan végrehajtani (tetszőleges Java nyelven leírt lépéssorozat). Sok gyakorlati életben előforduló problémában szükség van a vizsgált modellen történő tranzitív lezárt számítására, ezért a munka során nagy hangsúlyt kapott a mintanyelv tranzitív lezárttal, mint nyelvi elemmel történő kiterjesztése. Ehhez többféle tranzitív lezáró algoritmust implementáltam és hasonlítottam össze a hatékonyságukat szintetikus teljesítményteszteken keresztül. A diplomaterv másik fontos koncepcionális eredménye egy, a gráf erősen összefüggő komponenseinek inkrementális karbantartásán alapuló régebbi algoritmus átdolgozott változata, amely a leghatékonyabbnak bizonyult az általam vizsgált algoritmusok közül.

Az elkészült nyelvi kiegészítés és eseményvezérelt szabályvégrehajtó motor hatékonyságát többféle esettanulmányon keresztül vizsgáltam meg; EMF-IncQuery validációs keretrendszere, Peer-to-Peer VoIP rendszerek sztochasztikus szimulációja és egy egyszerű Design Space Exploration komponens kialakítása cloud-infrastruktúra modellek futási idejű rekonfigurációjához.

Abstract

Nowadays model driven software development (MDSD) is getting more and more important as usually the complexity of hardware and software systems is far beyond the limit which can be handled easily by traditional paradigms. This approach is an alternative to create domain-specific development environments to help the rapid prototyping for not only software engineers but for experts in the given domain too, speeding up the whole development lifecycle of the product. Model transformation is tightly connected to MDSD as it automates the mapping between different kinds of models during development. The MDSD related technologies have been an important research area in the past years, but creating domain-specific tools still requires expertise in a wide range of areas: metamodeling, transformations, language design.

The thesis relies on the Eclipse Modeling Framework as the modeling environment which is an industry standard platform used for (meta)modeling. EMF-IncQuery is based on this technology and provides a pattern language to evaluate declarative queries over EMF models efficiently. This efficiency is due to incremental graph pattern matching, that is, the queries are answered using a cache that is continuously maintained after each model manipulation.

One of the main conceptual results of the thesis is the EMF-IncQuery backed event-driven rule execution engine. This extension allows attaching predefined actions (written in Java) to queries and automatically executing them when a new match of a query appears (or disappears). Nevertheless, in many practical problems transitive closure of models is widely used which requires to extend the pattern language of EMF-IncQuery. During the thesis work I have investigated various incremental and non-incremental transitive closure algorithms and created prototype implementations to compare their runtime characteristics using synthetic performance tests. As part of the work, I have adapted an algorithm which is based on the incremental maintenance of the strongly connected components of the graph and it came out to be the most efficient out of the ones that I have implemented.

To illustrate the practical applications of the results I have carried out three case studies as part of the thesis work: the validation framework of EMF-IncQuery, stochastic simulation of Peer-to-Peer VoIP networks and a basic module for Design Space Exploration for runtime reconfiguration of cloud-infrastructure models.

Contents

Chapter 1

Introduction

Model driven software development (MDSD) is getting more and more important nowadays as the complexity of many software and hardware systems is far beyond the limit that can be handled by traditional development paradigms. MDSD is about using domain-specific languages to create models that express application structure and interactions between its components. The developers work at a higher abstraction level during most of the development cycle thus allowing non-experts to also understand the interaction between the modules of the system. This approach has also the benefit to grasp only the crucial parts of the system architecture without delving into much unnecessary details. In the MDSD context the aim is that developers deal only with high level models and additionally use tools that can automate the other development tasks to the highest possible degree.

The models (either in some form of textual or graphical representation) are easily readable by end-users; however they need to be translated to the target platform in order to run. For high level models, model transformation is used, where we define transformation rules for each model element. A transformation rule consists of a pre- and postcondition; when a model element matches the precondition then the action defined in the postcondition can be applied and the model element will be transformed. In general, the term model-to-model transformation is used; a special case of this is model-to-text transformation which results source code or documentation generation from the high level domain-specific models. Model transformation can also be used for validation purposes; to check well-formedness requirements, enforce project specific design principles and derive metrics and other properties from the high level model. For example, well-formedness checker transformation rules have a precondition which defines the model elements and their connection which must not appear in the model, however, if they do, then the postcondition can be applied and an appropriate error marker can be placed on the user interface of the developer. Nevertheless, in the domain of safety critical systems it is often required that the system's behavior must be checked with certain formal verification techniques (e.g. for Safety Integrity Level 4 the use of formal methods is highly recommended [26]). In this scenario Petri nets can serve as a high level model, which later can be used by model checking tools as described in our paper [39].

The Eclipse Modeling Framework (EMF) is an industry standard Eclipse-based [7] modeling framework for defining models and provides code generation facilities for building tools and other applications based on the structured data model. It can be also easily integrated with graphical editors thus allowing the creation of intuitive interfaces for model manipulations. Because of these properties, in the thesis I will use EMF as the target platform for modeling purposes.

Requirements against a model transformation framework

Model transformation requires evaluating certain queries on the source models. However, when dealing with evolving models, it is often advantageous if we do not need to start over in batch mode after every model modification, rather perform predefined actions in an event-driven manner (for example upon query-match appearance or disappearance). Such frameworks address this issue by defining a pattern language to form the queries and evaluate them efficiently on the input models. It should support the definition of the postcondition action which can be, for example, model manipulation, displaying some error message or code generation. Rule based expert systems share a lot of common properties with model transformation frameworks, however they lack EMF support which is the target modeling platform of my thesis.

On the other hand, EMF-IncQuery [4] is a framework for defining declarative queries over EMF models (treated as a graph model), and executing them efficiently without manual coding. The query language is based on the concepts of graph patterns (which is widely used in many graph-based model transformation frameworks). High performance is achieved by adapting the fundamentals of incremental graph pattern matching, which would be a great advantage in a graph transformation framework as it provides low latency with a slight memory overhead. One of the main contributions of my thesis work is the extending of the EMF-IncQuery framework with the so called 'Rule Engine'. This additional module will allow the user to define transformation rules with precondition as an EMF-IncQuery pattern and a postcondition written in the Java programming language. These rules can be executed in an event-driven manner; when the source model is modified it activates some rules and they can be fired (like in the rule based expert systems) to perform transformation activities.

Extending the pattern language of EMF-IncQuery

EMF-IncQuery has a pattern language with the expressiveness of the first order logic, thus it is not possible to express recursive patterns with it, however it would be highly desired in a lot of cases, for example when computing transitive closures over models. Transitive closures are frequently used in a number of modeling applications, e.g. to compute model partitions or reachability regions in traceability model management, business process model analysis, or stochastic simulation of complex systems. They may also provide the underpinnings for n-level metamodeling hierarchies where transitive type-subtype-instance relationships need to be maintained. During the thesis work I have experimented with various (incremental) transitive closure algorithms and adapted the one presented in [31] into the EMF-IncQuery framework. The adaptation had three main requirements: (i) align with the general performance characteristics of incremental graph pattern matching, (ii) handle cyclic closures (closures in general graphs with cycles) correctly and (iii) support generic transitive closures, i.e. the ability to compute closures of not only simple graph edges (edge types), but also derived edges defined by binary graph patterns that establish a logical link between a source and a target vertex. These contributions allow to use EMF-IncQuery as a basic graph transformation environment with full support for transitive closure in the pattern language.

Structure of the thesis

The rest of the paper is structured as follows; Chapter 2 introduces (meta)modeling, graph transformation and transitive closure related preliminaries necessary to understand the rest of the discussion, and describes briefly the running example - the stochastic simulation of Peer-to-peer VoIP networks. Chapter 3 presents the various transitive closure algorithms I have experimented with and also discusses the most interesting implementation details. Chapter 4 gives an in-depth introduction to the EMF-IncQuery based Rule- and Trigger Engine, while Chapter 5 presents three case studies in-depth to demonstrate the usage of the Rule Engine and the language extension. Detailed measurement results can be found in Chapter 6 on the transitive closure algorithms and measurements concerning the case studies. Finally, the summary of the thesis is given in Chapter 7 along with notes on future work.

Chapter 2

Preliminaries

The following sections give a detailed overview about the preliminaries required to understand the latter chapters of the thesis:

- Details on the running example in Section 2.1
- Modeling, metamodeling in Section 2.2.
- Introduction to graph patterns and graph transformations in Section 2.3.
- The transitive closure related definitions and examples in Section 2.4.
- Discussion about the Union-Find data structure in Section 2.5. The data structure is used by a transitive closure algorithm, which will be discussed later.
- A brief introduction in Section 2.6 of the EMF-IncQuery Base library which provides powerful query operations for EMF models.

2.1 Running example - stochastic simulation of Peer-to-peer VoIP networks

In distributed and mobile systems with volatile bandwidth and fragile connectivity, non-functional aspects like performance and reliability become more and more important. To measure and profile these properties, stochastic methods are required. These systems, however, are characterized with high degree of architectural reconfiguration. Viewing the architecture of such systems as a graph, the problem is naturally evaluated with graph transformations.

In the thesis, I will use the stochastic simulation of Peer-to-Peer VoIP systems (for example Skype [13]) as a demonstrating example. In such systems, the network is modelled as a graph with nodes of two types: clients and super nodes. The simple client nodes represent the users of the VoIP network and in the presence of more hardware resources they can be upgraded to work as super nodes, which operates as a router and manager between clients. A connection between two nodes represent a link which can be used for communication. Super nodes form an overlay network over the physical one and each

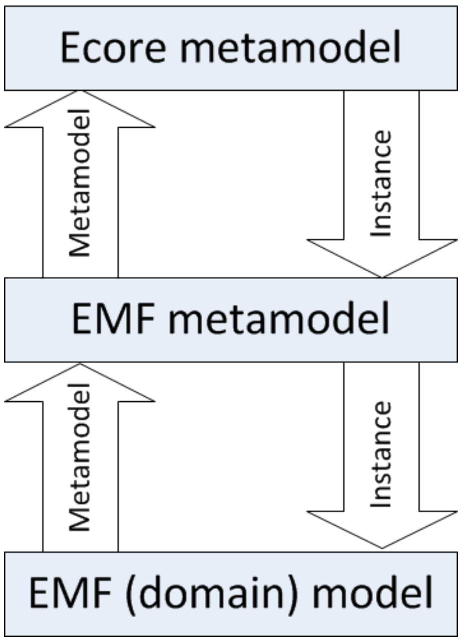


Figure 2.1: *Three-tier model-metamodel relationship*

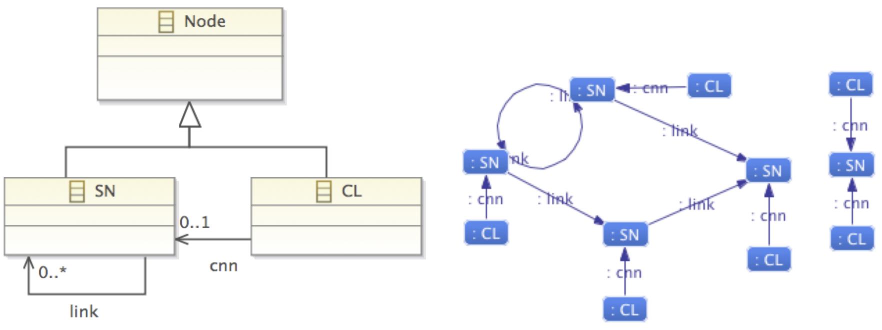
client registers itself to a super node, through which it will establish and receive calls, send messages, etc. The only central component in the network is the registration server, which stores the user account information, handles the login procedure on the network and knows the addresses of the super nodes. All other information regarding the users' status and connection is stored in a distributed manner among the super nodes. If a super node becomes unavailable because of hardware or link failure, the connected clients must reconnect to another super node. Naturally, the aim is to avoid node-isolation, such node cannot establish connection with another client.

Measuring the non-functional properties is crucial is such scenarios to maintain a certain level of quality of service when the prototype system is deployed onto the Internet for daily use. Stochastic simulation as a case study is discussed in Section 5.3 in-depth, while the following sections give details about the key aspects of the domain of stochastic simulation.

2.2 Modeling preliminaries and the running example

A model is a projection of reality and is used to capture concepts related to the engineer's task. Models are frequently used to describe a system and the communication between the various components. Every model has a metamodel, which defines the concepts within the given domain. A model conforms to its metamodel in the same way that a computer program conforms to the grammar of the given programing language. A metamodel serves as an abstract syntax and defines the possible elements and their connection for its instance models.

The Eclipse Modeling Framework [36] is the leading industrial modeling ecosystem which is an easy-to-use platform for creating domain specific tools. It provides code generation facilities and tooling (persistence, editing, notification handling, etc.) for Java representation



(a) *Metamodel* **(b)** *Instance model*

Figure 2.2: *Meta- and instance models of the running example*

of models. The model of EMF models is the Ecore metamodel; every EMF model conforms to this metamodel, which results a three-tier model-metamodel relationship between the domain instance model (edited by the end-user), its metamodel and the Ecore metamodel (see Figure 2.1).

Models and metamodels in the context of the running example

A simple metamodel for the running example is given in Figure 2.2a. The network is defined as a graph that may consist of two kinds of nodes: they may either be simple clients (instances of the type CL) or super nodes (SN) to which clients may connect (through connections of type *cnn*). Additionally, super nodes form the overlay network by connecting to each other through connections of type *link*. A sample instance model is shown in Figure 2.2b.

2.3 Graph patterns, graph transformations and the RETE net

For querying domain models, we often use graph patterns that constitute an expressive formalism also used for various other purposes in model-driven development, such as capturing general-purpose model queries including model validation constraints, or defining the behavioral semantics of dynamic domain-specific languages. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of structural constraints prescribing the existence of vertices and edges of a given type. Languages usually include a way to express attribute constraints. A negative application condition (NAC) defines cases when the original pattern is not valid (even if all other constraints are met), in the form of a negative sub-pattern. A match (in the matching set) of a graph pattern is a group of model elements that have the exact same configuration as the pattern, satisfying all the constraints (except for NACs, which must be made unsatisfiable).

A sample graph pattern is shown in Figure 2.3 from the running example, using a graphical concrete syntax for illustration. This pattern expresses the logical linked relationship between any two super nodes $S1$ and $S2$ (both are required to be of type SN) that are

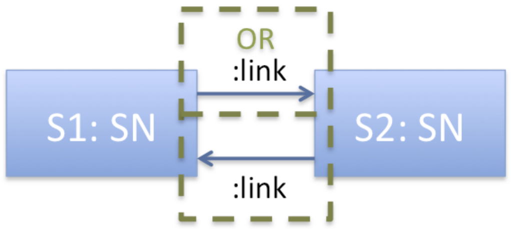


Figure 2.3: *Graph pattern example*

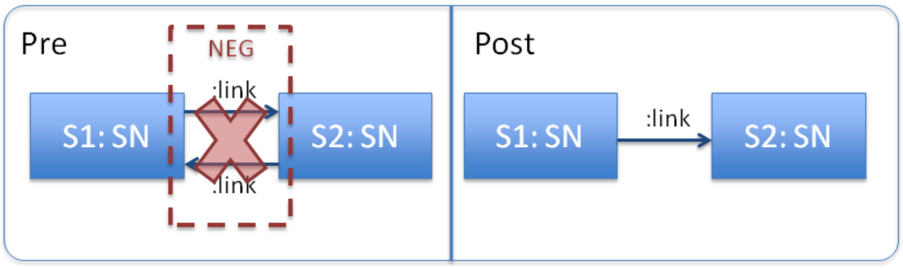


Figure 2.4: *Graph transformation rule example*

connected by a relation of type link in either direction.

Graph transformation is defined by a graph pattern and an action to perform manipulation of the source model. The transformation is done by replacing some parts of the graph by another graph. Formally, a graph transformation rule is a triplet $r = (LHS, RHS, NAC)$, where

- LHS is the left-hand side which defines the graph pattern that will be replaced
- RHS is the right-hand side which is the graph that will be glued to the model
- NAC is the negative application condition which prohibits the presence of certain model elements and connections between them. If a match of the NAC can be found then the rule can not be executed.

The NAC and LHS together is called the precondition of the rule and RHS is the postcondition. The application of the rule r on an instance model will remove a sub-graph which maps to LHS from the model and glue an image of RHS to it. A graph transformation is a series of such rule applications from the initial model. A sample graph transformation rule is given in Figure 2.4 where the left hand side pattern contains a NAC, that is, the rule can only be applied if there is no *linked* connection between the two super nodes. As a result of the rule execution, a *linked* relationship will be established between the two nodes in either direction.

Graph transformation systems use pattern matching algorithms to determine the parts of the model that correspond to the match set of a graph pattern. However, pattern matching is usually the most expensive (in terms of computation) part of the transformation, resulting to be crucial in the performance characteristics of the underlying engine. Incremental pattern matching engines (like EMF-IncQuery) aim to solve this issue by maintaining a cache in which the matches of a pattern are stored explicitly. The match set is readily available from the cache at any time without searching, and the cache is incrementally updated whenever changes are made to the model. The result can be retrieved in constant time - excluding the linear cost induced by the size of the match set itself -, making pattern matching extremely fast. The trade-off is space consumption of the match set caches, model manipulation performance overhead related to cache maintenance, and possibly the initialization cost of the cache.

RETE [21] is a well-known incremental pattern matching technique from the field of rule-based expert systems. A RETE net consists of RETE nodes (not to be confused with the vertices of the model graph), each storing a relation corresponding to the match set of a partial pattern, i.e. the set of model element tuples that satisfy a given subset of pattern constraints. RETE nodes are connected by RETE edges so that the content of a RETE node can be derived from its parent nodes. The RETE edges propagate incremental updates of the match sets, i.e. whenever the contents of a RETE node is changed, child nodes are also updated using the difference (inserted or deleted tuples). There are three types of nodes in the RETE net: (i) input nodes serve as the knowledge base of the underlying model, e.g. there is a separate node for each entity or relation type, enumerating the set of instances as tuples; (ii) intermediate nodes perform operations to derive a set of partial matches; finally, (iii) production nodes store the complete match set of a given pattern.

The stochastic simulation of the VoIP network can be performed with graph transformations. In this scenario, we define various graph patterns as the left hand side value of the graph transformation and put model manipulation actions on the right hand side. The simulation can be executed for a given amount of time or for a given amount of transformation steps. After the execution is done, the non-functional properties can be measured with so called probe rules that only check the existence of given subgraphs but do not perform model manipulation.

2.4 Transitive closure preliminaries

The stochastic simulation problem is addressed with graph models and their transformations. To measure the quality of service, various probe rules are used and some of them relies on the transitive closure of the base communication graph of the client nodes. The following paragraphs give formal definitions for the graph related terms to further discuss the running example.

Directed graph A directed graph (digraph) is a pair $G = (V, E)$, where V is a finite set of vertices and $E \subseteq V \times V$ is a set of ordered pairs called arcs or edges. When multiple graphs are present, I will use $V(G)$ and $E(G)$ to represent V and E of the underlying graph G. A digraph $G' = (V', E')$ is a subgraph of $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$.

Path and cycles A sequence $u_0 u_1 \dots u_n (n > 0)$ of vertices in G is a walk from u_0 to u_n if (u_{i-1}, u_i) is in E for each $i \in [1..n]$; the sequence is a path if it is a walk and

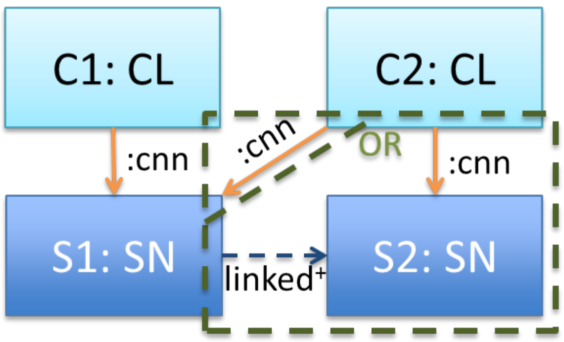


Figure 2.5: *Transitive closure example - pconnected pattern*

$u_i \neq u_j$ whenever $0 \leq i < j \leq n$. We say $u_i u_{i+1} \dots u_{j-1} u_j$ is a subpath of $u_0 u_1 \dots u_n$ for all $0 \leq i < j \leq n$; when $i \neq 0$ or $j \neq n$, we say the subpath is strict. The sequence $u_0 u_1 \dots u_n (n > 0)$ is a cycle if $u_0 = u_n$ and $u_i \neq u_j$ for all $0 \leq i < j \leq n$ such that $(i, j) \neq (0, n)$. An arc on a cycle is called a cyclic arc. G is acyclic if it contains no cycles.

Transitive closure For a graph G, the irreflexive transitive closure G^+ consists of (u, v) pairs of elements (transitive closure relation) for which there is a non-empty path from u to v in E. The definition can be generalised for any binary relation E over a domain D. In case of generic transitive closure, the base relation E is a 'derived edge', defined by an arbitrary two-parameter graph pattern, not restricted to simple graph edges. The current thesis focuses on generic, irreflexive transitive closure, as it is the most general and flexible approach.

Strongly connected component (SCC) A graph is strongly connected if all pairs of its vertices are mutually transitively reachable. An SCC of a graph is a maximal subset of vertices within a graph that is strongly connected. It is easy to see that the SCC of a vertex v is the intersection of the set of ancestors and descendants of the vertex; and thus each graph has a unique decomposition S into disjoint SCCs.

Reduced graph For a graph $G = (V, E)$, the SCCs form the reduced graph $G_c = (S, E_c)$, where two SCCs are connected if any of their vertices are connected: $E_c = \{(s_i, s_j) | s_i, s_j \in S \wedge \exists u \in s_i, v \in s_j : (u, v) \in E\}$. It follows from the definitions that a reduced graph is always acyclic.

Topological sorting The topological sorting of a directed graph is an ordering of its vertices such that, for every edge (u, v), u comes before v in the ordering. A topological sorting is possible if and only if the graph contains no cycles, that is, if it is a directed acyclic graph (DAG). All DAG graphs have at least one topological sorting and it can be computed in linear time.

The example in Figure 2.5 demonstrates transitive closure features in graph pattern matching. A transitive closure over the overlay network of super nodes is specified by the pattern *pconnected* that defines the relationship between any two client nodes $C1$, $C2$ which are (i) either sharing a common super node to which they are both directly connected by *cnn* edges, or (ii) their *cnn* connection is indirect in the sense that their super nodes $S1$, $S2$ are reachable from each other through a transitive *linked*+ relationship. The latter is the generic transitive closure of a derived edge defined by the binary linked pattern (Figure 2.3).

To align with the performance characteristics of incremental pattern matching, it naturally comes to mind to use incremental transitive closure algorithms. Great effort has been put into the research of transitive closure algorithms and there are numerous static and dynamic (incremental) algorithms available in the literature. Chapter 3 gives an in-depth overview about some of these algorithms and their application inside the RETE net for pattern matching purposes.

Tarjan's algorithm

Tarjan's algorithm [35] is used to compute the strongly connected component partitioning of a graph. The algorithm is based on depth-first search which starts from an arbitrary node. During the traversal the nodes are placed on a stack in the order they are visited. When the traversal returns from a subtree, the nodes are taken from the stack until the root node of the current strongly connected component is found. Here, root node simply means the first node from the SCC that was encountered during the traversal. If a node is a root node, then it and all of the other nodes that were taken off from the stack before it form an SCC.

Section A.1 presents an iterative implementation of Tarjan's algorithm in Java.

2.5 The Union-find data structure

The union-find (or disjoint-set) data structure [37] keeps track of a set of elements partitioned into a number of disjoint subsets. There are usually two operations defined on a union-find structure:

- union: merge two subsets into a single one
- find: determine which subset a particular element is in.

The union-find data structure can be extended to handle the delete operation too: remove a given element from a set (the container set can be found with the find operation). In order to define the operations more precisely, some way of representing the sets is needed. A common approach is to select a fixed element of each set, called as the representative (which will represent the whole set).

There are several representations of a union-find data structure; for example linked lists, forests etc. This section gives an overview about possible implementations for the operations when the structure is represented with forests. In this case each set will be represented by a tree with the representative node being the root of it.

MakeSet Creates a new set containing the single element x and initializing its parent as itself.

Listing 2.1: Union-Find MakeSet operation

```
function MakeSet(x) {
     x.parent := x
}
```

Find Finds the representative element of the set which contains the given element x.

Listing 2.2: Union-Find Find operation

```
function Find(x) {
     if x.parent == x
        return x
     else
        return Find(x.parent)
}
```

Union First it finds the representative elements for each passed element and the merge operation is performed on the representatives.

Listing 2.3: Union-Find Union operation

```
function Union(x, y) {
     xRoot := Find(x)
     yRoot := Find(y)
     xRoot.parent := yRoot
}
```

Delete The delete operation is done by first finding the representative node of the passed element and then remove the element from the container set. If the representative node is deleted, then it is required to promote a node from the set to be the new representative.

Listing 2.4: Union-Find Delete operation

```
function Delete(x) {
     xRoot := Find(x)
     Delete x from the container set whose representative node is xRoot
}
```

There are several enhancements for the union-find data structure to improve performance:

- Path compression: it is a way of flattening the structure of the tree whenever Find is called. The idea is to attach each node directly to the representative node of the tree during the traversal. As a result, the tree is much flatter which speeds up future Find calls not only on these elements but on those referencing them.
- Union by rank: the idea is to always attach the smaller tree to the root of the larger tree when performing Union operations. Since it is the depth of the tree that affects the running time, it is advantageous to increase the size of the tree only if necessary. Always attaching to the larger tree will only increase the depth when the two trees being merged are of equal depth. Note that, here the term rank is used, as depth is no longer equal to it if path compression is also used.

These two techniques will result that the amortized time per operation is only $O(\alpha(n))$, where n is the number of elements in the structure and $\alpha(n)$ is the inverse Ackermann function [12].

2.6 EMF-IncQuery Base

EMF-IncQuery Base [3] is an additional component over the EMF-IncQuery framework and it aims to provide several useful features for querying EMF models:

- Inverse navigation along EReferences
- Finding all model elements by attribute value/type (i.e. inverse navigation along EAttributes)
- Computing transitive reachability along given reference types (i.e. transitive closure of an EMF model)
- Getting all the (direct) instances of a given EClass [1].

The point of EMF-IncQuery Base is to provide all of these in an incremental way, which means that once the query evaluator is attached to an EMF model, as long as it stays attached, the query results can be retrieved instantly (as the query result cache is automatically updated). The first version of the component was developed by me and it is now heavily used in the EMF-IncQuery framework. It is a lightweight, small Java library that can be integrated easily to any EMF-based tool. The IncSCC (see Section 3.4) transitive closure algorithm was integrated in order to provide reachability computation.

Some of the functionality can be found in some Ecore utility classes (for example *ECross-ReferenceAdapter*) but these standard implementations are non-incremental, and are thus do not scale well in scenarios where high query evaluation performance is necessary (such as on-the-fly well-formedness validation or live view maintenance).

[1]EAttribute, EReference and EClass are part of the Ecore metamodel.

Chapter 3

Transitive closure and its application

Knowledge of transitive closure is useful in several scenarios; graph models in telecommunication, logistics or social networks. Motivating examples from the context of MDSD include checking of type hierarchy of classes, containment hierarchy of objects and stochastic simulation with graph transformation. However, in practice, the base graph is subject to change: for example in the case of telecommunication networks, new links are added or nodes become isolated because of link failures. In the context of large evolving graphs, the efficient computation of the new transitive closure relation after an update can be critical to performance.

Types of transitive closure algorithms There are various transitive closure algorithms in the literature which are characterized in the thesis with two properties: (i) how the transitive closure relation is computed after the graph is modified and (ii) whether the algorithm is able to handle cyclic closures correctly:

- Static transitive closure algorithms: these algorithms recompute the whole transitive closure relation from stretch after changes in the base graph occur. The Floyd-Warshall [20] algorithm is a popular static algorithm, which is based on the Floyd - all pairs shortest path graph algorithm.

- Dynamic or incremental algorithms: these algorithms maintain a cache of the transitive closure relation and are able to derive the new relation much faster than the static ones after graph modification. When an edge/vertex is inserted or deleted it is enough to derive only the changes in the relation which results much better performance characteristics with a slightly increased memory overhead. There are several incremental algorithms to compute the transitive closure; these range from simple approaches like DRED (Delete and REDerive) and Counting [23] to more complex ones like King [29] and IncSCC which was presented in our paper [14].

Note that, some of the investigated algorithms are not able to handle cyclic closures correctly, in other words, they give wrong reachability information if (directed) cycles are present in the graph. Throughout the discussion, it will be explicitly stated whether the given algorithm can be applied only on DAGs.

The chapter first gives a brief overview about the base requirements against a transitive closure algorithm (Section 3.2) and presents a brief survey on the literature of transitive closure algorithms in Section 3.3. The implemented algorithms are also discussed in-depth. Section 3.4 introduces an adaptation of the algorithm presented in [31], which is based on the incremental maintenance of strongly connected components. The adaptation of the algorithm was presented in our paper [14]. Both the $VIATRA2$ [8] VTCL language and the query language of EMF-IncQuery were extended with transitive closure as a language element and as both of them use RETE for incremental graph pattern matching Section 3.5 describes the integration of IncSCC into the RETE net.

3.1 Notations used in this section

In the literature one can read mixed definitions about the terms incremental and fully dynamic algorithms. An algorithm or a problem is called fully dynamic if both edge insertions and deletions are allowed, and it is called partially dynamic if either edge insertions or edge deletions (but not both) are allowed. In the case of edge insertions (respectively deletions), the partially dynamic algorithm or problem is called incremental (respectively decremental). However, incremental on the other hand, means that the changes are derived upon modifications without the recomputation of the whole transitive closure relation. The thesis does not distinguish between the fully dynamic and incremental properties and this chapter describes algorithms that are able to handle edge deletion and insertion too.

The discussion is based on a directed graph $G = (V, E)$, where V denotes the set of vertices and E the set of edges in the graph, while $|V|$ and $|E|$ stand for the arity of the set of vertices and the set of edges respectively. Consider a vertex $v \in V$ and let $\circ v$ denote the vertices from which v is transitively reachable, and $v\circ$ denotes the vertices which are reachable from v. The term complexity is used to describe the performance (time) complexity of the algorithms.

3.2 Transitive closure operations

Any program module computing the transitive closure G^+ of a graph G is required to expose a subroutine $Construct(G)$ that builds a data structure for storing the result and possibly auxiliary information as well. Afterwards, the following reachability queries can be issued: $Query(Src, Trg)$ returns whether Trg is reachable from Src; $Query(Src, ?)$ returns all targets reachable from Src (also denoted as $Src\circ$), while $Query(?, Trg)$ returns all sources from where Trg can be reached ($\circ Trg$); finally $Query(?, ?)$ enumerates the whole G^+.

In case of incremental computation, the following additional subroutines have to be exposed: $Insert(Src, Trg)$ updates the data structures after the insertion of the (Src, Trg) edge to reflect the change, while $Delete(Src, Trg)$ analogously maintains the data structures upon an edge deletion. To support further incremental processing, both of these methods return the delta of G^+, i.e. the set of source-target pairs that became (un)reachable due to the change.

3.3 Survey of transitive closure algorithms

While there are several classical algorithms (depth- and breadth-first search, etc.) for computing transitive reachability, efficient incremental maintenance of transitive closure is a more challenging task. As transitive closure can be defined as a recursive Datalog [1] [25] query, incremental Datalog view maintenance algorithms such as Counting and DRED [23] can be applied as a generic solution. There is also a wide variety [18] of algorithms that are specifically tailored for the fully dynamic transitive reachability problem. Some of these algorithms provide additional information (shortest path, transitive reduction), others may be randomized algorithms (typically with one-sided error); the majority focuses on worst-case characteristics in case of dense graphs. The spectrum of solutions offers various trade-offs between the cost of operations specified earlier in this section. Even if the original graph has a moderate amount of edges (sparse graph), the size of the transitive closure relation can easily be a quadratic function of the number of vertices, raising the relative cost of maintenance. A key observation, however, is that in many typical cases vertices will form large SCCs. This is exploited in a family of algorithms [31][22] and one of the main contributions of the thesis - namely the IncSCC algorithm [14] - is also based on this idea. The algorithm maintains (a) the set of SCCs using a dynamic algorithm, and also (b) the transitive reachability relationship between SCCs. Choosing such an algorithm is justified by simplicity of implementation, the sparse property of typical graph models and the practical observation that large SCCs tend to form.

3.3.1 Static transitive closure algorithms

Depth first search The depth-(breadth-) first search algorithms can be used to compute reachability information from a given vertex - by starting a single traversal. $Query(Src, Trg)$ is true if Trg is reached during the traversal started from Src, otherwise false. Naturally, $Query(Src, ?)$ can also be answered; it will consist of the vertices that are visited during the traversal from Src. In the case of $Query(?, Trg)$ the only difference is to reverse the direction of edges and start the traversal from Trg. The complexity of one traversal is $O(|V| + |E|)$ and the whole recomputation of the transitive closure relation is $O(|V|(|V| + |E|))$. The algorithm is able to handle cyclic closures correctly.

Floyd-Warshall algorithm [20] The Floyd-Warshall algorithm is based on the Floyd all pairs shortest paths algorithm. It is implemented as a dynamic programming problem and runs with $O(|V|^3)$ complexity (whole relation). The algorithm compares all possible paths through the graph between each pair of vertices and continuously increases the interval of intermediate vertices that can be used to form the current shortest path. Consider a matrix P with dimensions $|V| \times |V|$, where $P[i][j]$ defines whether vertex j is reachable from vertex i. The crux of the algorithm can be described with the pseudocode in Listing 3.1.

[1] Datalog is a truly declarative logic programming language that syntactically is a subset of Prolog.

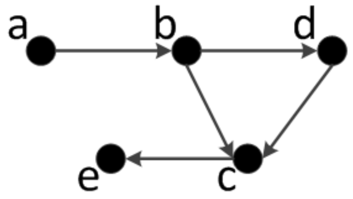


Figure 3.1: *Example graph for topological sorting*

Listing 3.1: Floyd-Warshall algorithm

```
for each k in [0, n) {
        for each i in [0, n) {
                for each j in [0, n) {
                        P[i][j] = P[i][j] or (P[i][k] and P[k][j]);
                }
        }
}
```

Vertex i is reachable from vertex j, if (i) there is an edge between the two vertices or (ii) there exists an intermediate vertex from the interval $1 \dots k$, which is reachable from vertex i and vertex j is reachable from it. G^+ is stored explicitly in the matrix P and the *Query* operations are answered based on the elements of the matrix. The algorithm is able to handle cyclic closures correctly.

Transitive closure based on the topological sorting This solution can be applied on DAG graphs, where the topological sorting is defined. The algorithm iterates through the reversed topological sorting. Consider an iteration in which the current vertex is v; the algorithm inserts the Cartesian product of the set of the source vertices of v (the ones which have an outgoing edge that ends in n) and the $(v \bigcup v\circ)$ set (this is known from the previous iteration) into the transitive closure relation. The algorithm works with a worst case complexity of $O(|V|^3)$, from which $O(|V|)$ is for computing the topological sorting.

The only topological sorting of the graph shown in Figure 3.1 is (a, b, d, c, e) and the tuples are inserted into the relation in the following order: (1) the source vertices of e is $\{c\}$, the reachable one is itself $\{e\}$, the Cartesian product is $\{c\} \times \{e\} = \{(c, e)\}$ (2) $\{b, d\} \times \{c, e\} = \{(b, c), (b, e), (d, c), (d, e)\}$ (3) $\{b\} \times \{c, d, e\} = \{(b, c), (b, d), (b, e)\}$, only (b, d) is a new tuple (4) $\{a\} \times \{b, c, d, e\} = \{(a, b), (a, c), (a, d), (a, e)\}$ (5) processing vertex a produces no new tuples.

As topological sorting is only defined for DAGs, this algorithm cannot handle cyclic closures.

3.3.2 Incremental transitive closure algorithms

Counting

In [23] an incremental algorithm is presented based on Datalog queries to define and maintain non-recursive views in databases. This implies that the algorithm can only be applied on DAG graphs. The Counting algorithm works by storing the number of alternative derivations of each tuple in the transitive closure relation. This number is called *count(t)* and a

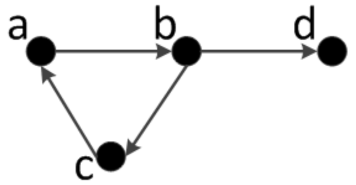


Figure 3.2: *Example graph for the Counting algorithm*

tuple t exists in the relation if and only if the *count(t)* value is greater than 0. $Construct(G)$ initializes the relation with the non-incremental topological sorting based algorithm.

Consider a relation G and let $\Delta(G)$ denote the changes made on G, while G^v refers to the relation after incorporating the changes on G. With this notation we define two rules for computing the transitive closure; (r_1) $tc(u,v) \vdash e(u,v)$ meaning that the tc (transitive closure) relation consists of the tuples defined by the edges of a graph and (r_2) $tc(u,v) \vdash e(u,w)\&tc(w,v)$ meaning that v is reachable from u, if there is an (u,w) edge and v is reachable from w.

With every rule r of the form $(r) : p \vdash s_1\& \ldots \&s_n$ n delta rules $(\Delta_i(r)), 1 \leq i \leq n$ are defined, and predicate $\Delta(p)$ is as follows:
$(\Delta_i(r)) : \Delta(p) \vdash s_1^v\& \ldots \&s_{i-1}^v\&\Delta(s_i)\&s_{i+1}\& \ldots \&s_n$.

Based on this definition we can derive 3 rules for the Counting algorithm. These rules are used to compute the delta of G^+ and the tuples are stored explicitly:

- $(\Delta_1(r_1))$: $\Delta(tc(u,v)) \vdash \Delta(e(u,v))$: the newly inserted/deleted edges
- $(\Delta_1(r_2))$: $\Delta(tc(u,v)) \vdash \Delta(e(u,w))\&tc(w,v)$: paths starting with the newly modified edges
- $(\Delta_2(r_2))$: $\Delta(tc(u,v)) \vdash e^v(u,w)\&\Delta(tc(w,v))$: paths after incorporating the edge insertion/deletion. This rule is applied repeatedly until a local fixpoint is reached.

The Counting algorithm is not able to deal with cyclic closures because of the 'circular deception' problem, that is, a cycle includes at least two count which will result false reachability information after edge deletion. Consider the graph on Figure 3.2, here $count(b,d) = 2$ because of the cycle. After deleting edge (b,d), the derivations of the rules given above will result only -1 count for the tuple (b,d) which results that vertex d remains reachable from every vertex in the cycle.

The transitive closure relation is implemented with two maps; one to index the tuples forward and one for the reversed direction. This way, the *Query* operations are answered with simple look-ups in the map data structure.

The Counting algorithm can be modified to count the number of different but not necessarily independent paths between any two vertices. In this case the transitive closure maintenance is quite simple; when an edge (u,v) is inserted (deleted) it simply computes the Cartesian product of the vertices from which u is reachable (set S_1) and the ones that are reachable from v (set S_2). Suppose $x \in S_1, y \in S_2$, then $count((x,y)) = count((x,u)) \cdot count((v,y))$.

DRED - Delete and REDerive

A view-maintenance algorithm which works for recursive views is also presented in [23]. This simple algorithm explicitly stores G^+ and $Construct(G)$ uses the topological sorting based algorithm like Counting.

The name comes from the handling mechanism of edge deletion:

[$1-del$] **Delete** a superset of derived tuples that need to be deleted. Suppose an (u,v) edge is deleted and the overestimate is computed by a naive Cartesian product method which marks the tuples $(\circ u \bigcup u) \times (v \bigcup v\circ)$ to be deleted.

[$2-del$] **REDerive** those tuples that have alternative derivations after incorporating the changes made on the graph. This step is performed with similar rules that are used in the Counting algorithm resulting to compute a local fixpoint.

[$1-ins$] Insert new tuples after edge insertion using the naive Descartes product method (presented in the first step), but in this case those tuples will be inserted back into the relation.

The first two steps are used to handle edge deletions, while the third step handles the edge insertion.

This algorithm is able to handle cyclic closures too and the *Query* operations are directly answered based on the map-backed relation of G^+.

King's algorithm

In [29] algorithms are presented to compute approximate and exact shortest paths and transitive closure in graphs. For transitive closure, it maintains $k = \lceil log_2(|V|) \rceil$ forests $F_1, F_2, \ldots, F_k$ where each F_i contains a pair of breadth-first search trees In_i^v and Out_i^v of depth 2 for each vertex $v \in V$. Forest F_i is built from the graph which contains an (u,v) edge if there is a directed path from u to v in the original graph with length greater than 2^{i-1} and smaller or equal to 2^i.

During the thesis work I have implemented an algorithm which is highly influenced by King's one. The overall idea is kept in my implementation but different data structures are used to maintain the transitive closure relation. Let $A \overrightarrow{\bowtie} B$ denote the natural join of the sets A and B (both contain pairs of vertices), but with the restriction that a pair (u,v) is only present in the resulting relation if $\exists (u,w) \in A \wedge (w,v) \in B$ in this direction.

The algorithm stores G^+ explicitly in levels:

- E_0: the edges of the graph, the paths with a length of exactly 1.
- E_1: the tuples that can be derived with the evaluation of $E(0) \overrightarrow{\bowtie} E(0)$, the paths with a length exactly 2.
- E_2: the tuples that can be derived with the evaluation of $(E(1) \overrightarrow{\bowtie} E(0)) \bigcup (E(1) \overrightarrow{\bowtie} E(1))$, the paths with a length of 3 or 4.

- ...
- E_i: the tuples that can be derived with the evaluation of $\bigcup_{j=0}^{i-1}(E_{i-1}\overrightarrow{\bowtie}E_j)$, the paths with a length from $2^{i-1}+1$ to 2^i.

When the graph changes, the levels can be maintained incrementally based on the changes in the lower levels. On each level, each tuple is assigned a count value, that is, the different derivations of the tuple on the given level. However, the significant level of a tuple t is the lowest number i, such that t appears on E_i. Each tuple in the transitive closure relation is assigned exactly one significant level.

Significant level movement The algorithm maintains the transitive closure relation based on the tuples' significant level. There can be two kinds of change in the significant level:

- **HOP_UP** (the significant level is incremented): when a tuple's derivation disappears on a lower level then the significant level moves up. It is also possible that there are no more levels, where the tuple has derivation. In this case the tuple will be deleted from the relation (the tuple's significant level 'hops up to infinity').
- **HOP_DOWN** (the significant level is decremented): when a tuple's derivation appears on a lower level then the significant level moves down. It is also possible that the tuple hasn't got any derivation yet, and this is the first path (the tuple's significant level 'hops down from infinity').

Derivation of new tuples

In the case of a tuple (u, v) which hops up from level i to level j:

Derivation	Action	Step required if j $= \infty$?
$(u,v)\overrightarrow{\bowtie}(v,w)\|_{k\in[0,i]}$	(u,w) gets -1 count on level $i+1$	yes
$(u,v)\overrightarrow{\bowtie}(v,w)\|_{k\in[0,j]}$	(u,w) gets $+1$ count on level $j+1$	no
$(w,u)\|_{k\in[i,j-1]}\overrightarrow{\bowtie}(u,v)$	(w,v) gets -1 count on level $k+1$	yes

Table 3.1: *Derivation of new tuples in the King-like algorithm for HOP_UP*

In the case of a tuple (u, v) which hops down from level j to level i:

Derivation	Action	Step required if j $= \infty$?
$(u,v)\overrightarrow{\bowtie}(v,w)\|_{k\in[0,i]}$	(u,w) gets $+1$ count on level $i+1$	yes
$(u,v)\overrightarrow{\bowtie}(v,w)\|_{k\in[0,j]}$	(u,w) gets -1 count on level $j+1$	no
$(w,u)\|_{k\in[i,j-1]}\overrightarrow{\bowtie}(u,v)$	(w,v) gets $+1$ count on level $k+1$	yes

Table 3.2: *Derivation of new tuples in the King-like algorithm for HOP_DOWN*

The update mechanism

When an edge is inserted or deleted the level number will be 0, as it is an edge in the base graph. The *Insert* and *Delete* methods derive the new tuples that will be inserted / deleted. The *Insert* method adds +1 count for the (Src, Trg) tuple on the given level number.

Listing 3.2: King-like algorithm update method after edge insertion

```
function Insert(Src, Trg, levelNumber = 0) {
        //it will be infinity if such tuple does not exist
        oldSL = (Src, Trg).significantLevel
        insert (Src, Trg) on level levelNumber
        newSL = (Src, Trg).significantLevel
        if (newSL < oldSL) {
                handle HOP_DOWN based on the table above
        }
}
```

The *Delete* method adds -1 count for the (Src, Trg) tuple on the given level number.

Listing 3.3: King-like algorithm update method after edge deletion

```
function Delete(Src, Trg, levelNumber = 0) {
        oldSL = (Src, Trg).significantLevel
        remove (Src, Trg) on level levelNumber
        //it will be infinity if the tuple has no derivations anymore
        newSL = (Src, Trg).significantLevel
        if (newSL > oldSL) {
                handle HOP_UP based on the table above
        }
}
```

The initialization is done with the static formula $E_i = \bigcup_{j=0}^{i-1}(E_{i-1}^{*} \overrightarrow{\bowtie} E_j^{*})$, where the $*$ is a restriction to the tuples whose significant level is on the given level. The *Query* operations are answered based on the explicitly stored G^{+} relation.

'Jacob's ladder' problem and its solution

The pseudocode given in Listing 3.3 introduces an error with 'similar behavior like the high voltage device, Jacob's ladder'. Consider a tuple t_1 that we delete with no more derivations, that is, its significant level will be increased. The problem occurs if there exists a tuple t_2 such that t_1 and t_2 mutually reinforces each other's derivation. The deletion will result that t_1 will appear on an upper level in a new derivation. When t_1 appears on an upper level, t_2 will also hop up from the lower level to the upper level. With this mutual reinforcement t_1 and t_2 will keep hopping up continuously until the level number overflows. To overcome the presented problem, a tuple is only added to the transitive closure relation if the $levelNumber$ is smaller or equal to $\lceil log_2(|V|) \rceil$.

3.4 IncSCC

IncSCC is an algorithm to maintain strongly connected components incrementally which is based on the ideas presented in [31] and [22]. A transitive closure algorithm based on this idea is a key contribution of the thesis and this solution was applied in the pattern language of the VIATRA2 and EMF-IncQuery frameworks.

The crux of the algorithm, from Poutré and Leeuwen's paper [31], is to reduce update time and memory usage by eliminating unnecessary reachability information, namely, that each vertex is reachable from every other vertex within the same SCC. Thus, the two concerns of the algorithm are maintaining (i) a decomposition S of the graph into SCCs, and (ii) transitive reachability within the condensed graph. The latter is a simpler problem with several efficient solutions, as the condensed graph is acyclic; our implementation relies on the Counting algorithm which simply keeps track of the number of derivations of each transitive reachability pair. The main differences opposed to the original algorithm are; (i) the transitive closure relation is not stored explicitly in our solution (ii) they operate with matrices of graphs opposed to our solution which is based on efficient Union-Find data structure implementation (see Section 2.5) and (iii) the solution is able to send notifications about the changes in the transitive closure relation.

Implementing Construct(G) The SCC partitioning of the initial graph are computed with Tarjan's algorithm [35] (see Section 2.4 for more details). Afterwards, the condensed graph is constructed, and the Counting algorithm is initialized to provide reachability information between SCCs.

Implementing Query$()$ *operations* As the most significant deviation from [31], the transitive closure relation G^+ is not stored explicitly in our IncSCC solution to reduce the memory footprint. However, reachability in graph $G(V, E)$ can be reconstructed from the partitioning S of SCCs and the reachability relation G_c^+ of condensed graph $G_c(S, E_c)$, since for $s_1, s_2 \in S, u \in s_1, v \in s_2 : (s1, s2) \in G_c^+$ if and only if $(u, v) \in G^+$. Therefore when receiving a reachability query, the vertices in question are mapped to SCCs, where reachability information in the condensed graph is provided by the Counting algorithm. Vertices enumerated in the answer are obtained by tracing back the SCCs to vertices. This is solved by using the Union-Find data structure for the SCC partitioning (see Section 2.5 for details).

Implementing Insert(Src, Trg) First a look-up in S maps the vertices to SCCs. Afterwards, there are three possible cases to distinguish.

1. If Src and Trg are in different SCCs, but no cycle will appear with the edge insertion, then it is simply handled by the Counting algorithm. Computation of $\Delta(G^+)$ is done by the formula $(\circ Src \bigcup Src) \times (Trg \bigcup Trg\circ)$ having representative nodes of SCCs on both sides of a tuple. The notification is handled by Counting itself, and tuples which have already existed with an alternative derivation are omitted. Note that, these sources and targets are SCCs of the graph so it needs to be traced back to the actual vertices when notifying observers of the transitive closure relation.

   This case can be seen on Figure 3.3 where an edge (b, d) is inserted into the graph (the edge being inserted is marked with red and the SCCs of the graph are marked with blue rectangles). The only notification issued is the tuple $\{b, d\}$. Note that, if multiple edges are present between two vertices, then notifications are issued only after the first appearance of such an edge.

2. If Src and Trg are in different SCCs and the inserted edge results a cycle in the condensed graph then the cycle is collapsed into a single SCC. The following steps are required:

   (a) Compute the intersection of the predecessor representative nodes of Src and the successor representative nodes of Trg. These SCCs form the cycle.

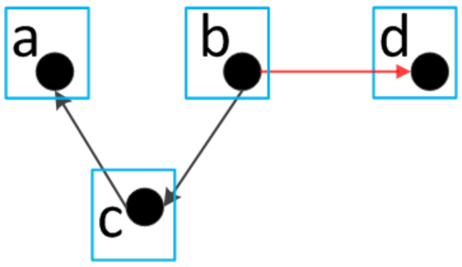


Figure 3.3: *IncSCC maintenance - two different SCCs*

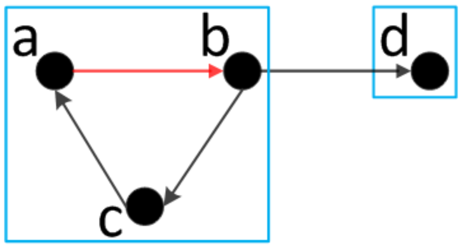


Figure 3.4: *IncSCC maintenance - SCC collapsing after edge insertion*

(b) Delete the edges from the reduced graph going in and out from these SCCs and after that the SCCs too (now being isolated vertices in the condensed graph).

(c) Merge the SCCs together by calling the Union operation on the Union-Find structure.

(d) Add the newly collapsed SCC as a single vertex to the condensed graph.

(e) Insert the appropriate incoming and outgoing edges of this SCC. These edges are derived with the use of the base graph.

Computation of $\Delta(G^+)$ is done with a similar formula like in the previous case. During the process the algorithm omits the notifications about the reachability tuples which have already existed (and been notified about); (i) when the current target was already reachable from the current source and (ii) when source and target are the same representative node. Again, tracing back of SCCs is needed when issuing notifications on the relation.

This is presented on Figure 3.4 where the edge (a, b) is inserted. This results to collapse the one-sized SCCs of the vertices a, b and c into one bigger SCC. According to the notification mechanism described before, the tuples $\{\{b, a\}, \{b, c\}, \{b, d\}, \{c, a\}\}$ are omitted.

3. If *Src* and *Trg* are in the same SCC there is no required action. This case can be seen on Figure 3.5 where an edge (a, b) is inserted again into the graph. It would also hold for self-cycles within the SCC, however, a first appearance of a self-reachability tuple would be subject of notification.

Implementing Delete(Src, Trg) The algorithm first performs a look-up in S to map the vertices to SCCs; afterwards, we once again distinguish three possible cases.

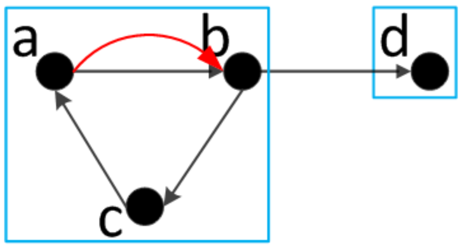


Figure 3.5: *IncSCC maintenance - inside an SCC*

1. If Src and Trg are in the same SCC but Trg remains reachable from Src after the edge deletion (as confirmed by a breadth-first search in the subgraph associated to the actual SCC), no further actions are required.

   This is the case when an edge a, b is deleted from the graph presented on Figure 3.5 - however, in these scenarios, the edge marked with red is the one being deleted. No reachability tuple will be ceased as vertex b still remains reachable from vertex a.

2. If Src and Trg are in the same SCC but Trg is no longer reachable from Src after the edge deletion, then the SCC is broken up (using Tarjan's algorithm) into smaller SCCs, because it is not strongly connected anymore. The following steps are required:

   (a) Store the subgraph referring to the SCC being removed and compute the new SCC partitioning with Tarjan's algorithm.

   (b) Remove the incoming and outgoing edges of the SCC and after that the SCC too.

   (c) Remove the corresponding set from the Union-Find structure.

   (d) Insert the new SCCs into the condensed graph and the appropriate incoming and outgoing edges too.

   $\Delta(G^{+}) = (\circ Src \bigcup Src) \times (Trg \bigcup Trg\circ)$, but omitting those tuples where (i) source is still reachable from target (known from Counting) and (ii) source is the same as target. These tuples are deleted from the transitive closure relation.

   Figure 3.6 presents this case, where the SCC of vertices a, b and c will be broken up into one-sized SCCs after the deletion of the (a, b) edge. No notification will be issued about the dismissal of the tuple $\{b, c\}$ for example, as it is known from Counting, that it still holds, even though the two vertices were in the same bigger SCC.

3. If Src and Trg are in different SCCs, then the edge is deleted from the condensed graph, which is in turn is handled by the Counting Algorithm. $\Delta(G^{+})$ is computed similarly as in the previous case, but only those tuples must be omitted where *source* is still reachable from *target*.

   An example of this case is the one presented on Figure 3.3 where the edge (b, d) is deleted. No changes in the SCC decomposition will occur, however, the tuple $\{b, d\}$ is ceased.

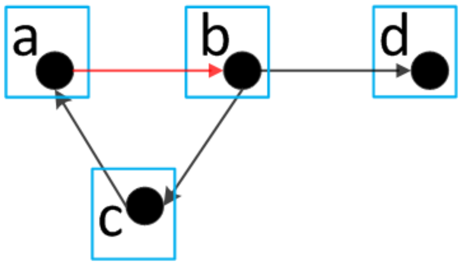


Figure 3.6: *IncSCC maintenance - breaking up an SCC after edge deletion*

Source code for the *Insert* and *Delete* operations can be found in Section A.2. Detailed performance comparison for the described algorithms can be found in Chapter 6.

3.5 Integrating the IncSCC algorithm into the RETE net

This section describes how a transitive closure algorithm - which provides notification handling too - can be integrated into RETE. Transitive closure is represented by a RETE node, like any other pattern. Generic transitive closure is achieved by attaching such a RETE node to a parent node that matches a graph edge or an arbitrary binary graph pattern (derived edge).

Figure 3.7 shows the transitive closure node in the RETE network. It is an intermediate node which receives updates from a binary graph pattern (here denoted as binary relation E) and forms a two-way interface between RETE and a transitive closure maintenance algorithm. Whenever the RETE node for E^+ receives an insertion / deletion update from its parent node E, the $Insert()/Delete()$ subroutine is invoked. The subroutine computes the necessary updates to E^+, and returns these delta pairs, which will then be propagated along the outgoing edge(s) of the RETE node. Queries are invoked when initializing the child nodes, and later as a quick look-up to speed up join operations against the node contents. Alternatively, transitive closure could have been expressed as a recursive graph pattern. This solution was rejected, as RETE, having first-order semantics without fixpoint operators, might incorrectly yield a (still transitive) superset of the transitive closure: in graph models containing cycles, obsolete reachabilities could cyclically justify each other after their original justification was deleted.

Case study example (transitive closure RETE node) Here the behavior of the RETE node is demonstrated which computes the transitive closure E^+ of the binary graph pattern E, e.g. *linked+* for the overlay network where *linked* is defined between super nodes. Initially, as seen in Figure 3.7 (a), the parent node E stores *linked*, i.e. the binary relation $\{(A, B), (B, C)\}$. Its child node E^+ contains the set of reachable pairs: $\{\{A, B\}, \{A, C\}, \{B, C\}\}$. Figure 3.7 (b) shows the insertion of edge (C, A) into E. RETE propagates this update from the E to E^+, where the operation $Insert(C, A)$ is invoked to adjust the transitive closure relation to $\{\{A, B\}, \{A, C\}, \{B, A\}, \{B, C\}, \{C, A\}, \{C, B\}\}$, i.e. the whole graph becomes strongly connected. The computed difference (delta) is the insertion of $\{\{B, A\}, \{C, A\}, \{C, B\}\}$ into E^+, which is propagated in the RETE net-

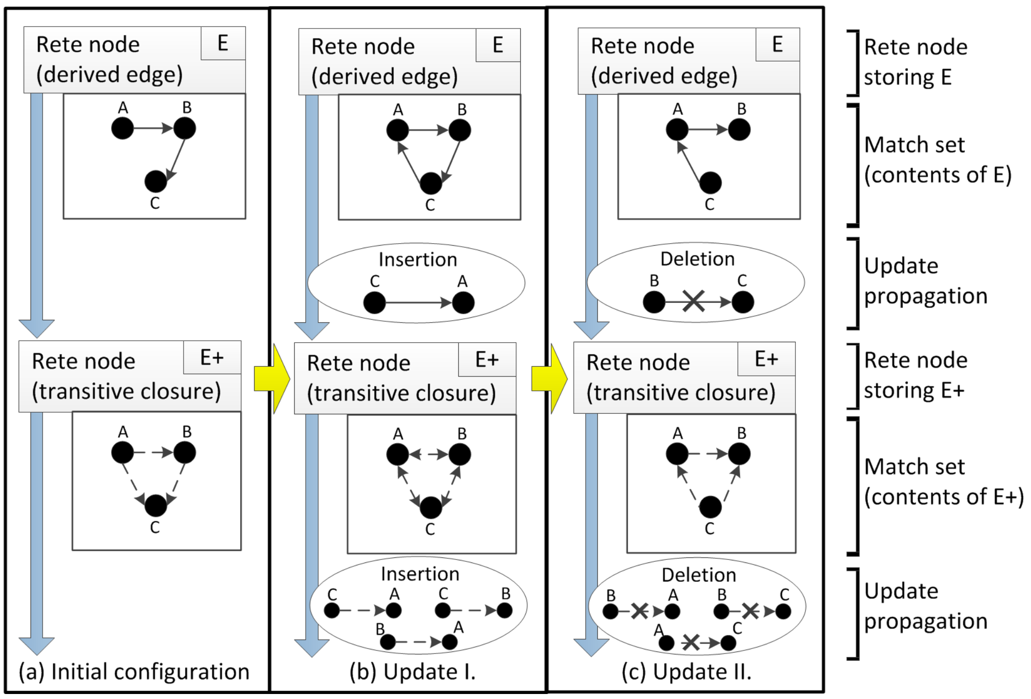


Figure 3.7: *Transitive closure node inside RETE*

work to child nodes of E^+. Finally, Figure 3.7 (c) shows an edge deletion. E^+ is notified of the deletion of (B, C) from E, and invokes $Delete(B, C)$. Thus E^+ becomes $\{\{A, B\}, \{C, A\}, \{C, B\}\}$, and the delta is the deletion of $\{\{A, C\}, \{B, A\}, \{B, C\}\}$.

3.5.1 Integration with VIATRA2

The main objective of the *VIATRA2* [8] (VIsual Automated model TRAnsformations) framework is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. The stochastic simulation case study was carried out with the *VIATRA2* framework. In its core engine it uses the RETE net for incremental graph pattern matching.

The VIATRA Textual Command Language (VTCL)

From the VIATRA documentation: VIATRA2 has its own programming language called VTCL. Transformations written in VTCL are stored in .vtcl files, and each of them describes a so-called machine of the GTASM (Graph Transformation Abstract State Machine) formalism. The definition of the control flow of the transformation is similar to the mathematical formalism of Abstract State Machines (ASM), and looks just like a conventional programming language (with a syntax somewhat resembling C); this is extended by the declarative features of graph transformations. For more information on the exact syntax and language constructs please see the VIATRA2 documentation ([8]).

Running example aspects

To support transitive closure as a language element a RETE node was implemented which is backed by the IncSCC algorithm. On the frontend this appears as an annotation to be used for the computation of the transitive closure of a given pattern. The following example is from the stochastic simulation case study. Here the *linked* pattern is defined with the VIATRA2 VTCL language and *transitiveClosureOfLinked* is a pattern which is customized with the annotation to represent the transitive closure of the *linked* pattern (set with the *ofPattern* attribute).

Listing 3.4: Transitive closure support in the VIATRA2 VTCL language

```
shareable pattern linked(S1,S2) = {
    SN(S1);
    SN(S2);
    SN.link(E1,S1,S2);
} or {
    SN(S1);
    SN(S2);
    SN.link(E2,S2,S1);
}

@Incremental(reinterpret=transitiveClosure, ofPattern=linked)
pattern transitiveClosureOfLinked(S1, S2) = {}
```

3.5.2 Integration with EMF-IncQuery

The EMF-IncQuery framework also uses a RETE implementation for pattern matching and it has an XText [10] based pattern language to define queries. It is equipped with similar constructs like the VTCL of VIATRA2 but it does not deal with graph transformations. In this case, the *transitiveClosureOfLinked* pattern is simplified as the + operator is introduced for transitive closure computation.

Listing 3.5: Transitive closure support in the pattern language of EMF-IncQuery

```
pattern linked(S1 : SN, S2 : SN) = {
    SN.link(S1, S2);
} or {
    SN.link(S2, S1);
}

pattern transitiveClosureOfLinked(S1 : SN, S2 : SN) = {
    find linked+(S1, S2);
}
```

Chapter 4

Event-driven rule execution engine

Model transformations are a core technology in the context of MDSD. It provides the basic fundamentals to propagate information between the various models during the product lifecycle. The primary use cases for model transformations are mapping between models but they are also widely used for code generation, simulations or executing validation constraints. Model transformations can be implemented in various ways [32]:

- Simply as a conventional program written in a programming language such as Java or C++
- With the use of a dedicated model transformation framework such as the VIATRA2 model transformation framework.
- Combining native tools for the different parts of the workflow resulting a hybrid transformation engine.

The target modeling framework of the thesis is EMF and although EMF-IncQuery incorporates an efficient graph pattern matching engine to query these models, it lacks support for executing transformation rules. One of the main contributions of the thesis is an extension to EMF-IncQuery which allows to attach client-defined rules to the framework and execute them in an event-driven manner.

This chapter first gives a brief overview in Section 4.1 about the existing approaches for rule-based expert systems and model transformation frameworks. Section 4.2 and Section 4.3 introduce the EMF-IncQuery-based approach along with implementation details in Section 4.4. An additional extension for automatic rule execution is discussed in Section 4.5 and an example use-case is given in Section 4.6.

4.1 Survey of rule-based expert systems and model transformation frameworks

Rule based expert systems In its simplest form the Prolog programming language [17] can be thought of as a rule engine. It has its roots in the first-order logic and it is a declarative programming language where the logic is expressed in terms of relations, represented as facts and rules. The computation is initiated by running a query over these relations. However, the domain objects can be captured only with low level types thus not applicable to create complex domain-specific tools. On the other hand, JBoss Drools [5] is one of the leading open-source solutions for business rule management systems primarily tailored for the Java language. Drools is based on the RETE engine (like EMF-IncQuery) but adapts it as a more sophisticated object-oriented version opposed to the one presented in the original paper. It is capable of running rules written in numerous programming languages (Python, C#, Groovy) due to pluggable language extensions. The underlying knowledge base is represented as simple Java objects, but Drools is also flexible enough to match the semantics of the problem domain with domain specific languages via XML. The engine CLIPS [1] (C Language Integrated Production System) is also based on a RETE implementation. It also deals with rules and facts and is written in pure C, although CLIPS incorporates a complete object-oriented language COOL for writing expert systems. The Eclipse based framework, Jess is a re-implementation of the rule-based portion of CLIPS with the Java programming language.

Model and graph transformation frameworks

Apart from VIATRA2, GROOVE [34] is also a RETE-based graph transformation framework. Primarily it is intended to be used as a software model checking framework. The state space exploration is achieved by graph transformation, that is, it attempts to generate the whole state space by firing all applicable graph transformation rules. GROOVE uses non-attributed, edge-labelled graphs without parallel edges to represent the explored state space. TefKat [30] is a model transformation framework and it operates on the Eclipse Modeling Framework. It has a declarative SQL-like language to define transformation rules (with additional syntactic sugars like object literals) and change propagation is done by model merging.

4.2 Architecture overview

Figure 4.1 presents the designed architecture of the event-driven rule execution engine. The full stack is based on the **EMF-IncQuery** framework, which is used to execute queries over EMF models (for more details on the query language of EMF-IncQuery see [4]). Agenda, rule and activation are well-known terms from the context of rule-based expert systems (like JBoss Drools). A **rule** consists of a precondition (Left Hand Side - LHS part) which defines the condition that should met in order to the postcondition (Right Hand Side - RHS part) to become executable. The RHS in this context is not to be confused with the graph transformation rule's RHS or postcondition. There it usually defines some kind of model manipulations, however in this context, arbitrary actions (for example simply printing to the standard output, updating a user interface, etc.) can be set. In the context of the Rule Engine, the LHS consists of an EMF-IncQuery pattern and the RHS is a list of actions written in Java. An **activation** is a created for a rule when the preconditions (LHS) are fully satisfied with some domain model elements and the rule becomes eligible for execution (or in other words becomes fireable). An activation references the rule and

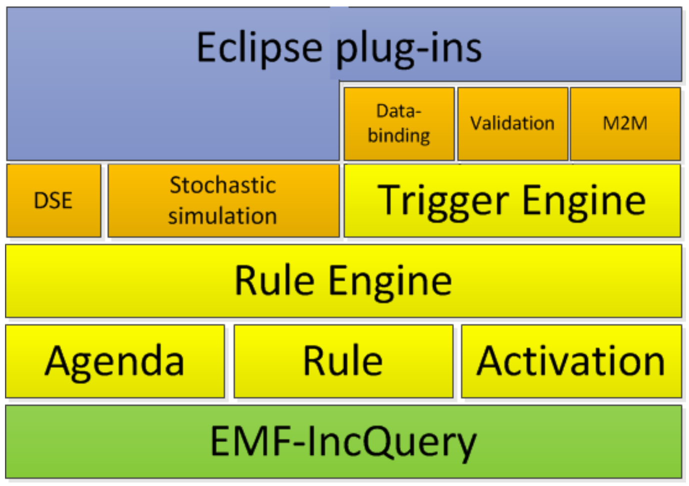


Figure 4.1: *Event-driven rule execution engine architecture*

the matched facts (model elements), and placed onto the agenda. The **agenda** keeps track of the changes of the activations (ie. the rules which are eligible to be applied) and provides an interface to query the applicable activations at any time.

The **Rule Engine** manages the lifecycle of the activations in the context of EMF-IncQuery and is able to maintain multiple agendas for different models. Activations are maintained after pattern match appearance, disappearance or update (meaning that the activation is still fireable, but the model elements that fulfill the LHS of the rule are updated). The **Trigger Engine** is a thin layer on top of the Rule Engine, which can be used to automatically execute applicable rules on the model. This way no user code is needed to fire the applicable activations, however there are certain scenarios when only the Rule Engine will be used, as the activation firing is done on an individual basis.

This stack is provided for the end users and sample use cases are shown on the figure. **Design space exploration (DSE)** and the **stochastic simulation** engine are built on the Rule Engine and use a self-managed rule execution strategy, however the EMF-IncQuery **Validation** Framework and **databinding** facilities rely on the Trigger Engine. Although the thesis does not deal with model to model transformations (**M2M**), it can be easily implemented on top of the Trigger Engine too. A similar approach is presented in [33] for creating trigger-based model transformations on top of VIATRA2.

4.3 Life cycle of activations

Figure 4.2 presents the state machine of the activations' lifecycle. The entry point is the **Inactive** state, where the activation is not present in the agenda. In the Active states the activations are eligible for execution (firing). The end user can define whether the **Disappeared** and **Updated** states will be used in the lifecycle.

- **Appeared**: the model elements that fulfill the LHS of the rule have just appeared in the model, resulting the creation of an applicable activation in the agenda. The user has not yet fired the activation.

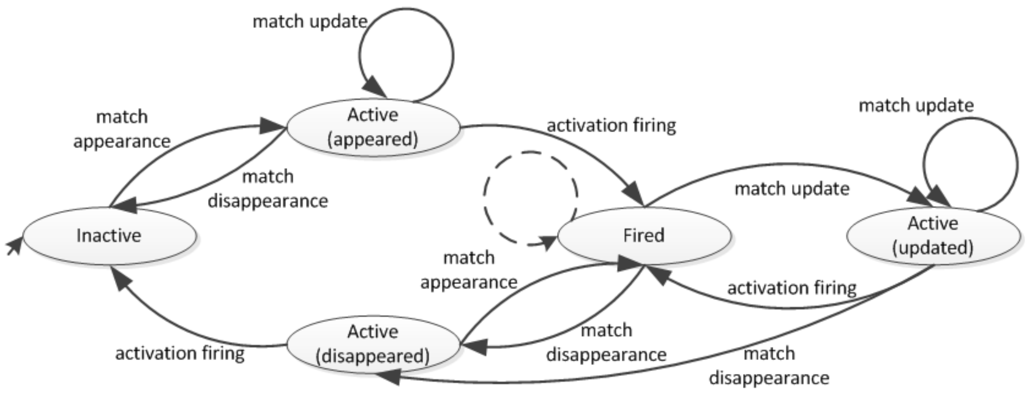


Figure 4.2: *The activations' lifecycle*

- **Disappeared**: the pattern match which fulfilled the LHS of the rule has disappeared after the activation was fired. The activation becomes fireable again so that the end user can process the disappearance of the pattern match.

- **Updated**: after the activation firing, the model elements (more specifically the values of their attributes) that fulfill the LHS have been modified. This results that the activation becomes fireable again.

The fireable activation's state is changed to **Fired** if it is applied. It can only be fired again if multiple firing of the same activation is enabled. This feature is used, for example, in Design Space Exploration (Section 5.2). Note that, it does not make sense to use multiple firing and the **Disappeared** and **Updated** states at the same time and it is enforced by the Agenda to not to do so.

Match appearance

Inactive ⇒ *Appeared* the precondition is satisfied and a new activation is created (this activation instance will be modified later, no new activation will be created in the other states).

Disappeared ⇒ *Fired* a previously fired activation's match appears again. The activation cannot be fired.

Match disappearance

Appeared ⇒ *Inactive* the match of the activation disappears resulting to remove it from the agenda.

Fired ⇒ *Disappeared* the match of a previously fired activation disappears resulting to become fireable again.

Updated ⇒ *Disappeared* the match of the activation which was in the *Updated* state disappears. The activation remains fireable after the state-change.

Match update

Fired ⇒ *Updated* the model elements of the match of a previously fired activation are modified. The activation becomes fireable again.

Appeared ⇒ *Appeared* self-loop to be used for consecutive model element manipulations for an activation which is in the *Appeared* state. The activation was not fired yet and remains fireable.

Updated ⇒ *Updated* self-loop to be used for consecutive model element manipulations for an activation which is in the *Updated* state. The activation remains fireable.

Activation firing

Active ⇒ *Fired*, *Updated* ⇒ *Fired*, *Disappeared* ⇒ *Inactive* the activation is fired and the postcondition of the activation's rule is applied. When the firing is done from the *Disappeared* state, the activation is removed from the agenda.

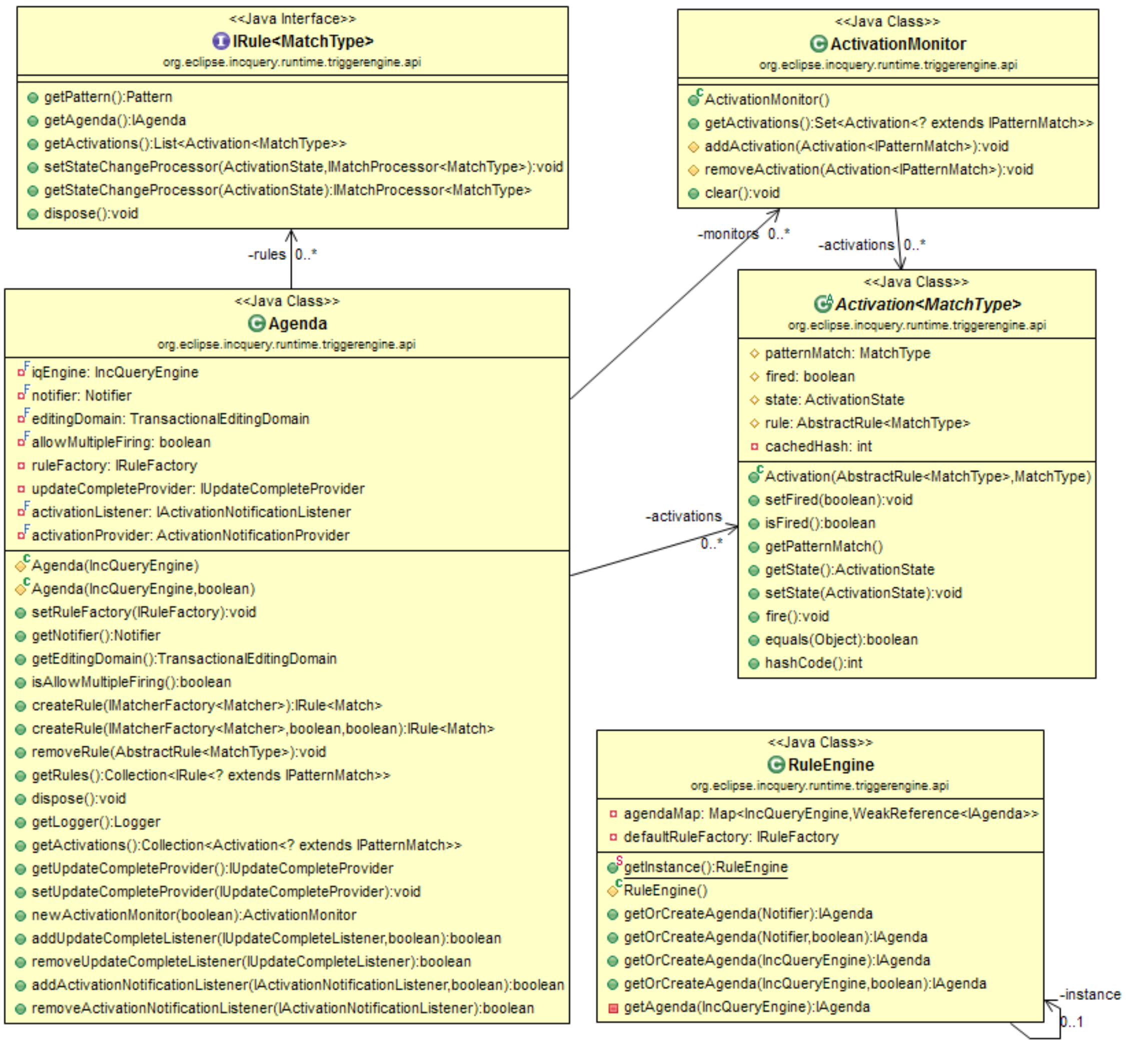
Fired ⇒ *Fired* this transition is only allowed if multiple firing of the same activation is enabled.

4.4 The designed API of the Rule Engine

The Rule Engine was implemented in Java as an Eclipse plug-in and the class diagram is presented on Figure 4.3.

- **RuleEngine**: it is a singleton class which is responsible to create and maintain multiple agendas for the various instance models.
- **Agenda**: an agenda is associated to an instance model and it provides an unmodifiable view for the collection of activations. Clients can create and remove rules through the agenda.
- **Rule**: a rule defines the underlying logic of the application which uses the Rule Engine. The rule's precondition is an EMF-IncQuery pattern and the postcondition is implemented in Java as an IMatchProcessor (the interface is defined in EMF-IncQuery) and one can define different jobs for match appearance, disappearance and update.
- **Activation**: an activation consists of the EMF-IncQuery pattern's match and the activation's state. Depending on the state, the appropriate job of the rule is executed upon firing.
- **ActivationMonitor**: instances of this class are used by clients to monitor and process the collection of activations within an agenda on an individual basis. This means that while the agenda always provides an up-to-date view of the applicable activations, an ActivationMonitor instance accumulates all the activations and the user

can clear the collection in the monitor when it is needed (for example after those have been processed). Upon instantiation it can be set to be filled with the initial collection of activations.



<<Java Interface>>
IRule<MatchType>
org.eclipse.incquery.runtime.triggerengine.api
getPattern():Pattern
getAgenda():IAgenda
getActivations():List<Activation<MatchType>>
setStateChangeProcessor(ActivationState,IMatchProcessor<MatchType>):void
getStateChangeProcessor(ActivationState):IMatchProcessor<MatchType>
dispose():void
-rules 0..*
<<Java Class>>
Agenda
org.eclipse.incquery.runtime.triggerengine.api
iqEngine: IncQueryEngine
notifier: Notifier
editingDomain: TransactionalEditingDomain
allowMultipleFiring: boolean
ruleFactory: IRuleFactory
updateCompleteProvider: IUpdateCompleteProvider
activationListener: IActivationNotificationListener
activationProvider: ActivationNotificationProvider
Agenda(IncQueryEngine)
Agenda(IncQueryEngine,boolean)
setRuleFactory(IRuleFactory):void
getNotifier():Notifier
getEditingDomain():TransactionalEditingDomain
isAllowMultipleFiring():boolean
createRule(IMatcherFactory<Matcher>):IRule<Match>
createRule(IMatcherFactory<Matcher>,boolean,boolean):IRule<Match>
removeRule(AbstractRule<MatchType>):void
getRules():Collection<IRule<? extends IPatternMatch>>
dispose():void
getLogger():Logger
getActivations():Collection<Activation<? extends IPatternMatch>>
getUpdateCompleteProvider():IUpdateCompleteProvider
setUpdateCompleteProvider(IUpdateCompleteProvider):void
newActivationMonitor(boolean):ActivationMonitor
addUpdateCompleteListener(IUpdateCompleteListener,boolean):boolean
removeUpdateCompleteListener(IUpdateCompleteListener):boolean
addActivationNotificationListener(IActivationNotificationListener,boolean):boolean
removeActivationNotificationListener(IActivationNotificationListener):boolean
<<Java Class>>
ActivationMonitor
org.eclipse.incquery.runtime.triggerengine.api
ActivationMonitor()
getActivations():Set<Activation<? extends IPatternMatch>>
addActivation(Activation<IPatternMatch>):void
removeActivation(Activation<IPatternMatch>):void
clear():void
-monitors 0..*
-activations 0..*
<<Java Class>>
Activation<MatchType>
org.eclipse.incquery.runtime.triggerengine.api
patternMatch: MatchType
fired: boolean
state: ActivationState
rule: AbstractRule<MatchType>
cachedHash: int
Activation(AbstractRule<MatchType>,MatchType)
setFired(boolean):void
isFired():boolean
getPatternMatch()
getState():ActivationState
setState(ActivationState):void
fire():void
equals(Object):boolean
hashCode():int
-activations
0..*
<<Java Class>>
RuleEngine
org.eclipse.incquery.runtime.triggerengine.api
agendaMap: Map<IncQueryEngine,WeakReference<IAgenda>>
defaultRuleFactory: IRuleFactory
getInstance():RuleEngine
RuleEngine()
getOrCreateAgenda(Notifier):IAgenda
getOrCreateAgenda(Notifier,boolean):IAgenda
getOrCreateAgenda(IncQueryEngine):IAgenda
getOrCreateAgenda(IncQueryEngine,boolean):IAgenda
getAgenda(IncQueryEngine):IAgenda
-instance
0..1

Figure 4.3: *Rule engine class diagram*

4.5 Notification mechanism and the Trigger Engine

Event-driven transformation execution is triggered by the changes in the domain model. The Rule Engine registers callbacks in the RETE net to receive notifications when a pattern match of a maintained rule's LHS appeared or disappeared, resulting a change in the collection of activations. Nevertheless, RETE does not issue notifications about the model element manipulations within a pattern match parameter, but those also need to be tracked in order to update the activation according to the lifecycle presented in Section 4.3. This requires to observe the values of the pattern match parameters' attributes and it is achieved with EMF databinding [1], which provides facilities to bind two data/information sources together in the context of EMF models and maintain synchronization between those. The rules define listeners for the model manipulations and bind them to the parameters of the pattern matches (which fulfill the LHS of the rule and result an activation). After that, these listeners will be automatically called - in a one-way synchronization scheme - when changes in the attributes' values occur. The rules then update the activations to reflect the changes in the pattern matches and their parameters' values. Through the rules the agenda provides an up-to-date view of the activations and those can be fired with the use of the Trigger Engine or some other external strategies.

Trigger Engine

The aim of the *Trigger Engine* is to provide a mechanism to automatically execute fireable activations without the need to write any client code. The main idea is that the Engine uses the *ActivationMonitor* of an *Agenda* and defines (i) when to fire an activation and (ii) which activation should be fired from the collection of fireable ones. This way the user should not worry about whether the *Rule Engine* has reached a consistent state in the aspect of activation updates, because the *Trigger Engine* handles this issue.

At the current state of development the *Trigger Engine* exposes the so called *AutomaticFiringStrategy* which does two things:

- The strategy is invoked by the *Rule Engine* after a match appearance/disappearance/update, which causes the event-driven behavior.

- Upon calling it selects the first fireable activation from the attached *Agenda* instance.

It can be seen that this is a very basic way of the timing and execution of the transformations, but it is sufficient for many scenarios (including the Validation Framework of EMF-IncQuery Section 5.1). In the future the *Trigger Engine* can be extended with additional firing strategies in order to provide more complex transformation execution.

The whole process of the flow of notifications is visualized on Figure 4.4.

[1]Databinding [2] is often used on user interfaces, when, for example, we want to bind a UI widget to a data source and automatically update the contents of that widget when the data source is changed. This approach deals with one-way synchronization, however, it is also possible to provide two-way synchronization e.g. between an input field and an information source. In this approach, when the user edits the contents of the input field, the data source is automatically updated.

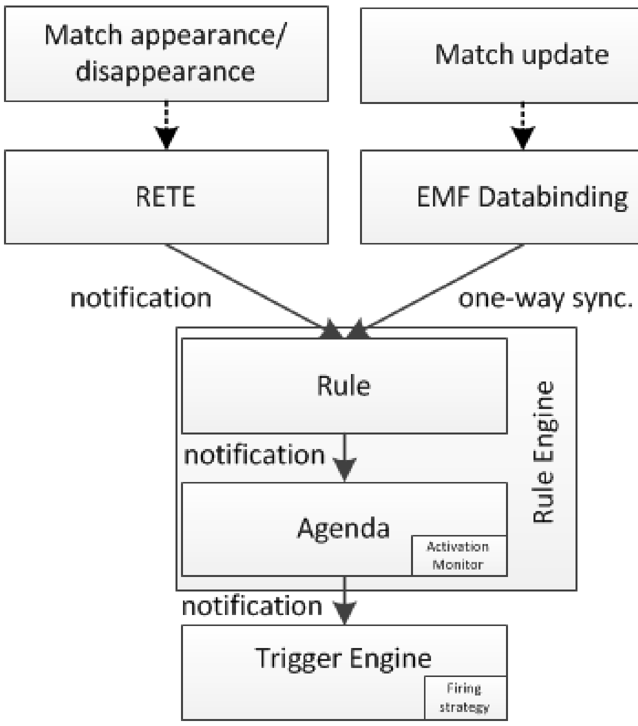


Figure 4.4: *Event-driven transformation execution - flow of notifications*

4.6 Example from the context of stochastic simulation

Listing 4.1 presents the VIATRA2 graph transformation rule (written in the VIATRA2 VTCL language) to create a new *linked* connection between any two super nodes which is not yet connected (in the current direction). The action keyword precedes the postcondition of the rule.

Listing 4.1: Graph transformation rule in VIATRA2

```
gtrule AddLink () = {
        precondition shareable pattern lhs(S1,S2) = {
                SN(S1);
                SN(S2);
                neg find linked(S1,S2);
        }
        action {
                let I1 = undef in new(SN.link(I1,S1,S2));
        }
}
```

In the context of EMF-IncQuery and the Rule Engine, first the precondition is formalised with an EMF-IncQuery pattern (see Listing 4.2). For more details on the *linked* pattern see Section 3.5.

Listing 4.2: Precondition of the graph transformation rule in EMF-IncQuery

```
pattern AddLinkPrecondition(S1 : SN, S2 : SN) = {
        neg find linked(S1, S2);
}
```

EMF-IncQuery provides code-generation facilities and an appropriate type-safe *IncQueryMatcher* class will be generated for the *AddLinkPrecondition* pattern. Instances of the *IncQueryMatcher* class keep track of the matches of the given EMF-IncQuery patterns. This matcher class will be used for the rule instantiation. In this case, only the *afterAppearanceJob* is set for this rule, which will be an instance of *AddLinkPreconditionProcessor* (Listing 4.3).

Listing 4.3: Postcondition of the graph transformation rule in EMF-IncQuery

```
public class AddLinkPostcondition extends AddLinkPreconditionProcessor {

        @Override
        public void process(SN S1, SN S2) {
                S1.getLink().add(S2);
        }

}
```

The whole rule is put together with the Java code presented in Listing 4.4. First a rule is created for the *AddLinkPrecondition* pattern by passing the generated factory of the *AddLinkPreconditionMatcher*. After that, the *AddLinkPostcondition* job is assigned to the rule.

Listing 4.4: Rule instantiation in the Rule engine

```
Rule<AddLinkPreconditionMatch> addLinkRule =
        agenda.createRule(AddLinkPreconditionMatcher.factory(), false, false);
addLinkRule.afterAppearanceJob = new AddLinkPostcondition();
```

Example triggers Suppose the *addLinkRule* is defined in the agenda and from now on, if two super nodes can be found in the network model which are not connected to each other then an activation with *Appeared* state will be created. The source of this trigger can be the designer by editing the model or some external tool which automatically creates model elements. Later on, if, for example the name of the super nodes are modified then the activation will become fireable again but now with *Updated* state. The source of this trigger can be again the designer, but in this case the mechanism is originated from the EMF databinding rather than the RETE net like in the previous case.

Chapter 5

Case studies

This chapter presents three case studies that heavily rely on the work presented in the previous chapters;

- The Validation Framework of the EMF-IncQuery tool to check constraints against EMF models (Section 5.1). This component is based on the Rule & Trigger Engine, and the pattern language of EMF-IncQuery was also extended with transitive closure as a language element.

- A Design Space Exploration (DSE) framework (Section 5.2) which is an EMF-IncQuery port of an original implementation based on VIATRA2. It is built on top of the Rule Engine and also uses the extended pattern language, however, it does not rely on the Trigger Engine as the activation firing is done by the DSE engine itself.

- A stochastic simulation framework - extended with efficient transitive closure support - which aims to provide a design-time analysis tool for dynamic and network systems (Section 5.3). This tool depends on the VIATRA2 framework.

5.1 Validation Framework

EMF-IncQuery provides the so called Validation Framework which can be used to check well-formedness constraints on various EMF instance models. A typical validation scenario with the framework is as follows:

1. Create a metamodel and instance model for the given domain.

2. Define patterns with the language of EMF-IncQuery. Out of these patterns the user can define (with annotation) which one should be treated as a constraint by the engine.

3. Initiate the validation on the given instance model with the set of previously created patterns. This can be achieved from the user interface of EMF-IncQuery, by providing pluggable context menu extensions to the pattern and instance model editors.

4. The constraints are evaluated on the model and upon violation, problem markers will appear on the user interface. The current implementation uses the Problems View (a view inside the workbench) of Eclipse, where various project-specific problem indicators are placed. These markers can have different severities (information, warning and error), typically a label which tells the crux of the problem and they can be traced back to the source of the problem (either a portion of code, a model element or even a misconfigured setting in a configuration file).

5. A key aspect of the solution is that the problem markers are data-bound to the problem sources, meaning that the model changes are automatically reflected on the user interface. This means, if the constraint violation is no longer present in the model, the appropriate marker is automatically removed or when the corresponding model elements are updated, but those still violate a constraint, the label of the marker is updated.

A prior validation framework was part of the VIATRA2 tool; during the thesis work I have adapted the solution to work in the EMF-IncQuery context. The new version is implemented as a thin layer on top of the Rule and Trigger Engine (see Chapter 4 for more details). It uses the Rule Engine to create transformation rules from the user-defined constraints (EMF-IncQuery patterns). The precondition of these rules will met if certain model elements violate the defined constraints and the transformation action will place/update/remove the appropriate problem marker in the Problems View. The postcondition is written in Java and it implements the *IMatchProcessor* interface. Additionally, the Trigger Engine is used to automatically fire the activations after a model manipulation resulting to update the user interface.

5.1.1 User interface extension

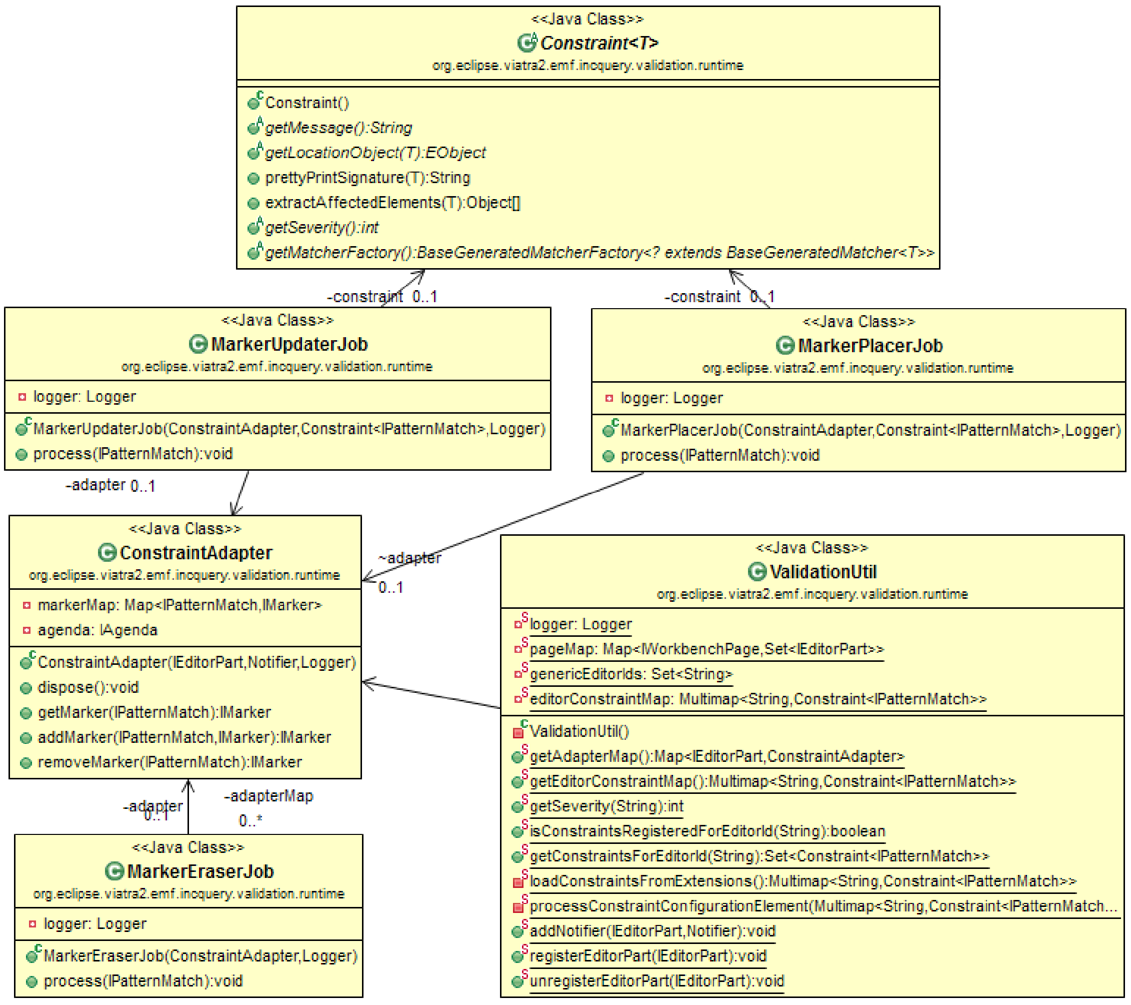
The Validation Framework exposes the *@Constraint* annotation to mark patterns as validation constraints. The definition of the annotation and its attributes is the following (from the EMF-IncQuery documentation):

Constraint: This annotation is used to mark a pattern for use in the EMF-IncQuery Validation Framework.

- **location**: The location of constraint represents the pattern parameter (the object) which the constraint violation needs to be attached to.
- **message**: The message to display when the constraint violation is found. The message may refer the parameter variables between $ symbols, or their EMF features, such as in $Parameter.name$.
- **severity**: Possible values: 'error' and 'warning'.
- **targetEditorId**: An Eclipse editor ID where the Validation Framework should register itself to the context menu. Use * as a wildcard if the constraint should be used always when validation is started.

5.1.2 The designed architecture

The class diagram of the framework-related components is shown on Figure 5.1.



<<Java Class>>
Constraint<T>
org.eclipse.viatra2.emf.incquery.validation.runtime
Constraint()
getMessage():String
getLocationObject(T):EObject
prettyPrintSignature(T):String
extractAffectedElements(T):Object[]
getSeverity():int
getMatcherFactory():BaseGeneratedMatcherFactory<? extends BaseGeneratedMatcher<T>>
-constraint 0..1
-constraint 0..1
<<Java Class>>
MarkerUpdaterJob
org.eclipse.viatra2.emf.incquery.validation.runtime
logger: Logger
MarkerUpdaterJob(ConstraintAdapter,Constraint<IPatternMatch>,Logger)
process(IPatternMatch):void
<<Java Class>>
MarkerPlacerJob
org.eclipse.viatra2.emf.incquery.validation.runtime
logger: Logger
MarkerPlacerJob(ConstraintAdapter,Constraint<IPatternMatch>,Logger)
process(IPatternMatch):void
-adapter 0..1
~adapter
0..1
<<Java Class>>
ConstraintAdapter
org.eclipse.viatra2.emf.incquery.validation.runtime
markerMap: Map<IPatternMatch,IMarker>
agenda: IAgenda
ConstraintAdapter(IEditorPart,Notifier,Logger)
dispose():void
getMarker(IPatternMatch):IMarker
addMarker(IPatternMatch,IMarker):IMarker
removeMarker(IPatternMatch):IMarker
<<Java Class>>
ValidationUtil
org.eclipse.viatra2.emf.incquery.validation.runtime
logger: Logger
pageMap: Map<IWorkbenchPage,Set<IEditorPart>>
genericEditorIds: Set<String>
editorConstraintMap: Multimap<String,Constraint<IPatternMatch>>
ValidationUtil()
getAdapterMap():Map<IEditorPart,ConstraintAdapter>
getEditorConstraintMap():Multimap<String,Constraint<IPatternMatch>>
getSeverity(String):int
isConstraintsRegisteredForEditorId(String):boolean
getConstraintsForEditorId(String):Set<Constraint<IPatternMatch>>
loadConstraintsFromExtensions():Multimap<String,Constraint<IPatternMatch>>
processConstraintConfigurationElement(Multimap<String,Constraint<IPatternMatch...
addNotifier(IEditorPart,Notifier):void
registerEditorPart(IEditorPart):void
unregisterEditorPart(IEditorPart):void
-adapter 0..1
-adapterMap 0..*
<<Java Class>>
MarkerEraserJob
org.eclipse.viatra2.emf.incquery.validation.runtime
logger: Logger
MarkerEraserJob(ConstraintAdapter,Logger)
process(IPatternMatch):void

Figure 5.1: *Validation Framework class diagram*

- **Constraint**: the Constraint (abstract) class is used for the extension point definition in the EMF-IncQuery framework. One may annotate arbitrary patterns with the *@Constraint* annotation (for more details see Section 5.1.1), marking the pattern as it will be used for validation purposes later. A *Constraint* subclass will be generated for every such pattern with additional information about the corresponding EMF-IncQuery pattern and the ones extracted from the annotation's attributes; target editor id, label of the problem marker and severity. More information on the code generation mechanism is given in Section 5.1.3.

- **ConstraintAdapter**: an instance of this class is assigned to every domain model which is loaded into the Validation Framework. Along with the domain model, the corresponding editor is also referenced (the one the model was loaded from) because it is possible to edit the same domain model in different kinds of editors or in different instances of the same editor. The Framework listens to the lifecycle of editors and upon editor closing, the appropriate problem markers will be removed from the Problems View.

The adapter collects the constraints from the *ValidationUtil* class and uses an *Agenda* to create the transformation rules that will update the user interface. The *MarkerPlacerJob*, *MarkerEraserJob* and *MarkerUpdaterJob* are used, respectively, to create a marker upon pattern match appearance, remove the marker when the match disappears and update the marker when the corresponding model elements are changed.

- **ValidationUtil**: the class collects the necessary helper methods to interact between the user interface and the Validation Framework, collects the available constraints from the workspace of the host Eclipse and manages the lifecycle of the *ConstraintAdapter*s.

5.1.3 Generated validation code

The EMF-IncQuery framework keeps track of the *@Constraint* annotations and if at least one is present on a pattern, a corresponding plug-in will be generated containing all the artifacts which are necessary for the Validation Framework. The name of the generated plug-in is originated from the name of the EMF-IncQuery project (which contains the annotated pattern(s)), but will end with *.validation*. This *.validation* plug-in can be used in a runtime Eclipse [1] configuration and the Validation Framework can be initialized from the context menu of the designated target editor.

The code generator is implemented with Xtend [9] which is a statically-typed programming language. Xtend compiles into Java source code and is a powerful library on top of the Java Development Kit with several additional facilities; for example lamda expressions, multiple dispatch, operator overloading, etc.

The crux of the type-safe *Constraint* subclass generation is done with template expressions and a portion of the corresponding code is given in Listing 5.1. Templates can be defined between the special characters '«' and '»' and basically the method defines simple string concatenations to produce the code.

The *getSeverity()* and *getMessage()* methods extract the necessary information from the *@Constraint* annotation, while the *getLocation()* method uses the passed pattern match to get the appropriate pattern parameter which will be the location object. This location object will be used in the problem marker as a link to trace back to the actual model element.

The *getMatcherFactory()* returns the generated factory of the *IncQueryMatcher*, associated to the annotated pattern. This factory will be used by the *ConstraintAdapter* class to create the pattern matcher for the constraint.

[1] During Eclipse development we distinguish between the host and runtime Eclipse configurations. The developer usually implements the various plug-ins in the host Eclipse, where the numerous development tools are installed too. After that, one can select the necessary plug-ins (from to host workspace) to form a runtime Eclipse configuration, where only these plug-ins will be loaded. This way it is much easier to maintain the dependent plug-ins of a project and also reduces the hardware resources needed for development. The runtime Eclipse configuration will always run in a separate Java Virtual Machine.

Listing 5.1: Xtend template for Constraint generation

```
private static String annotationLiteral = "Constraint"
...
def getElementOfConstraintAnnotation(Annotation annotation, String elementName) {
  val ap = annotation.getAnnotationParameterValue(elementName)
  if(ap != null && ap.size == 1) {
    return (ap.get(0) as StringValue).value
  } else {
    return null
  }
}
...
def patternHandler(Pattern pattern, Annotation annotation) '''
  package <<pattern.packageName>>;
  ...
  //imports are omitted here
  ...

  public class <<pattern.name.toFirstUpper>><<annotationLiteral>>
        <<pattern.annotations.indexOf(annotation)>>
        extends <<annotationLiteral>><<<pattern.matchClassName>>> {

    private <<pattern.matcherFactoryClassName>> matcherFactory;

    public <<pattern.name.toFirstUpper>><<annotationLiteral>>
        <<pattern.annotations.indexOf(annotation)>>() throws IncQueryException {
      matcherFactory = <<pattern.matcherFactoryClassName>>.instance();
    }

    @Override
    public String getMessage() {
      return "<<getElementOfConstraintAnnotation(annotation, "message")>>";
    }

    @Override
    public EObject getLocationObject(<<pattern.matchClassName>> signature) {
      Object location = signature.
        get("<<getElementOfConstraintAnnotation(annotation, "location")>>");
      if(location instanceof EObject){
        return (EObject) location;
      }
      return null;
    }

    @Override
    public int getSeverity() {
      return ValidationUtil.getSeverity(
        "<<getElementOfConstraintAnnotation(annotation, "severity")>>");
    }

    @Override
    public BaseGeneratedMatcherFactory<<<pattern.matcherClassName>>>
        getMatcherFactory() {
      return matcherFactory;
    }
  }
'''
```

5.1.4 Validation example from the context of stochastic simulation

Suppose the designer is creating the instance model of a stochastic simulation experiment. During editing, we would like to give support for the designer to be able to check whether the model is error-free. In this case study only one constraint is used, which prohibits the existence of a client node that is not connected to at least one super node. This constraint is presented in Listing 5.2. The *notConnectedClient* pattern is used as a constraint (see the *@Constraint* annotation definition) which matches when *connectedClient* does not have a match for the given CL (client) node.

Listing 5.2: Validation Framework constraint example

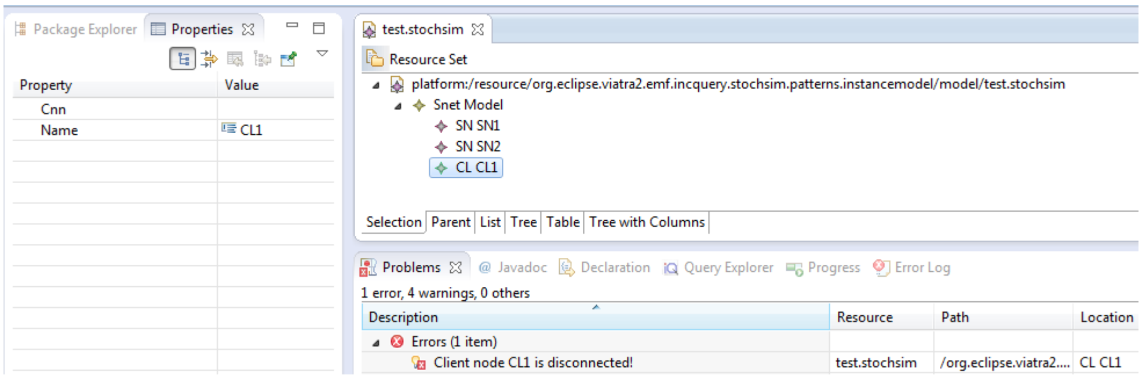
```
package org.eclipse.viatra2.emf.incquery.stochsim.patterns

import "http://org.eclipse.viatra2.emf.incquery.stochsim.model"

@Constraint(location = "CL", message = "Client node $CL.name$ is disconnected!",
        severity = "error", targetEditorId = "*")
pattern notConnectedClient(CL : CL) {
        neg find connectedClient(CL);
}

pattern connectedClient(CL : CL) {
        CL.cnn(CL, _SN);
}
```

On Figure 5.2 an instance model is given with two super nodes $S1$ and $S2$ and a client node $CL1$ which is not connected to any of the super nodes. The Validation Framework can be initialized from the context menu of the tree viewer editor. The constraints are collected from the workspace of the host Eclipse and those are evaluated on the loaded instance model. A constraint violation results an error marker in the Problems View (which is the bottom view on the figure). The severity of the constraint was defined as 'error' and the $CL1$ model element is assigned as the location of the constraint violation. Suppose the model is changed by the designer; for example a new connection is established between the $CL1$ node and one of the super nodes. The state of the activation - that was fired to create the error marker - changes from *Fired* to *Disappeared* and it is automatically applied (again) by the Trigger Engine. This results the immediate removal of the corresponding error marker from the Problems View, meaning that the designed model is error free.



Property	Value
Cnn	
Name	CL1

Resource Set
platform:/resource/org.eclipse.viatra2.emf.incquery.stochsim.patterns.instancemodel/model/test.stochsim
Snet Model
SN SN1
SN SN2
CL CL1

1 error, 4 warnings, 0 others

Description	Resource	Path	Location
Errors (1 item)			
Client node CL1 is disconnected!	test.stochsim	/org.eclipse.viatra2....	CL CL1

Figure 5.2: *Validation Framework example model and constraint violation*

Note that, the marker is data-bound to the corresponding model elements (and their attributes) - in this case $CL1$ - of the match, so that when the violating model elements are modified, e.g. renamed, the literal of the marker is updated in the Problems View even if the match set itself does not change. If the designer editor (where the validation was initialized from) is closed the appropriate markers are also disposed of.

5.2 Design space exploration

Design space exploration (DSE) [24] is a process to compare and analyze functionally equivalent implementation alternatives, which meets all design constraints in order to identify the most suitable choice (solution) based on various metrics, such as performance, cost or reliability. DSE is widely used in many areas where model-driven development is already popular; it is a challenging problem for the design of safety critical systems, runtime reconfiguration of IT systems or automotive design scenarios. DSE can be used at design-time to choose between the various solutions and also during runtime, for example, to reconfigure the model of a cloud infrastructure to meet certain target functions and properties.

There are two requirements for a DSE solution: (i) it must be reachable from the initial state through a sequence of applications of given exploration rules (ii) it must meet the global constraints of the given problem. While traditional DSE problems use only numerical constraints (e.g. time, cost, etc.), nowadays' software and hardware systems require to define complex structural constraints of the graph-based model of the system-under-design. Nevertheless, when applying DSE during runtime the framework must handle dynamic creation and deletion of model elements too.

In [24] a VIATRA2 based solution is presented for guided design space exploration. During the thesis work portions of this approach were adapted to make it work on EMF models and to use EMF-IncQuery as the underlying graph pattern matching framework. The aim of the tool was to create a proof of concept solution for EMF models and the Rule Engine was the component which eased the development in many aspects. Indeed, the automatically updated Agenda is really useful in the DSE scenarios and the major part of the implementation effort could be focused on the exploration strategies/techniques.

A detailed survey of existing DSE solutions in given in [24].

5.2.1 Overview of the VIATRA2 based DSE framework

A schematic overview of the architecture of the VIATRA2-based tool is given on Figure 5.3. DSE can be thought of as traversing through the search tree of the system-under-design. In this solution the system states are stored as nodes of a graph, operations are defined as graph transformations and goals and constraints are given as graph patterns. Naturally, the aim is to find the optimal solution under the shortest possible time and this needs to leave out certain parts of the search tree where it is guaranteed that no solutions can be found. To reduce the number of alternatives that are evaluated, several hints are used during exploration:

- **Rule dependency** to define a partial ordering between the transformation rule applications. This is derived from the analysis on the Petri net abstraction of the system.
- **Occurrence vector** is used to define the maximal application number of a given rule during exploration.
- **Cut-off criteria** is used to identify dead end states.
- **Selection criteria** defines priorities between the applicable exploration rules.

The search process consists of the following steps:

1. **Check operation applicability**: check executability of transformation rules.
2. **Evaluate criteria**: based on the hints, applicable rules are selected from the ones that are fireable.
3. **Cut-off**: if the cut-off criteria is met or no applicable rule is present, then the state is a dead end and can be cut off.
4. **Select rule**: select a rule from the evaluation results which will be applied by the engine.
5. **Apply rule**: a new state will be reached with the rule application. Recording of the model manipulations are required in order to be able to backtrack the step if it is needed later.
6. **Check new state**: checking global constraints and goals on the current state. If it is an invalid state, then the exploration must be backtracked to the parent state. If the state represents a solution, the engine saves it and it may stop if only one solution is needed.
7. **Continue search**: the process is started again from the new state or from the parent state after a backtrack.

The framework can be extended with various exploration strategies; depth- or breadth-first search (possibly with depth limit) and heuristics based algorithms like A*, etc.

5.2.2 Overview of the EMF-IncQuery based DSE framework

During the thesis work I have adapted the framework presented in Section 5.2.1 to work on top of the EMF-IncQuery framework and the Rule Engine. The components denoted with red text (Figure 5.3) are included in my prototype implementation, that is, the current state of the new framework uses rule priorities as guidance to reduce the number of states being explored. In this case these priorities are set by the designer not by some external tools and the exploration is applicable for EMF instance models with rule postcondition written in Java (for the Rule Engine). This solution is aimed to be used at runtime for exploring solutions in a dynamically changing model.

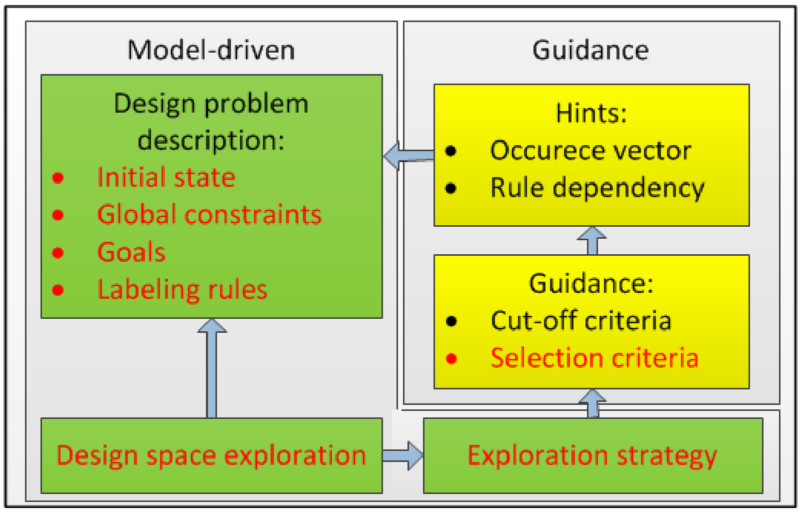


Figure 5.3: *Architecture of the VIATRA2 based DSE tool*

- **Design problem description**
  - **Initial state** is given by the starting EMF instance model. All states are reachable from this initial model during exploration.
  - **Goals** are defined as EMF-IncQuery patterns. If in the current state, all patterns (which correspond to a goal) have a match then the state represents a solution. If the current state does not satisfy all of the goals then it is possible that further rule applications will result in an expected solution. Note that these patterns are often parameterless queries which only check the existence of certain model elements. It can be set for the framework to finish exploration when a solution is found or continue it until a given number of states are explored.
  - **Global constraints** are like goals, but they are used to restrict the valid states during exploration. In fact, if at least one of the global constraints is violated in the current state then the DSE framework backtracks to the previous state as it is not possible to reach a goal on that specific branch of the search tree. Such constraints are often used to prohibit the existence of a certain structure in the model.
  - **Exploration/transformation rules** are defined with the Rule Engine, that is, an EMF-IncQuery pattern as the precondition and a Java *IMatchProcessor* as the postcondition. See Section 5.2.3 for a specific example.
- **Design space exploration**: the framework is much like the VIATRA2 based conforming to much of the existing interfaces.
- **Exploration strategy**: the prototype implementation contains a depth-first search based exploration strategy with fixed depth-count for the traversal. In this scenario, the algorithm knows the length of the optimal trajectory for the given problem and sets a depth limit for the traversal by 1.5 times of the optimal length. Also, if a solution is found with length L, the algorithm sets the depth limit to $L-1$, meaning

that from this point only shorter solutions are taken into account. If the optimal trajectory is found, the exploration is stopped.

- **Guidance**

  - **Rule priorities** can be defined for the exploration rules with annotations. The rules and their priorities are stored in a map structure, where the key is the priority value and the value is a list containing the rules with the given priority. If there are two or more active (fireable) rules with different priorities are present, then the framework applies the one with the lower priority value. For rules with the same priority value, the first from the list is fired.

5.2.3 Cloud infrastructure example

This section gives an example for the various terms presented in Section 5.2.2 through the design of a simple cloud infrastructure and the evaluation of the prototype tool was also carried out with this example. The used metamodel is presented on Figure 5.4. A cloud model (*CloudModel*) represents the cloud infrastructure which is under (re)configuration. *CloudCompNode* is the superclass of all components within the model. Cloud nodes (*CloudNode*) are units of cloud based configurations and are the top elements inside a cloud model. High availability clusters (*ClusterNode*) and servers (*ServerNode*) can be added onto cloud nodes. Databases (*DbNode*) are installed on servers and applications (*AppNode*) are executed over databases. Servers can also be deployed on clusters while storage elements (*StorageNode*) can only operate on clustered servers.

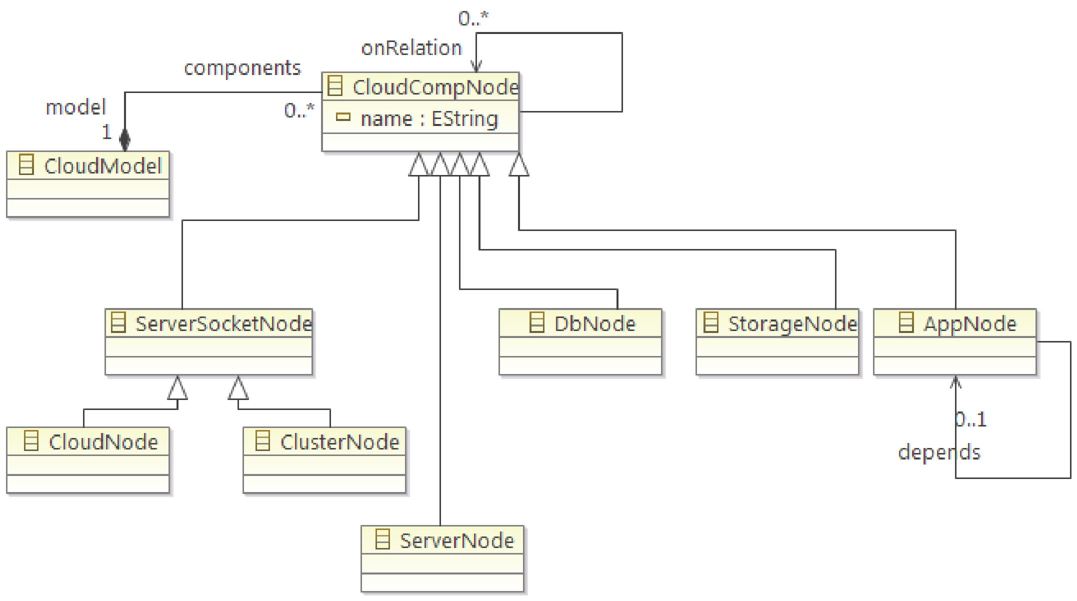


Figure 5.4: *Metamodel of the cloud infrastructure*

Initial state

The initial state consists of a *CloudModel* instance.

Global constraints

Listing 5.3 presents the *limitedUnusedNodes* and *nonClusteredDB* constraints. The first is used to prohibit the existence of more than 5 unused *CloudCompNode* instances, that is, the ones on which no other nodes are deployed. The second constraint is used to make sure that a *DbNode* instance is always deployed on top of two clustered *ServerNode* instances. The rest of the patterns are helper patterns which are used in the constraints.

Listing 5.3: Cloud infrastructure DSE global constraints

```
//Global constraint - no matches
pattern limitedUnusedNodes() = {
        N == count find unusedNode(_Node);
        check(N > 5);
}

//Global constraint - no matches
pattern nonClusteredDB(DB) = {
    DbNode(DB);
    ServerNode(server1);
    ServerNode(server2);
    server1 != server2;
    find nodeOnHost(DB,server1);
    find nodeOnHost(DB,server2);
    neg find serversOnCluster(server1,server2);
}

//Helper patterns
pattern unusedNode(Node) = {
        CloudCompNode(Node);
        neg find nodeOnHost(_OnNode, Node);
}

pattern nodeOnHost(Node,Host) = {
    CloudCompNode.onRelation(Node, Host);
}

pattern serversOnCluster(server1, server2) = {
        ServerNode(server1);
        ServerNode(server2);
        server1 != server2;
        ClusterNode(cluster);
        find nodeOnHost(server1, cluster);
        find nodeOnHost(server2, cluster);
}
```

Exploration rules

Some exploration rules are given through the following listings (for in-depth details see the source code); first the precondition is given as an EMF-IncQuery pattern and after that the Java postcondition used by the Rule Engine.

AddDBToServer: the rule can be applied on two (free) *ServerNode* instances and as a result a *DbNode* instance will be created on top of the two servers.

Listing 5.4: AddDBToServer rule precondition

```
pattern addDBToServerGT(server1, server2) = {
        ServerNode(server1);
        ServerNode(server2);
        server1 != server2;
        find nodeOnHost(server1,Host);
        find nodeOnHost(server2,Host);
        neg find busyServer(server1);
        neg find busyServer(server2);
}
```

Listing 5.5: AddDBToServer rule postcondition

```
public class AddDBToServerJob extends AddDBToServerGTProcessor {

        @Override
        public void process(ServerNode server1, ServerNode server2) {
                DbNode db = CloudFactory.eINSTANCE.createDbNode();
                db.setName("DB"+server1.getName()+server2.getName());
                db.getOnRelation().add(server1);
                db.getOnRelation().add(server2);
                server1.getModel().getComponents().add(db);
        }
}
```

AddAppToDB: an application node can be deployed on two *DbNode* instances (if an other application is not deployed on them yet). Upon rule application the appropriate *onRelation* references are set.

Listing 5.6: AddAppToDB rule precondition

```
pattern addAppToDBGT(db1, db2) = {
        DbNode(db1);
        DbNode(db2);
        db1 != db2;
        neg find busyDB(db1);
        neg find busyDB(db2);
}
```

Listing 5.7: AddAppToDB rule postcondition

```
public class AddAppToDBJob extends AddAppToDBGTProcessor {

        @Override
        public void process(DbNode db1, DbNode db2) {
                AppNode appNode = CloudFactory.eINSTANCE.createAppNode();
                appNode.setName("A"+db1.getName()+db2.getName());
                appNode.getOnRelation().add(db1);
                appNode.getOnRelation().add(db2);
                db1.getModel().getComponents().add(appNode);
        }

}
```

Goals

The expected solutions/goals are discussed in Section 6.3.1 along with the analysis of the measurements results.

5.2.4 Implementation notes - EMF transactions

The crucial aspect of a design space exploration engine is to efficiently (i) execute the exploration rules and (ii) provide a mechanism to backtrack the current state to the previous (parent) state. The approach deals with the first part with the use of the Rule Engine (Chapter 4), however, for the latter, the EMF Model Transaction (EMF-MT) API is used. It provides automatic data integrity for EMF models when multiple model manipulations occur simultaneously. Key points regarding the use of EMF MT are as follows:

- *TransactionalEditingDomain*: it is an EMF MT service which extends the standard *EditingDomain's* functionality by applying transactional semantics for reading and writing the model. Using the service results that the model can only be manipulated within a transaction from now on.

- *Command*: represents an executable (possibly undoable, redoable) portion of code that manipulates the model. One can still use arbitrary Java code to manipulate the EMF model; the DSE engine uses the *RecordingCommand* to record the effects of rule executions and wrap them into a *Command* structure that can undo or redo the changes. The backtracking of operations is achieved by undoing these changes.

- *CommandStack*: the command stack collects all the commands that were executed on a given *EditingDomain*. The *undo()* operation of the *CommandStack* pops the top of the stack and undoes the last command if it is undoable.

5.3 Stochastic simulation

To support the design-time analysis of dynamic systems, a simulation framework for generalized stochastic graph transformation [28] has been introduced in [38], built on the foundations of the RETE-based pattern matching infrastructure of VIATRA2. The Graph-based Stochastic Simulation (GRaSS) uses graph transformation rules that are augmented with probability distributions governing the delay of their application, for the purpose to derive continuous-time semi-Markov processes to verify stochastic properties of the system-under-design. As part of the thesis work, the transitive closure language extension was also introduced into VIATRA2 (via annotation) and it became possible to use it in the GRaSS tool. In our paper [14], we have carried out a case study with this language extension where we used it in statistics-collecting probe rules as well as in behavioral rules resulting to improve the expressive power of the tool. The main result of this work is that it is now possible to define parts of a complex simulation step with efficient transitive closure support in a declarative way. The implementation outperforms the previous VIATRA2 based approaches and the full performance evaluation is given in Section 6.2.1.

We have presented a simple modeling scenario for a Skype-like system in [14] originated from [38][27]. The concepts, metamodel and example instance model of the system is presented throughout Chapter 2.

5.3.1 Survey of stochastic network simulation tools

Generalised stochastic graph transformation (as supported by the GRaSS tool [38]) has been presented as a modelling language that is intuitive as well as very expressive. In the context of peer-to-peer VoIP simulation, the use of real-valued attributes and generalised probability distributions can help in representing traffic at a higher level; indeed, realistic modelling of jitter and bandwidth have shown to be possible [27][28]. As always with discrete event systems [16], the size of the system can be a problem, and the queue of the possible events at each simulation step can grow very large. Previously, with basic dedicated tools such as *ns*2 [15] and *GloMoSim* [41], simulations of at most hundreds or thousands of network nodes were possible. With today's advanced tools, networks of thousands to tens of thousands of nodes have become feasible on a modern workstation [40]. The performance of the transitive closure extension is highly relevant to match the scalability of the GRaSS tool. Additionally, as transitive closures can now be used both in statistics-collecting probe rules as well as behavioral rules, the expressive power of the tool has also been extended.

5.3.2 Simulation with graph transformations

In order to experiment with the characteristics of a continuously changing Skype network, the graph based model of the system is used. To illustrate the behaviour of the network we have used several graph transformation rules, each one of them assigned a probability distribution. During a simulation run, the execution of transformation rules is automatically scheduled according to a round-based scheme that maintains a priority queue. The execution engine, with the aid of the incremental pattern matcher, keeps track of enabled and disabled rules by registering changes in the match sets of the rule precondition patterns. Probe rules - the ones without postcondition - are used to measure important properties of the network throughout the simulation runs.

The majority of the rules were presented in [38] with slight fine-tuning for the extended pattern language. The VIATRA2 VTCL code of the rules are given in the appendix.

Behavioral rules

CreateClient (Listing A.4.1) Creates a new client node when applied, however it can only be applied if the number of disconnected nodes is less than 5. This is used to avoid the creation of too many isolated nodes during simulation.

DeleteClient (Listing A.4.2) Deletes an isolated client node.

ConnectClient (Listing A.4.3) Connects a client node to the super node which has the fewest clients connected to it. This way load balancing mechanism can be introduced to the simulation.

DisconnectClient (Listing A.4.4) Disconnects a client node from the super node that it was connected to.

UpgradeClient (Listing A.4.5) Upgrades a client node to a super node, if (i) the super node it is connected to has at least 4 connections to clients and (ii) there are at least 3 isolated clients available. As a result, a new super node is created and the isolated clients will be connected to this new node. Figure 5.5 displays the process of upgrading a client.

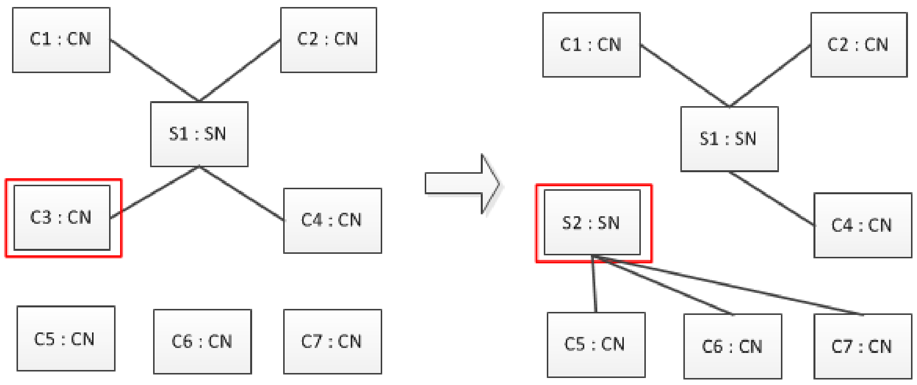


Figure 5.5: *Stochastic simulation - UpgradeClient rule*

DowngradeSuperNode (Listing A.4.6) Downgrades super node S1 which is linked to super node S2, resulting to remove S1 and create a new client node which is connected to S2.

AddLink (Listing A.4.7) Establishes a new link between two super nodes which are not currently linked.

Probe rules

P_NetworkSize (Listing A.4.8) Probe rule used to log the size of the network; number of super nodes and client nodes together.

P_ConnectedOverlayPairs (Listing A.4.9) Probe rule to report the number of pairwise linked overlay (SN - Super Node) pairs. This rule uses the transitive closure of the overlay network for computing the reachability regions. More details on the *transitiveClosureOfLinked* pattern can be found in Section 3.5.1.

5.3.3 Distributions and simulation execution

The stochastic graph transformation system is defined as a triplet $S = \langle R, G_0, F \rangle$; R is the set of rules (ε_R is the corresponding set of events), G_0 is the initial graph model and

$F : \varepsilon_R \to (R \to [0, 1])$ assigns each event a continuous distribution function. The engine uses exponential distribution functions, that is, $F(x) = 1 - e^{\lambda x}$ if $x > 0$ and 0 if $x \leq 0$. The exponential distribution function is memoryless (or forever young) which means that the probability of an event does not depend on the past events, only on the current state of the system.

The execution of simulations is described in-depth in [38].

During the simulation one can set the number of runs per experiment and the maximal depth; either by the number of steps or simulation time. The implementation uses the SSJ Java library [6], which provides facilities to deal with probability distributions and collects statistics.

5.3.4 Connection with the Rule Engine

Currently the GRaSS tool is implemented as a VIATRA2 extension, but it naturally comes to mind that in an EMF-IncQuery version the Rule Engine could be easily applied as an option to create the model transformation rules. In fact, the solution would probably not need the Trigger Engine as the activation firing in this scenario is evaluated by a more complex, time-dependent logic.

Chapter 6

Performance measurements

This chapter presents three benchmarking scenarios to evaluate the performance characteristics of the various components developed as part of the thesis work:

- Performance comparison of the transitive closure algorithms (described in Chapter 3) over graph models.
- Benchmarking of simulations carried out with the stochastic simulation framework (presented in Section 5.3). These measurements mainly dealt with the transitive closure extension in the VIATRA2 framework based on the DRed and IncSCC algorithms. It also proves the applicability of incremental transitive closure computation within a complex case study.
- Experimenting with the Design Space Exploration component (described in Section 5.2).

The performance measurements were carried out on Intel Core i5 2,5 GHz, 8GB RAM, Java Hotspot Server vm build 1.7.0_07-b10 on 64-bit Windows 7 Professional. Eclipse 4.2 x64 (Juno) release was used as the measurement environment in all scenarios. Throughout the stochastic simulations 4 RETE threads were used in the VIATRA2 framework.

6.1 Benchmarking of the transitive closure algorithms

6.1.1 Measurement scenario

To compare the performance characteristics of the implemented transitive closure algorithms I have used two directed graph models (one dense and one sparse):

- Erdős-Rényi graph [19]: the p parameter of this graph defines the probability that a given edge exists in the graph, independently of other edges. The first measurement scenario used this graph, with $p = 0,005$ while increasing the number of vertices from 1000 to 20000. A key observation for this model is that it results a lot of edges even for a small values of p; a graph with 20000 vertices will have about 2 million edges after generation.

- MinMax graph: the name is originated from the generation method of this graph; *minDegree* defines the minimum out-degree of a vertex, while *maxDegree* defines the maximal out-degree. During model generation each vertex will be assigned a uniform random number between $[minDegree, maxDegree]$ which defines the number of outgoing edges. DAG generation is done by only allowing to insert an edge where the index of the target vertex is larger than or equals to the index of the source vertex. The DAG version was generated with $minDegree = 2$ and $maxDegree = 5$, while the number of vertices ranged from 1000 to 10000. The non-DAG version used the same parameters but the number of vertices reached 100000.

The measured parameters and properties consist of the following ones:

- Number of vertices and edges in the generated graph, and in case of non-DAG graphs, the number of SCCs in the initial model.
- The wall-times for the incremental transitive closure algorithms; the time spent to initialize the transitive closure relation and the average time spent for relation maintenance after a series of random edge insertion / deletion. The time values are always in milliseconds.
- The initialization time (in milliseconds) for the static transitive closure algorithms.

Experiments were carried out for the DRed, King, Counting and IncSCC incremental algorithms and for the FW - Floyd Warshall and DFS - depth-first search based static algorithms (see Chapter 3 for more information). During measurements the time limit was set to 2 minutes.

6.1.2 Results and analysis

Table 6.1 presents the first series of measurements on the Erdős-Rényi graph. A key observation is that the static algorithms do not scale well for larger models and also the large amount of edges affects the performance in a negative way. The King algorithm is not presented in this series, because even for the smallest graph, the initialization time exceeds 2 minutes. Another important fact is that DRed handles edge deletions with fixpoint computation on the transitive closure relation which costs more computation time when the relation is large. It can be seen that from 3000 vertices the whole graph is strongly connected. IncSCC takes advantage over this property and maintains the reduced graph efficiently, however the initialization takes more than 2 minutes for the Tarjan algorithm implementation on the graph with 20000 vertices. Indeed, its performance is affected by the number of edges in the graph, which is more than 2 million for that graph.

Table 6.2 presents the measurement results on MinMax DAGs. In this scenario the King implementation's update time is competitive with the results of IncSCC, however, the initialization time follows a rapid growth in time with the increasing model size and for a graph with 5000 vertices it takes more than 2 minutes. The update time for Counting and IncSCC is almost exactly the same, which is the expected result, as in this case, IncSCC

Erdős-Rényi (0,005)			DRed			FW
# vertices	# edges	# SCCs	init	insert	delete	init
1000	5063	14	546	214	33116	2341
3000	45149	1	18310	4232	>120000	55130
5000	124470	1	116596	17961	>120000	>120000
10000	500324	1	>120000	-	-	-
20000	2000135	1	-	-	-	-
Erdős-Rényi (0,005)			**IncSCC**			**DFS**
# vertices	# edges	# SCCs	init	insert	delete	init
1000	5063	14	140	0	8	1264
3000	45149	1	415	0	10	28860
5000	124470	1	1390	0	33	114301
10000	500324	1	19536	0	186	>120000
20000	2000135	1	152832	0	1064	-

Table 6.1: *Measurements on transitive closure algorithms - Erdős-Rényi graph*

works as a simple wrapper for Counting, passing all edge insertions and deletions to be handled by the latter one. Note that, however, the two implementations use a different method for initialization.

MinMax graph (2,5)			DRed			Counting			FW
# vertices	# edges	DAG?	init	insert	delete	init	insert	delete	init
1000	3541	yes	372	14	230	219	0	6	1560
3000	10472	yes	981	0	3097	1295	0	4	39032
5000	17589	yes	2782	3	6817	3385	0	137	> 120000
10000	35036	yes	10581	0	29501	16896	0	265	-
MinMax graph (2,5)			**KING**			**IncSCC**			**DFS**
# vertices	# edges	DAG?	init	insert	delete	init	insert	delete	init
1000	3541	yes	8518	33	2	218	0	8	187
3000	10472	yes	129527	24	17	1201	0	11	828
5000	17589	yes	>120000	-	-	3182	0	167	1762
10000	35036	yes	-	-	-	15070	0	269	10952

Table 6.2: *Measurements on transitive closure algorithms - MinMax DAG*

Measurement results on MinMax non-DAGs are presented in Table 6.3. Here, again, King is not included because the initialization takes more than 2 minutes on the smallest graph. IncSCC scales well along the increasing model sizes and the average update time also remains under 0,5 seconds for each case. Note that, there is major difference between the scalability of static algorithms for cyclic and acyclic graphs; for example the DFS-based algorithm generates the relation for the MinMax DAG with 10000 vertices under 11 seconds, while it takes more than 2 minutes for non-DAG version (see Listing A.3.1 for more details). The scaling of runtime is also visualized on Figure 6.1 (note that, the metric of the y-axis is logarithmic with base 10).

Summarizing the key experiences from the measurement results, it can be seen that:

- The King implementation's initialization is not efficient, which is due to the multilevel data structure used to store the relation. The performance of its update mechanism is competitive with the relation maintenance of IncSCC or Counting.

- Most of the algorithms - except IncSCC - scale poorly for larger graphs because they do not utilize the existence of large SCCs in the graph. For a quite dense graph, there

MinMax graph (2,5)				DRed			FW
# vertices	# edges	# SCCs	DAG?	init	insert	delete	init
1000	3548	26	no	1295	235	23150	2293
3000	10559	114	no	14556	2528	>120000	49422
5000	17495	186	no	75629	14378	>120000	>120000
10000	34874	330	no	>120000	-	-	-
30000	104749	1044	no	-	-	-	-
50000	174407	1746	no	-	-	-	-
100000	350251	3348	no	-	-	-	-

MinMax graph (2,5)				IncSCC			DFS
# vertices	# edges	# SCCs	DAG?	init	insert	delete	init
1000	3548	26	no	47	5	21	1232
3000	10559	114	no	124	0	11	17738
5000	17495	186	no	204	0	16	65506
10000	34874	330	no	406	21	31	>120000
30000	104749	1044	no	5133	0	151	-
50000	174407	1746	no	23073	0	223	-
100000	350251	3348	no	90654	0	478	-

Table 6.3: *Measurements on transitive closure algorithms - MinMax Non-DAG*

is high probability that a newly inserted / deleted edge will not affect the structure of the reduced graph.

- Fixpoint computation dominates the runtime of DRed and Counting (and with this, implicitly IncSCC too). On Table 6.4 a portion of a YourKit [11] CPU profiling snapshot can be seen for the Counting algorithm on a MinMax DAG graph with 10000 vertices, after initialization and a series of graph manipulations. The transitive closure relation contains more than 6 million tuples (stored explicitly) and the relation manipulation dominates the whole runtime. *Own time* means the time spent in the given method, also expressed as the percentage of the total runtime. Note that, the increased runtime is due to the detailed method call instrumentation.

- Memory consumption: Table 6.5 presents a portion of a YourKit Memory snapshot for the IncSCC algorithm run on the MinMax non-DAG with 100000 vertices. The objects that are reachable from GC roots via strong references take 60 MB of heap space. Shallow size of an object is the amount of memory allocated to store the object itself, not taking into account the referenced objects. Retained size of an object is its shallow size plus the shallow sizes of the objects that are accessible, directly or indirectly, only from this object. This value is presented in the table for the given classes.

Name	Time (ms)	Own time (ms)	Invocation count
java.util.HashMap.put(Object, Object)	41931	41931 (21 %)	25536240
java.util.HashMap.get(Object)	33008	33008 (16 %)	25718235
.TcRelation.put(Object, Object)	76201	24941 (12 %)	12822578
java.util.HashSet.add(Object)	20918	20918 (10 %)	12822578
.CountingTcRelation.addTuple(Obj, Obj, int)	82627	16563 (8 %)	6353402
java.util.HashMap.containsKey(Object)	15661	15661 (8 %)	12657477
java.util.HashMap$KeyIterator.next()	14413	14413 (7 %)	20169748

Table 6.4: *Yourkit profile results - hot spots*

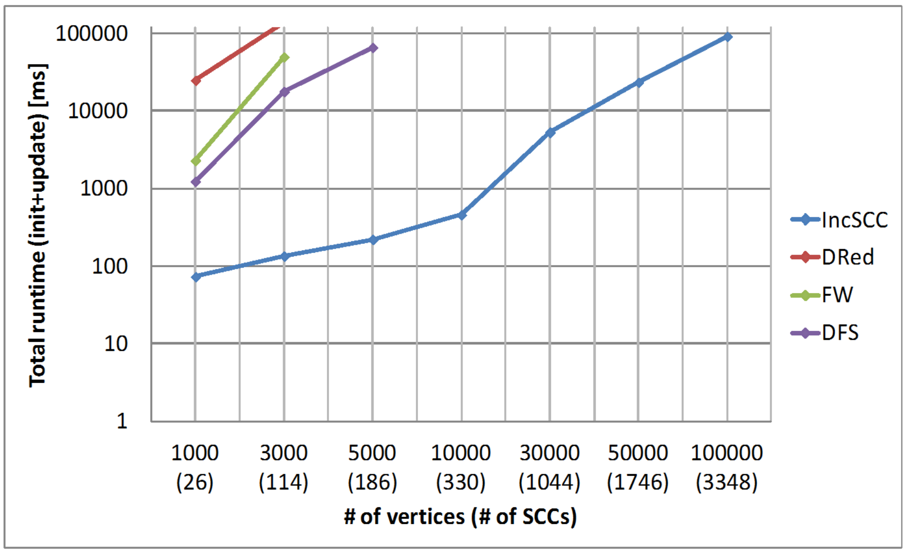


Figure 6.1: *Measurements on transitive closure algorithms - scaling on Min-Max Non-DAG*

Class Name	Shallow Size (bytes)	Retained Size (bytes)
java.util.HashMap$Entry[]	3964880	59195184
java.util.HashMap	464408	59194408
java.util.HashMap$Entry	19800096	55870832
....measurements.MinMaxGraph	40	33860280
java.util.ArrayList	5729952	28863176
java.lang.Object[]	13507312	23194104
java.lang.Integer	12782320	12782320
....graphimpl.Graph	48	10867912
....incscc.IncSCCAlg	48	8652064
....incscc.UnionFind	24	6910032
....incscc.UnionFindNodeProperty	2400000	2400000

Table 6.5: *Yourkit profile results - memory snapshot*

6.2 Benchmarking of the Stochastic simulation framework

6.2.1 Measurement scenario

To find out the performance differences between various pattern matching algorithms for transitive closure, I ran a series of measurements on simplified stochastic model simulation processes (see Section 5.3 for more information on the framework). The simulation aims to analyse the probability of the network being (fully) connected (so that each client can communicate with every other one, through their direct super node connections and the transitive overlay links between super nodes). The connectivity measure was registered through the *P_ConnectedOverlayPairs probe* which reports the size of the match set of the *transitiveClosureOfLinked* pattern (Listing 3.4) after each simulation step.

A simulation run consisted of 2000 steps (rule applications). Along with the total execution time of the run, the wall times for various sub-phases were registered - such as the time it took to propagate updates through the transitive closure RETE node - using code instrumentation.

The experiments were carried out with three different strategies of evaluating graph patterns and transitive closure:

1. Local search pattern matching as implemented in VIATRA2.
2. RETE-based incremental matching with the DRed algorithm for transitive closure.
3. RETE with IncSCC for transitive closure.

The measurement environment is characterized with the following metrics:

- NSCC: number of isolated but strongly connected components in the initial configuration.
- NSN: number of super nodes per SCC in the initial configuration.
- NCL: number of client nodes connected to every super node in an SCC.

A model with $NSCC = 2, NSN = 2, NCL = 3$ is visualized on Figure 6.2. The number of model elements is $2 \times 2 \times (3 + 1) = 16$.

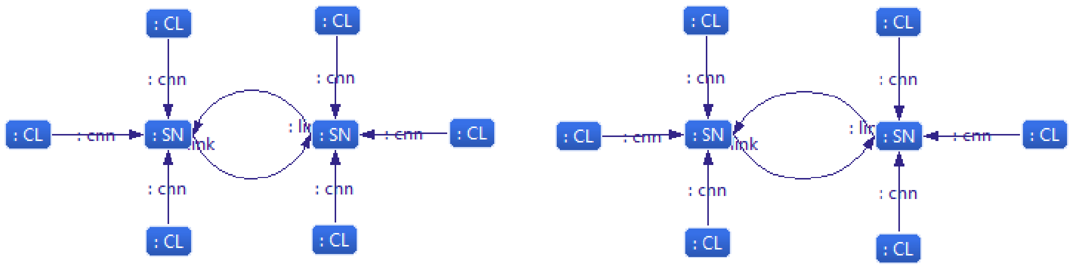


Figure 6.2: *Stochastic simulation model visualization*

The performance of these solutions were investigated with two series of experiments. The first series considered various model structures induced by different probability weights of the *addLink* rule. It was run on an initial model of 2000 vertices in 20 (NSCC) isolated components, each containing 10 super nodes (NSN) and 9 client nodes (NCL) per super node. The second series settled on a fixed value 0,005 of *addLink* weight, and considered increasingly larger model sizes (from 1000 to 10000 nodes), initially divided into 10 to 100 (NSCC) isolated components similarly to the first series.

6.2.2 Results and analysis

Table 6.6 shows the results of the first experiment series. For each value of *addLink*, the table displays (i) the values of the probes (as well as the number of strongly connected components) averaged over an entire simulation run; (ii) for each of the three solutions the total execution time (in milliseconds) and, in case of the incremental algorithms, (iii) the time spent initializing and updating the transitive closure node (expressed as a percentage of total time).

Table 6.7 shows the results of the second experiment series. Here, the rate of the *addLink* rule is set to 0,005 and the model size is continuously increasing.

addLink	Graph properties (avg.)			Local search	DRed		IncSCC	
	# SCCs	Net size	Overlay connectivity	Total time	Total time	Tc time [%]	Total time	Tc time [%]
0,005	15,30	428,55	107,56	54833	17236	13,9	18833	1,3
0,01	15,22	420,44	111,72	51681	16875	14,8	16461	1,9
0,05	16,74	417,13	133,70	50533	22295	15,3	19228	1,8
0,1	18,65	415,53	149,70	55562	21297	17,4	18736	1,9
1	11,45	459,25	1663,45	151913	47211	59,9	20707	3,3
2	5,27	509,01	4543,02	309476	67718	70,5	21008	3,9
5	2,63	594,35	7480,20	579774	97755	78,2	26643	3,6

Table 6.6: *Stochastic simulation measurement results - varying addLink rate*

NSCC	Graph properties (avg.)			Local s.	DRed		IncSCC	
	# SCCs	Net size	Overlay connectivity	Total time	Total time	Tc time [%]	Total time	Tc time [%]
10	7,36	1000	30,51	19812	4963	24,1	6150	3,6
20	15,26	2000	106,80	48672	19441	14,1	17571	1,7
30	25,77	3000	223,89	102071	53928	10,7	43353	0,8
40	33,44	4000	379,97	191927	97360	11,4	96684	0,3
50	43,09	5000	625,27	314261	177918	9,4	173031	0,4
60	53,72	6000	903,19	492659	300378	9,0	309973	0,2
70	61,91	7000	1206,53	713140	434165	8,9	456706	0,2
80	74,12	8000	1548,99	930715	633987	7,9	647776	0,1
90	84,13	9000	2023,90	1283557	894999	7,4	892698	0,1
100	94,69	10000	2549,10	1770635	1209676	7,1	1240087	0,1

Table 6.7: *Stochastic simulation measurement results - fixed addLink rate*

The first series of experiments reveals that as the probability weight of *addLink* increases, the frequent rule executions make the graph more and more connected. DRed performance significantly degrades for more connected graphs (e.g. as larger and larger number of pairs have to be rederived after deletion), to the point that transitive closure maintenance dominates the total execution time of the simulation. IncSCC however takes advantage of SCCs and runs efficiently in all cases, having a negligible impact on the overall runtime of the simulation and RETE maintenance. Local search in VIATRA2 is orders of magnitudes slower than either of the incremental approaches.

The second measurement series verifies that IncSCC can be efficiently applied in this scenario too, as there are only a small number of SCCs compared to the model size (for example 94 SCCs in a graph with 10000 vertices) thus the maintenance of the reduced graph does not require a lot of computation. It also demonstrates that IncSCC has a better complexity characteristic on large models than DRed, while both scale significantly better than the Local Search solution. The simulation time with DRed and IncSCC peaks at $\sim$ 1200 seconds and with Local Search it is increased with an additional $\sim$ 550 seconds.

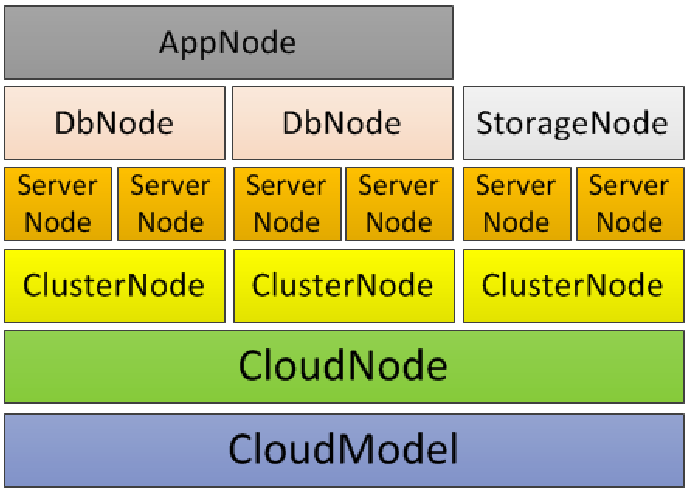


Figure 6.3: *POWER_ON_SMALL DSE problem goal*

6.3 Benchmarking of the Design Space Exploration component

6.3.1 Measurement scenario

The performance evaluation of the Design Space Exploration component (Section 5.2) was carried out with three different goals. These expected solutions differ in the trajectory length and are characterized by the expected configuration/structure of the system:

- **CLUSTERED_DB_SMALL**: the expected solution must have two *DbNode* instances installed. The optimal trajectory consists of 9 rule applications and one of the shortest solutions is: $addCloudNode \rightarrow addClusterToCloud \rightarrow addServer-ToSocket \rightarrow addServerToSocket \rightarrow addDBToServer \rightarrow addClusterToCloud \rightarrow addServerToSocket \rightarrow addServerToSocket \rightarrow addDBToServer$.

- **CLUSTERED_DB_BIG**: the expected solution must have three *DbNode* instances installed. The optimal trajectory consists of 13 rule applications.

- **POWER_ON_SMALL**: the expected solution must have an *AppNode* instance and a *StorageNode* instance installed. The optimal trajectory consists of 14 rule applications and the resulting model stack is shown on Figure 6.3. Note that, the *CloudModel* instance (on the bottom), already exists in the initial model.

The main goal of the measurements was to get an overview about the performance characteristic of the simple EMF-IncQuery-backed solution opposed to the VIATRA2-based one. Both engines used a depth-first search like traversal algorithm and were executed without guidance and also with priorities. Priorities were assigned to rules in an order that leads to a solution under the smallest number of steps possible. According to Figure 6.3, this means that the *addCloudNode* rule was assigned 5, *addClusterToCloud* 4, and so on, ending with *addAppToDB* being assigned priority value 1 (the one with the highest priority).

Measured parameters (averaged over a series of explorations):

- The **number of solutions found** during the exploration.
- **Length of the shortest solution found** and whether it is the optimal one.
- The **number of visited states** for the given exploration (a limit of 500000 states was used for each run).
- **Total runtime** of the exploration (in milliseconds); if the optimal solution is not found during the exploration it is the time required to explore 500000 states in the given scenario.

6.3.2 Results and analysis

Table 6.8 presents the measurement results for the three scenarios. Key observations:

- It can be seen that for all scenarios, the number of solutions found and the length of the shortest solution found is the same for the two solutions with slight differences.
- Exploration time is about $1,5$ times longer for the EMF-IncQuery-backed solution than it is for the VIATRA2-based one in the CLUSTERED_DB_SMALL case, and about 2 times longer in the other two cases. Based on YourKit profiling, it is clear that the transaction handling of the VIATRA2 framework is much more efficient than the one provided by EMF (based on the times spent in the *lock/unlock/commit* calls during transaction handling). Additionally, change propagation in EMF about model manipulations involves a lot of operations with collections and those increase the runtime of the exploration.
- Using priorities is only beneficial for smaller models, as with increasing state space size, a random (unguided) exploration strategy finds more solutions and the shortest one is also closer to the optimal trajectory. This is due to the fact that priorities strictly set the order of rule applications opposed to random firings. A good example for this is the CLUSTERED_DB_BIG scenario, where the unguided EMF-IncQuery strategy finds 5 solutions with the shortest being 14 in length, while the strategy with priorities only finds one solution with a length of 16.
- Without depth limit or global constraints on the number of rule applications the exploration is not feasible as there are certain rules that can always be applied and this would result an infinite state space.

The EMF-IncQuery based solution is only a proof of concept at the current state of development which needs further optimization and incorporation of additional techniques in order to reduce the state space and cut off unnecessary branches in the search tree. A more efficient transaction handling or just simply an efficient operation redo functionality (as there are no parallel access present which would require transactions) in EMF models would result better performance characteristics.

The measurements were carried out with a version of the VIATRA2-based solution that is of equal capabilities (guidance with priorities or no guidance at all based on a depth-first

Problem	Length of optimal solution	Exploration strategy	Number of sol. found	Length of shortest sol. found	# of visited states	Runtime [ms]
DB_SMALL	9	VIATRA no guid.	5	9 (optimal)	435217	42968
		EIQ no guid.	5	9 (optimal)	421588	77056
		VIATRA priority	3	9 (optimal)	206864	21056
		EIQ priority	3	9 (optimal)	207893	35141
DB_BIG	13	VIATRA no guid.	4	15	500000	50612
		EIQ no guid.	5	14	500000	99244
		VIATRA priority	1	16	500000	59176
		EIQ priority	1	16	500000	100521
POWER_ON	14	VIATRA no guid.	3	16	500000	51085
		EIQ no guid.	3	15	500000	102652
		VIATRA priority	1	16	500000	55151
		EIQ priority	1	16	500000	110294

Table 6.8: *Measurements results on Design Space Exploration*

search like algorithm). However, in [24] a more sophisticated solution is presented which utilizes additional techniques like occurrence vectors and rule-dependency.

Chapter 7

Conclusions

Over the past few years the concepts of model driven software development have proved to be an efficient paradigm when it comes to design or implement large distributed and safety critical systems. Although numerous tools are present for the various modeling purposes and a wide range of research effort has been put into the underlying techniques, it is still very expensive for a company to adapt the toolset of MDSD. The Eclipse Modeling Framework tries to overcome these difficulties by defining a common platform for model creation, management and code generation and both the open-source and commercial software industry have produced several novel tools to extend these functionalities.

Among these tools, EMF-IncQuery aims to provide an efficient declarative framework to define queries over EMF models. The thesis focuses on this framework and investigates the special requirements of both applied and research use-cases. These investigations revealed that (i) the computation of transitive reachability over models is widely used in many scenarios and (ii) several applications would take advantage of an extension which allows the definition of model transformation rules over the EMF models. Usability and performance of these solutions were investigated with three case studies from different modeling scenarios.

Results of the thesis work

Transitive closure related results

During the first part of the thesis work I have experimented with transitive closure algorithms and their applications. These involved several static algorithms (Floyd-Warshall, graph traversal and topological sorting based solutions) and also fully-dynamic ones, like Counting, DRed and King. The performance characteristics of the various prototypical implementations were investigated through synthetic experiments and based on the experience gained from these measurements I have adapted an interesting approach too. This solution (named IncSCC) is based on the incremental maintenance of the strongly connected components of the graph and it came out to be the most efficient out of the ones that I have experimented with.

This part of the work was presented at the *Sixth International Conference on Graph*

Transformation in a publication entitled *Incremental pattern matching for the efficient computation of transitive closure* [14]. The IncSCC algorithm was integrated into the EMF-IncQuery Base library which is now able to provide efficient computation of transitive closure too. Additionally, transitive closure as a language element is now present in the pattern language of EMF-IncQuery, which required to integrate the algorithm into the RETE net in order to interact with it when evaluating model queries.

Rule execution engine related results

To support the definition of transformation rules in the EMF-IncQuery context, I have designed a robust component called the Rule Engine. With this engine one can create model transformation rules over an EMF model by defining the precondition of the rule as an EMF-IncQuery pattern and the required steps as statements written in Java. The Rule Engine provides similar functionality like a rule-based expert system. It provides the so called Agenda which is a component responsible for maintaining the activations of the defined rules. Optionally, these activations can be automatically fired by the Trigger Engine, without the need to write any client code.

Case studies

The work included the evaluation of three case studies which use either the language extension, the Rule- and Trigger Engine or both of them:

- Stochastic simulation of P2P network systems: the VIATRA2-based approach was extended by the introduction of transitive closure into the pattern language. This allowed to define some of the steps of a simulation scenario more efficiently opposed to the prior solutions offered by the VIATRA2 framework.

- Validation Framework of the EMF-IncQuery tool: it is now possible to define complex structural constraints over EMF models with transitive closure support. Moreover, one can initialize the validation from the user interface and the Rule- and Trigger Engine take care of the automatic creation, update and disposal of problem markers in the Problems View of Eclipse.

- Design Space Exploration component: a proof of concept adaptation was created from the VIATRA2-based solution. The component is backed by the EMF-IncQuery framework and uses the Rule Engine to keep track of the applicable activations.

During the thesis work I was a member of the EMF-IncQuery team and the project became an official subproject under the Eclipse Modeling Framework Technology (EMFT) Container Project (November 2012). Both the transitive closure extension and the Rule (& Trigger) Engine were integrated into the project. Note that, the implementations and class diagrams presented in the thesis reflect the state of development as it was in November of 2012.

Future work

Model transformation extension

Extending the pattern language of EMF-IncQuery with model transformation constructs would be beneficial for several reasons:

- The construction of transformation rules would be much easier opposed to the solution when the user has to write the postcondition in Java.
- With a well-defined language extension the editor would be automatically capable of checking the syntax of the transformation rules.
- One may restrict the actions that can be expressed with these constructs as the current implementation in the Rule Engine allows arbitrary Java code to be used in the rule postcondition.

Further development of transitive closure algorithms

A prospective follow-up of the started work would be to investigate parallel transitive closure algorithms to (i) achieve further speed-up during the relation maintenance and (ii) to be able to handle even larger models. There are several existing multithreaded approaches but it still remains a hard problem because it is easily possible that the overhead introduced by the parallel algorithm represses the expected characteristics. This has already happened with my attempt to create a divide-and-conquer like transitive closure algorithm with the Fork/Join Framework introduced in Java SE 7.

Stochastic simulation framework adaptation

A novel application of EMF-IncQuery would be to apply it in model-based simulations. The simulation of P2P networks was introduced in a VIATRA2-based approach and adapting that solution would result that the designer would be able to use the powerful pattern language of EMF-IncQuery. However, this approach also needs to extend the language with a model transformation component first.

Bibliography

[1] CLIPS project. http://clipsrules.sourceforge.net.

[2] Data Binding Overview in the .NET Framework. http://msdn.microsoft.com/en-us/library/ms752347.aspx.

[3] EMF-IncQuery Base documentation. http://viatra.inf.mit.bme.hu/incquery/documentation/base.

[4] EMF-IncQuery framework. http://viatra.inf.mit.bme.hu/incquery.

[5] JBoss Drools Documentation. http://www.jboss.org/drools/documentation.

[6] SSJ - Stochastic Simulation in Java. http://www.iro.umontreal.ca/~simardr/ssj/indexe.html.

[7] The Eclipse Foundation. http://www.eclipse.org/.

[8] VIATRA2 framework. http://viatra.inf.mit.bme.hu/viatra2.

[9] Xtend project. http://www.eclipse.org/xtend.

[10] Xtext project. http://www.eclipse.org/Xtext.

[11] YourKit Profiler Tool. http://www.yourkit.com/overview/index.jsp.

[12] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928. 10.1007/BF01459088.

[13] S. A. Baset and H. G. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11, April 2006.

[14] Gábor Bergmann, István Ráth, Tamás Szabó, Paolo Torrini, and Dániel Varró. Incremental pattern matching for the efficient computation of transitive closure. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin / Heidelberg, 2012.

[15] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, Huang P., S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation, 2000.

[16] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Kluwer, 2008.

[17] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[18] Camil Demetrescu and Giuseppe F. Italiano. Dynamic shortest paths and transitive closure: algorithmic techniques and data structures. *J. Discr. Algor*, 4:353–383, 2006.

[19] P. Erdős and A Rényi. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960.

[20] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

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

[22] Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM JOURNAL OF EXPERIMENTAL ALGORITHMICS*, 6:2001, 2000.

[23] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *IN: PROC. OF THE INTERNATIONAL CONF. ON MANAGEMENT OF DATA, ACM*, pages 157–166, 1993.

[24] Abel Hegedus, Akos Horvath, Istvan Rath, and Daniel Varro. A model-driven framework for guided design space exploration. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 173–182, Washington, DC, USA, 2011. IEEE Computer Society.

[25] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[26] International Electrotechnical Comission. *IEC 61508: Functional safety in electrical / electronic / programmable electronic safety-related systems*, 2010.

[27] Ajab Khan, Reiko Heckel, Paolo Torrini, and István Ráth. Model-based stochastic simulation of P2P VoIP using graph transformation. In *Proceedings of the 17th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, 2010.

[28] Ajab Khan, Paolo Torrini, and Reiko Heckel. Model-based simulation of VoIP network reconfigurations using graph transformation systems. In Andrea Corradini and

Emilio Tuosto, editors, *Intl. Conf. on Graph Transformation (ICGT) 2008 - Doctoral Symposium*, volume 16 of *Electronic Communications of the EASST*, 2009. http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/26.

[29] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, 1999.*, pages 81–89. IEEE, 1999.

[30] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05, pages 139–150, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] Johannes A. La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Graph-Theoretic Concepts in Computer Science, International Workshop, WG '87*, volume 314 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 1988.

[32] Istvan Rath. *Event-driven Model Transformations in Domain-specific Modeling Languages*. PhD thesis, Budapest University of Technology and Economics, Budapest, Hungary, March 2011.

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

[34] A. Rensink. The groove simulator: A tool for state space generation. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[35] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[36] The Eclipse Project. Eclipse modeling framework. `http://www.eclipse.org/emf`.

[37] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to algorithms*, chapter 21 - Data structures for Disjoint Sets, pages 498–524. MIT Press and McGraw-Hill, second edition edition, 2001. ISBN 0-262-03293-7.

[38] Paolo Torrini, Reiko Heckel, and István Ráth. Stochastic simulation of graph transformation systems. In David Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 154–157. Springer Berlin / Heidelberg, 2010. DOI: 10.1007/978-3-642-12029-9_11.

[39] Bartha T. Darvas D. Szabó T. Jámbor A. Vörös, A. and Á. Horváth. Parallel saturation based model checking. In IEEE Computer Society, editor, *The 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, pages 94–101, 2011. http://dx.doi.org/10.1109/ISPDC.2011.23.

[40] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. Slicetime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 19–33, Berkeley, CA, USA, 2011. USENIX Association.

[41] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

Appendix

A.1 Tarjan's algorithm

An iterative version of Tarjan's algorithm written in Java (SE 6).

Listing A.1.1: Tarjan's algorithm

```
public static <V> SCCResult<V> computeSCC(IGraphDataSource<V> g) {
  int index = 0;
  Set<Set<V>> ret = new HashSet<Set<V>>();
  Map<V, SCCProperty> nodeMap = new HashMap<V, SCCProperty>();
  Map<V, List<V>> targetNodeMap = new HashMap<V, List<V>>();
  Map<V, Set<V>> notVisitedMap = new HashMap<V, Set<V>>();

  //stores the nodes during the traversal
  Stack<V> nodeStack = new Stack<V>();
  //stores the nodes that are in the same strongly connected component
  Stack<V> sccStack = new Stack<V>();

  boolean sink = false, finishedTraversal = true;

  //initialize all nodes with 0 index and 0 lowlink
  Set<V> allNodes = g.getAllNodes();
  for (V n : allNodes) {
    nodeMap.put(n, new SCCProperty(0, 0));
  }

  for (V n : allNodes) {
    if (nodeMap.get(n).getIndex() == 0) {

      index++;
      sccStack.push(n);
      nodeMap.get(n).setIndex(index);
      nodeMap.get(n).setLowlink(index);

      notVisitedMap.put(n, new HashSet<V>());

      List<V> targetNodes = g.getTargetNodes(n);

      if (targetNodes != null) {
        targetNodeMap.put(n, new ArrayList<V>(targetNodes));
      }

      nodeStack.push(n);

      while(!nodeStack.isEmpty()) {

        V currentNode = nodeStack.peek();
        sink = false; finishedTraversal = false;
        SCCProperty prop = nodeMap.get(currentNode);

        //if node is not visited yet
        if (nodeMap.get(currentNode).getIndex() == 0) {
          index++;
          sccStack.push(currentNode);
          prop.setIndex(index);
          prop.setLowlink(index);

          notVisitedMap.put(currentNode, new HashSet<V>());

          //storing the target nodes of the actual node
          if (g.getTargetNodes(currentNode) != null) {
            targetNodeMap.put(currentNode,
              new ArrayList<V>(g.getTargetNodes(currentNode)));
          }
        }

        if (targetNodeMap.get(currentNode) != null) {

          if (targetNodeMap.get(currentNode).size() == 0) {
            targetNodeMap.remove(currentNode);

            //remove node from stack, the exploration of its children has finished
            nodeStack.pop();

            List<V> targets = g.getTargetNodes(currentNode);
            if (targets != null) {
              for (V targetNode : g.getTargetNodes(currentNode)) {
                if (notVisitedMap.get(currentNode).contains(targetNode)) {
                  prop.setLowlink(Math.min(prop.getLowlink(),
                    nodeMap.get(targetNode).getLowlink()));
                } else if (sccStack.contains(targetNode)) {
                  prop.setLowlink(Math.min(prop.getLowlink(),
                    nodeMap.get(targetNode).getIndex()));
                }
              }
            }

            finishedTraversal = true;
          }
          else {
            //push next node to stack
            V targetNode = targetNodeMap.get(currentNode).remove(0);
            //if _t has not yet been visited
            if (nodeMap.get(targetNode).getIndex() == 0) {
              notVisitedMap.get(currentNode).add(targetNode);
              nodeStack.add(targetNode);
            }
          }
        }
        //if _node has no target nodes
        else {
          nodeStack.pop();
          sink = true;
        }

        //create scc if node is a sink or an scc has been found
        if ((sink || finishedTraversal) &&
        (prop.getLowlink() == prop.getIndex())) {
        Set<V> sc = new HashSet<V>();
        V targetNode = null;

        do {
          targetNode = sccStack.pop();
          sc.add(targetNode);
        } while (!targetNode.equals(currentNode));

        ret.add(sc);
      }
    }
  }
  }
  return new SCCResult<V>(ret, g);
}
```

A.2 IncSCC implementation details

Implementation of the *Insert* operation in the IncSCC algorithm (Java SE 6).

Listing A.2.1: IncSCC algorithm update operation after edge insertion

```
public void edgeInserted(V source, V target) {
  V sourceRoot = sccs.find(source);
  V targetRoot = sccs.find(target);

  //Different SCC
  if (!sourceRoot.equals(targetRoot)) {

    //source is reachable from target?
    if (counting.isReachable(targetRoot, sourceRoot)) {

      Set<V> predecessorRoots = counting.getAllReachableSources(sourceRoot);
      Set<V> successorRoots = counting.getAllReachableTargets(targetRoot);

      //1. intersection
      Set<V> isectRoots = CollectionHelper.
        intersection(predecessorRoots, successorRoots);
      isectRoots.add(sourceRoot);
      isectRoots.add(targetRoot);

      //notifications must be issued before Union-Find modifications
      if (observers.size() > 0) {
        Set<V> sourceSCCs = new HashSet<V>();
        Set<V> targetSCCs = new HashSet<V>();

        sourceSCCs.add(sourceRoot);
        sourceSCCs.addAll(predecessorRoots);
        targetSCCs.add(targetRoot);
        targetSCCs.addAll(successorRoots);

        //tracing back to actual nodes
        for (V sourceSCC : sourceSCCs) {
          for (V targetSCC : CollectionHelper.
            difference(targetSCCs, counting.getAllReachableTargets(sourceSCC))) {
            boolean needsNotification = false;
      if (sourceSCC.equals(targetSCC) && sccs.setMap.get(sourceSCC).size()
        == 1 && graphHelper.getEdgeCount(sourceSCC) == 0) {
        needsNotification = true;
      }
      else if (!sourceSCC.equals(targetSCC)) {
        needsNotification = true;
      }
      //if self loop is already present omit the notification
      if (needsNotification) {
        notifyTcObservers(sccs.setMap.get(sourceSCC),
          sccs.setMap.get(targetSCC), Direction.INSERT);
      }
    }
  }
}

//2. delete edges, nodes
List<V> sources = new ArrayList<V>();
List<V> targets = new ArrayList<V>();

for (V r : isectRoots) {
  List<V> _srcList = graphHelper.getSourceSCCsOfSCC(r);
  List<V> _trgList = graphHelper.getTargetSCCsOfSCC(r);

  for (V _source : _srcList) {
    if (!_source.equals(r)) {
      reducedGraph.deleteEdge(_source, r);
    }
  }

  for (V _target : _trgList) {
    if (!isectRoots.contains(_target) && !r.equals(_target)) {
      reducedGraph.deleteEdge(r, _target);
    }
  }

  sources.addAll(_srcList);
  targets.addAll(_trgList);
}

for (V r : isectRoots) {
  reducedGraph.deleteNode(r);
}

//3. union
Iterator<V> iterator = isectRoots.iterator();
V newRoot = iterator.next();
while (iterator.hasNext()) {
  newRoot = sccs.union(newRoot, iterator.next());
}

//4. add new node
reducedGraph.insertNode(newRoot);

//5. add edges
Set<V> containedNodes = sccs.setMap.get(newRoot);

for (V _s : sources) {
  if (!containedNodes.contains(_s) && !_s.equals(newRoot)) {
    reducedGraph.insertEdge(_s, newRoot);
```

```
        }
      }
      for (V _t : targets) {
        if (!containedNodes.contains(_t) && !_t.equals(newRoot)) {
          reducedGraph.insertEdge(newRoot, _t);
        }
      }
    }
    else {
      if (observers.size() > 0 && graphHelper.getEdgeCount(source, target) == 1) {
        counting.attachObserver(countingListener);
      }
      reducedGraph.insertEdge(sourceRoot, targetRoot);
      counting.detachObserver(countingListener);
    }
  }
  else {
    //Notifications about self-loops
    if (observers.size() > 0 && sccs.setMap.get(sourceRoot).size() == 1 &&
      graphHelper.getEdgeCount(source, target) == 1) {
      notifyTcObservers(source, source, Direction.INSERT);
    }
  }
}
```

Implementation of the *Delete* operation in the IncSCC algorithm (Java SE 6).

Listing A.2.2: IncSCC algorithm update operation after edge deletion

```
public void edgeDeleted(V source, V target) {
  V sourceRoot = sccs.find(source);
  V targetRoot = sccs.find(target);

  if (!sourceRoot.equals(targetRoot)) {
    if (observers.size() > 0 && graphHelper.getEdgeCount(source, target) == 0) {
      counting.attachObserver(countingListener);
    }
    reducedGraph.deleteEdge(sourceRoot, targetRoot);
    counting.detachObserver(countingListener);
  }
  else {
    //get the graph for the scc whose root is sourceRoot
    Graph<V> g = graphHelper.getGraphOfSCC(sourceRoot);

    //if source is not reachable from target anymore
    if (!BFS.isReachable(source, target, g)) {
      List<V> reachableSources = null;
      List<V> reachableTargets = null;

      SCCResult<V> _newSccs = SCC.computeSCC(g);

      //delete scc node (and with its edges too)
      reachableSources = reducedGraphIndexer.getSourceNodes(sourceRoot);
      reachableTargets = reducedGraphIndexer.getTargetNodes(sourceRoot);

      if (reachableSources != null) {
        Set<V> tmp = new HashSet<V>(reachableSources);
        for (V s : tmp) {
          reducedGraph.deleteEdge(s, sourceRoot);
        }
    }
    if (reachableTargets != null) {
      Set<V> tmp = new HashSet<V>(reachableTargets);
      for (V t : tmp) {
        reducedGraph.deleteEdge(sourceRoot, t);
      }
    }
    sccs.deleteSet(sourceRoot);
    reducedGraph.deleteNode(sourceRoot);

    Set<Set<V>> newSccs = _newSccs.getSccs();
    Set<V> roots = new HashSet<V>();

    //add new nodes and edges to the reduced graph
    for (Set<V> _scc : newSccs) {
      V newRoot = sccs.makeSet((V[]) _scc.toArray());
      reducedGraph.insertNode(newRoot);
      roots.add(newRoot);
    }
    for (V _root : roots) {
      List<V> sourceNodes = graphHelper.getSourceSCCsOfSCC(_root);
      List<V> targetNodes = graphHelper.getTargetSCCsOfSCC(_root);

      for (V _s : sourceNodes) {
        V _sR = sccs.find(_s);
        if (!_sR.equals(_root))
          reducedGraph.insertEdge(sccs.find(_s), _root);
      }
      for (V _t : targetNodes) {
        V _tR = sccs.find(_t);
        if (!roots.contains(_t) && !_tR.equals(_root))
          reducedGraph.insertEdge(_root, _tR);
      }
    }

    //Must be after the union-find modifications
    if (observers.size() > 0) {
      V newSourceRoot = sccs.find(source);
      V newTargetRoot = sccs.find(target);

      Set<V> sourceSCCs = counting.getAllReachableSources(newSourceRoot);
      sourceSCCs.add(newSourceRoot);

      Set<V> targetSCCs = counting.getAllReachableTargets(newTargetRoot);
      targetSCCs.add(newTargetRoot);

      for (V sourceSCC : sourceSCCs) {
        for (V targetSCC : CollectionHelper.
          difference(targetSCCs, counting.getAllReachableTargets(sourceSCC))) {
          boolean needsNotification = false;

          if (sourceSCC.equals(targetSCC) && sccs.setMap.get(sourceSCC).size()
            == 1 && graphHelper.getEdgeCount(sourceSCC) == 0) {
            needsNotification = true;
          }
          else if (!sourceSCC.equals(targetSCC)) {
            needsNotification = true;
          }
          //if self loop is already present omit the notification
          if (needsNotification) {
              notifyTcObservers(sccs.setMap.get(sourceSCC),
                sccs.setMap.get(targetSCC), Direction.DELETE);
            }
          }
        }
      }
    }
    else {
      //only handle self-loop notifications - sourceRoot equals to targetRoot
      if (observers.size() > 0 && sccs.setMap.get(sourceRoot).size() == 1 &&
        graphHelper.getEdgeCount(source, target) == 0) {
        notifyTcObservers(source, source, Direction.DELETE);
      }
    }
  }
}
```

A.3 DFS-based generation of the transitive closure relation

Listing A.3.1: DFS-based algorithm - transitive closure relation generation

```
private void deriveTc() {
        tc.clear();
        this.visited = new int[gds.getAllNodes().size()];
        nodeMap = new HashMap<V, Integer>();

        int j = 0;
        for (V n : gds.getAllNodes()) {
                nodeMap.put(n, j);
                j++;
        }

        for (V n : gds.getAllNodes()) {
                oneDFS(n, n);
                initVisitedArray();
        }
}

private void initVisitedArray() {
        for (int i=0;i<visited.length;i++) {
                visited[i] = 0;
        }
}

private void oneDFS(V act, V source) {
        if (!act.equals(source)) {
                tc.addTuple(source, act);
        }

        visited[nodeMap.get(act)] = 1;

        List<V> targets = gds.getTargetNodes(act);
        if (targets != null) {
                for (V t : targets)     {
                        if (visited[nodeMap.get(t)] == 0) {
                                oneDFS(t, source);
                        }
                }
        }
}
```

A.4 Stochastic simulation source codes

Listing A.4.1: Stochastic simulation - CreateClient rule

```
gtrule CreateClient () = {
        precondition pattern lhs() = {
                neg find discnn5();
        }
        action {
                let C = undef in new(CL(C) in DSM.model.amod1);
                println ("client created");
        }
}
```

Listing A.4.2: Stochastic simulation - DeleteClient rule

```
gtrule DeleteClient() = {
        precondition find disconnected(C)
        action {
                println ("disconnected client deleted: "+C);
                delete(C);
        }
}
```

Listing A.4.3: Stochastic simulation - ConnectClient rule

```
gtrule ConnectClient() = {
        precondition shareable pattern lhs(S,C) = {
                find disconnected(C); // disconnected client
                find SN_MinClients(S);
        }
        action {
                let I = undef in new(CL.cnn(I,C,S));
                println ("client connected to supernode: "+S);
        }
}
```

Listing A.4.4: Stochastic simulation - DisconnectClient rule

```
gtrule DisconnectClient() = {
        precondition shareable pattern lhr(S,C,E) = {
                SN(S);
                CL(C);
                CL.cnn(E,C,S);
        }
                action {
                delete(E);
                println ("client disconnected from supernode: "+S);
        }
}
```

Listing A.4.5: Stochastic simulation - UpgradeClient rule

```
gtrule UpgradeClient() = {
        precondition shareable pattern lhr_slim(S,C) = {
                find connectedToRichSN(C,S);
                find globalManyDisconneced();
        }
        action {
                let I1 = undef, S1 = undef in seq {
                        new(SN(S1) in DSM.model.amod1);
                        let Ix = 0, E = undef in
                        iterate choose C1 with find disconnected(C1) do seq {
                                new(CL.cnn(E,C1,S1));
                                update Ix = Ix+1;
                                if (Ix==3) fail;
                        }
                        new(SN.link(I1,S,S1));
                        println ("client "+C+" upgraded to supernode: "+S1);
                        delete(C);
                }
        }
}
```

Listing A.4.6: Stochastic simulation - DowngradeSuperNode rule

```
gtrule DowngradeSupernode() = {
        precondition find linked(S1,S2)
        action {
                let C = undef, I1 = undef in seq {
                        new(CL(C) in DSM.model.amod1);
                        new(CL.cnn(I1,C,S2));
                        println ("supernode "+S1+" downgraded to client "+C);
                        delete(S1);
                }
        }
}
```

Listing A.4.7: Stochastic simulation - AddLink rule

```
gtrule AddLink () = {
        precondition shareable pattern lhr(S1,S2) = {
                SN(S1);
                SN(S2);
                neg find linked(S1,S2);
        }
        action {
                let I1 = undef in new(SN.link(I1,S1,S2));
                println("redundant link added between supernodes "+S1+" and "+S2);
        }
}
```

Listing A.4.8: Stochastic simulation - P_NetworkSize rule

```
gtrule P_NetworkSize() = {
        precondition pattern lhs(N) = {
                Node(N);
        }
        action {
                skip;
        }
}
```

Listing A.4.9: Stochastic simulation - P_ConnectedOverlayPairs rule

```
gtrule P_ConnectedOverlayPairs_LS() = {
        precondition find transitiveClosureOfLinked(S1, S2)
        action {
                skip;
        }
}
```