Incremental View Maintenance for Deductive Graph Databases Using Generalized Discrimination Networks

Thomas Beyhl
Hasso Plattner Institute
at the University of Potsdam
Potsdam, Germany
thomas.beyhl@hpi.de

Holger Giese
Hasso Plattner Institute
at the University of Potsdam
Potsdam, Germany
holger.giese@hpi.de

Nowadays, graph databases are employed when relationships between entities are in the scope of database queries to avoid performance-critical join operations of relational databases. Graph queries are used to query and modify graphs stored in graph databases. Graph queries employ graph pattern matching that is NP-complete for subgraph isomorphism. Graph database views can be employed that keep ready answers in terms of precalculated graph pattern matches for often stated and complex graph queries to increase query performance. However, such graph database views must be kept consistent with the graphs stored in the graph database.

In this paper, we describe how to use incremental graph pattern matching as technique for maintaining graph database views. We present an incremental maintenance algorithm for graph database views, which works for imperatively and declaratively specified graph queries. The evaluation shows that our maintenance algorithm scales when the number of nodes and edges stored in the graph database increases. Furthermore, our evaluation shows that our approach can outperform existing approaches for the incremental maintenance of graph query results.

1 Introduction

Nowadays, graph databases are employed when relationships between entities are in the scope of graph database queries, because graph databases *can* outperform relational databases, due to the fact that in graph databases a traversal from one node to an adjacent node is a constant time operation [27], in contrast to relational databases that require performance critical join-operations to traverse from one node to an adjacent node. Graph database queries employ graph pattern matching that is NP-complete for subgraph isomorphism [13]. Thus, the graph query evaluation can be very time-consuming in worst-case scenarios. One possibility to improve the performance of graph query evaluation is to employ graph database views that keep ready precalculated answers for graph queries. These graph database views store all matches for graph patterns implemented by graph queries. However, such graph database views must be kept consistent with the graphs stored in the graph database. When graphs in the graph database change, old matches that do no satisfy a certain graph pattern anymore must be removed from the graph database view and new matches that satisfy a certain graph pattern must be added to the graph database view. Otherwise, graph queries lead to different graph query results when they make use of inconsistent graph database views. Furthermore, Harrison et al. [16] state that *"in a deductive database, the task of maintaining materialized views is challenging, because views can be defined using negation and recursion"*. We refer to graphs stored in graph databases as *base graphs* and to graph database views derived from base graphs as *view graphs*.

In this paper, we describe how to use incremental graph pattern matching as technique for maintaining view graphs. The main contributions are a) an enumeration mechanism that enables to store graph

© T. Beyhl & H. Giese

pattern matches in view graphs in a manner that graph pattern matches can be reused effectively and can be maintained efficiently, b) an incremental maintenance algorithm for view graphs that works for imperatively and declaratively specified graph queries and scales when the number of nodes and edges in base graphs increases, and c) an evaluation which shows that our approach *can* outperform existing approaches for incremental maintenance of view graphs.

Sec. 2 describes the state of the art in incremental graph pattern matching. Sec. 3 introduces the running example used throughout this paper. Sec. 4 describes our concept of view graphs. Sec. 5 describes how view graphs are created and maintained using incremental graph pattern matching. Sec. 6 evaluates the performance of our approach. Sec. 7 compares related work with our approach. Sec. 8 concludes our paper and outlines future work.

2 State of the Art

In practice, discrimination networks are used to maintain large collections of working memory elements that satisfy certain conditions. For example, discrimination networks are widely used in active database management systems [14], view maintenance for relational databases [15] and incremental graph pattern matching [2]. Several kinds of discrimination networks exist such as Rete networks [11], TREAT [21], and Gator networks [15]. All kinds of discrimination networks consist of network nodes and edges that constitute a directed acyclic network structure, but differ in the kinds of employed network nodes. In general, a network node performs a condition test to check whether working memory elements (e. g., tuples of relational data or nodes of graph data) satisfy a certain condition and, afterwards, store which working memory elements satisfy this condition. Network edges describe the exchange of working memory elements, which satisfy or dissatisfy conditions, between network nodes. For graph pattern matching, network nodes employ graph conditions [9] as condition tests and store graph pattern matches that satisfy these graph conditions. Furthermore, successor network nodes reuse graph pattern matches stored by predecessor network nodes for condition testing. Therefore, each network node enumerates graph pattern matches that satisfy certain conditions. From database perspective, these enumerations are considered as database views that keep ready graph pattern matches. When the graph data changes, the changes are propagated through the network to update graph pattern matches stored by network nodes by re-evaluating graph conditions only for changed, added, and deleted nodes and edges of base graphs.

The most generalized kind of discrimination network is the Gator network [15] that allows network nodes with an arbitrary number of inputs. Rete networks [11] and TREAT networks [21] are extreme examples of Gator networks due to the following restrictions. Rete networks are limited to network nodes with at most two inputs. Furthermore, Rete networks must be either left- or right-associative [18] and must not consist of re-convergent network nodes, otherwise the original Rete match algorithm produces duplicated or missing matches [18]. TREAT networks are restricted to network nodes with at most one input and do not allow intermediate nodes. In TREAT, join conditions are computed on demand.

Bunke et al. [4] transferred the concepts of original Rete networks to the efficient implementation of graph grammars by deriving the Rete network from the left-hand sides of graph grammar rules. Furthermore, EMF-IncQuery [2] employs Rete networks for incremental graph pattern matching in several application domains such as model queries over EMF models [1], derivation of features in EMF models [25], live model transformations [24], and synchronization of view models [7]. However, current incremental graph pattern matching approaches are limited to Rete networks and, thus, do not allow arbitrary network structures although optimized generalized network structures such as Gator networks [15] can outperform Rete networks in time and space as described by Hanson et al. [15] for relational databases.

To our best knowledge, no approach exists that employs Gator networks as most generalized kind of discrimination network for incremental graph pattern matching and view maintenance of graph databases. Discrimination networks do not support recursion due to their acyclic network structure.

In general, Rete and Gator networks have the same expressiveness for graph pattern matching. A formal proof is left for future work. In contrast to current approaches for incremental graph pattern matching, our approach employs Gator networks as generalized kind of discrimination network to enable graph databases users to steer the tradeoff between memory consumption and time required to update the state of the discrimination network. Computing optimal network structures is left for future work.

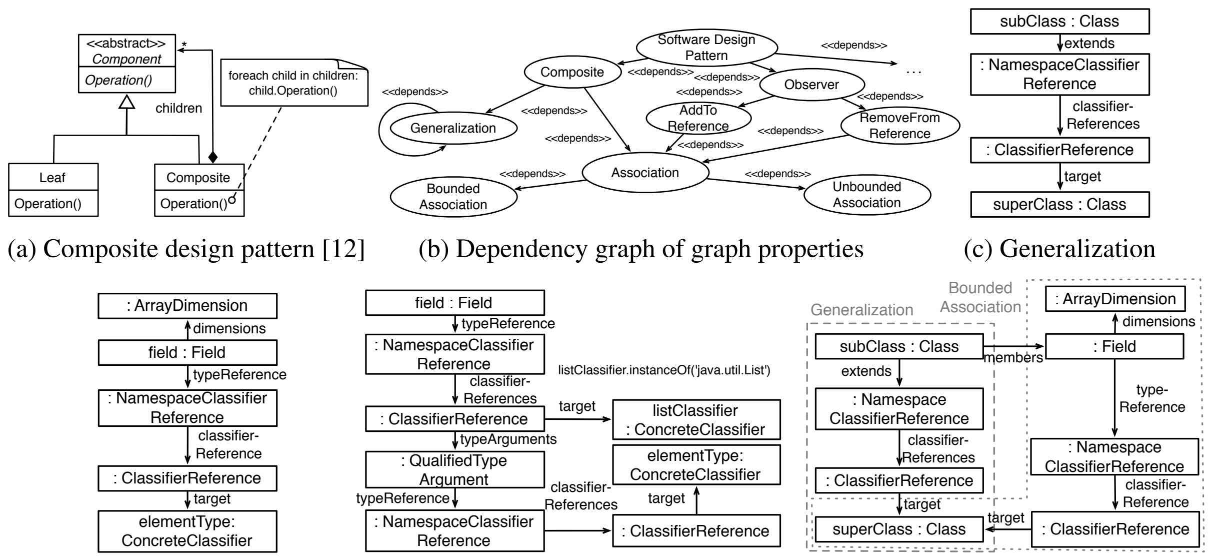
3 Running Example

In our running example, we search for graph pattern matches that describe employed software design patterns [12] in abstract syntax graphs (ASGs) of source code to analyze the evolution of software architectures. We aim for an incremental maintenance of these graph pattern matches for employed software design patterns when ASGs change. High-level graph properties such as the Composite design pattern can be recovered by detecting low-level graph properties such as generalizations and associations [22]. We employ graph pattern matching to detect such graph properties in terms of graph pattern matches in ASGs. Fig. 1a shows the Composite design pattern as UML class model. First, the Composite class must be a specialization of the Component class. Second, the Composite class must own an association with the Component class as target. Fig. 1b shows in a dependency graph that Generalizations and Associations have to be detected to recover Composite design patterns. Generalizations depend on Generalizations, because multiple generalizations can constitute a multi-level generalization. We distinguish Associations into BoundedAssociations and UnboundedAssociations, which employ data structures of immutable and mutable length to implement associations, respectively.

Fig. 1 shows the graph patterns used to detect Generalizations, BoundedAssociations, UnboundedAssociations, and Composite design patterns. The Generalization graph pattern (cf. Fig. 1c) describes that a subclass points via a namespace and classifier reference to a superclass. The BoundedAssociation graph pattern (cf. Fig. 1d) describes that a field consists of an array dimension and points via a namespace and classifier reference to a classifier of elements stored in an array data structure. The UnboundedAssociation graph pattern (cf. Fig. 1e) describes that a field points via a namespace and classifier reference to a classifier that is an instance of a list data structure and, additionally, points via a qualified type argument, namespace and classifier reference to a classifier of elements stored in a list data structure. The Composite graph pattern (cf. Fig. 1f) describes that a Generalization between a subclass and superclass exists (cf. dashed polygon) and that a field of a subclass is a BoundedAssociation that points via a namespace and classifier reference to a superclass of the generalization that describes the kind of the elements owned by the association (cf. dotted polygon). Note, the BoundedAssociation graph pattern can be replaced by the UnboundedAssociation graph pattern.

4 Views for Deductive Graph Databases

View graphs must store matches for graph patterns that are defined by graph database users. Thus, the question arises what exactly are view graphs and how to specify their content. View graphs must store graph pattern matches in a manner that a) graph pattern matches are stored memory-efficient and b) nodes with certain roles in graph pattern matches can be accessed effectively. Sec. 4.1 describes our notion of view graphs. Sec. 4.2 describes how our approach enables graph database users to define the content of view graphs. We present an exhaustive description in our technical report [3].



(a) Composite design pattern [12] (b) Dependency graph of graph properties (c) Generalization

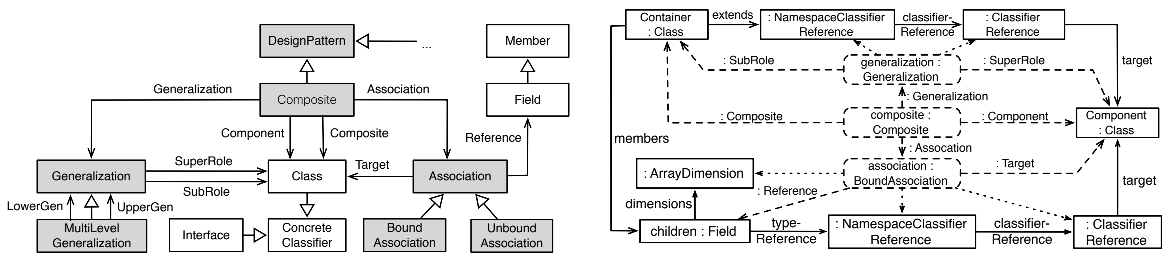
(d) BoundedAssociation (e) UnboundedAssociation (f) Composite

Figure 1: Overview of running example with graph patterns

4.1 Introduction of View Graphs

Base graphs and view graphs are typed graphs that must be conform to a type graph. Base graphs are graphs stored by graph databases to represent domain knowledge. View graphs are graphs that store typed nodes and edges which together mark matches of graph patterns. The node types in view graphs describe the kinds of graph pattern matches that are marked. The edge types in view graphs describe the roles of nodes in marked graph pattern matches. The edge types enable graph database users to effectively access nodes of graph pattern matches without the need to match them again to determine their role in graph pattern matches. Furthermore, each node that participates in a graph pattern match is either referenced by an edge or scope. When base graphs change in a way that nodes of view graphs do not mark valid graph pattern matches anymore, edges and scopes enable an efficient look-up of nodes in view graphs that must be revised. This look-up is performed by traversing edges and scopes in backward direction from changed nodes of base graphs to nodes of view graphs that own these edges and scopes.

According to our running example, Fig. 2a shows an excerpt of a type graph as UML class model that describes which kinds of nodes and edges exist in base graphs and view graphs. For example, the type graph describes that nodes of type Class, Interface and Field exist in base graphs as denoted by the white classes and that nodes of type Generalization, Association, and Composite exist in view graphs as denoted by gray classes. The type graph describes which edge types are used in view graphs to describe the roles of nodes in graph pattern matches. For example, the Generalization node type owns the SubRole and SuperRole edge types to describe that Class nodes are marked as super- and subclasses in matches of the Generalization graph pattern. Fig. 2b shows a view graph as UML object model that is an instance of the type graph in Fig. 2a. Solid rectangles denote typed nodes of base graphs. Solid lines denote edges between nodes of base graphs. Dashed rounded rectangles denote typed nodes in view graphs, which represent graph pattern matches. Dashed lines denote typed edges in view graphs, which describe the roles of nodes in marked graph pattern matches. Dotted lines denote scopes, which are edges that mark nodes without explicit roles in graph pattern matches. Fig. 2b shows a Generalization node that marks a graph pattern match for the generalization graph pattern. The SubRole and SuperRole edges owned by the Generalization node describe that the container class acts as subclass and the component class acts as superclass. The Generalization node references the namespace and classifier reference via scopes, because both nodes belong to the graph pattern match as well. Fig. 2b depicts a BoundedAssociation



(a) Base and view type graph as UML class model (b) Marked graph pattern matches as UML object model

Figure 2: View reference graph (left) and instantiated view graph (right)

node that marks a graph pattern match for the bounded association graph pattern. The Reference and Target edges owned by the BoundedAssociation node describe that the children field acts as reference and the component class acts as target type of the reference. The BoundedAssociation node owns scopes that reference the array dimension, namespace, and classifier reference, because they belong to the graph pattern match as well. Fig. 2b shows a Composite node that marks a graph pattern match for the Composite graph pattern. The Component and Composite edges owned by the Composite node describe that the component class acts as component and the container class acts as composite of the detected Composite design pattern. The Generalization and Association edges owned by the Composite node mark the reused matches of the Generalization and Association graph patterns.

4.2 Definition of View Graphs

Our approach enables to specify view modules that encapsulate graph transformation rules, which search and mark graph pattern matches by creating nodes and edges in view graphs. Each view module owns input connectors that describe which kinds of nodes are required by the hidden graph transformation rule. Each view module owns an output connector to describe which kind of node is created in view graphs by the hidden graph transformation rule to mark graph pattern matches. Our approach is independent from graph transformation languages, because view modules hide graph transformation rules.

Fig. 3 shows a generalized discrimination network that consists of the view modules Generalization, BoundedAssociation, UnboundedAssociation, and Composite. The Generalization view module describes that Class nodes and TypeReference (super type of NamespaceClassifierReference and ClassifierReference) nodes are required to produce nodes that mark matches of the Generalization graph pattern. The view modules BoundedAssociation and UnboundedAssociation create nodes of type Association that mark matches of the BoundedAssociation and UnboundedAssociation graph pattern. The Composite view module requires nodes of type Generalization and Association to create nodes in view graphs that mark matches of the Composite graph pattern.

Fig. 3 shows view modules that implement graph transformation rules for creating and maintaining view graphs that enumerate matches for generalizations, association, and composite graph patterns. When a graph transformation rule finds a graph pattern match, it marks the match by creating a node of a certain node type, edges of certain edge types, and scopes in view graphs to mark which nodes satisfy the graph pattern. Fig. 3 depicts graph pattern nodes as solid and dashed rectangles. Fig. 3 depicts graph pattern edges as solid and dashed lines. Solid rectangles and lines refer to nodes and edges in base graphs. Dashed rectangles and lines refer to nodes and edges in view graphs. Graph pattern nodes and edges can consist of create modifiers that are depicted as "++" (adapted from story diagrams [10]). These create modifiers describe which nodes and edges are created in view graphs when matches for graph patterns are found. Nodes and edges without create modifier depict the left-hand side of the graph transformation rule. Nodes and edges with and without create modifier depict the right-hand side of the graph transformation rule.

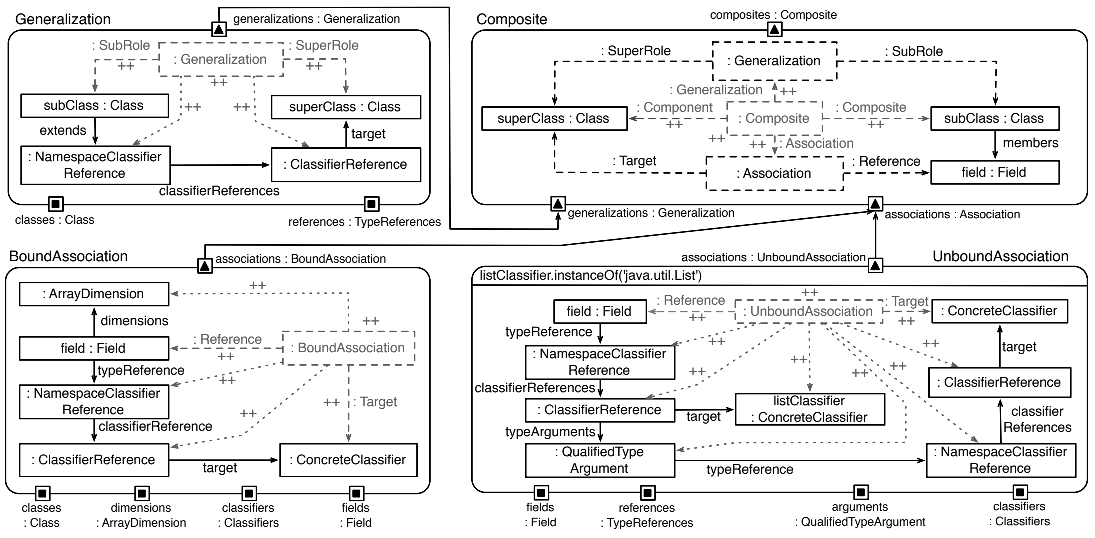


Figure 3: Generalized discrimination network of view modules

The graph patterns described in Fig. 1 define the left-hand side of the graph transformation rules depicted by Fig. 3. The right-hand side of the depicted graph transformation rules describe which kinds of node and edges are created to mark graph pattern matches. The graph transformation rules mark all nodes of graph pattern matches. Nodes with special roles in the graph pattern match are marked by edges with certain edge types (cf. dashed lines). Nodes without special roles in graph pattern matched are marked by scopes (cf. dotted lines). For example, the Generalization view module marks with SuperRole and SubRole edges which class acts as super- and subclass. The BoundedAssociation and UnboundedAssociation view modules marks with Reference and Target edges which field acts as reference and which classifier is the target of the reference. The Composite view module reuses nodes that mark matches for the Generalization and Association graph patterns. The Composite view module marks with Composite, Component, Generalization, and Association edges which class acts as composite and component and which generalization and association graph pattern matches are reused.

4.2.1 Mapping Graph Conditions

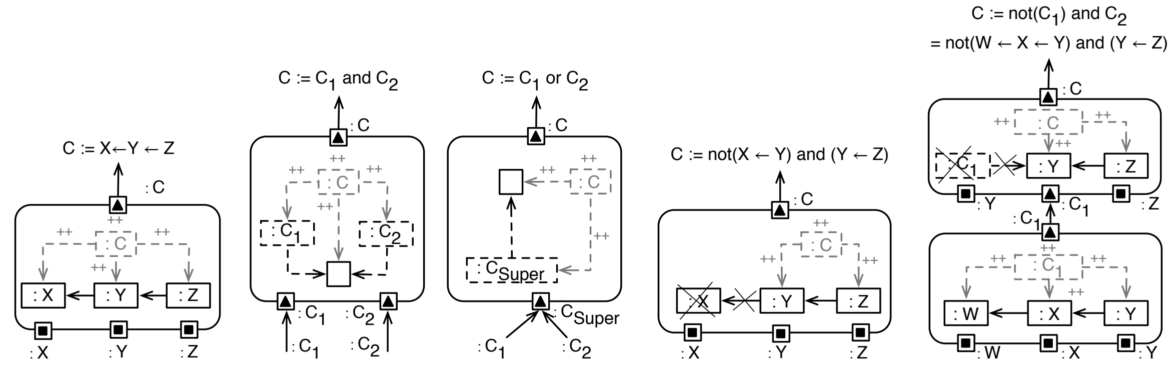
Fig. 4 shows a schematic mapping of graph conditions [9] to our network of view modules. If c is a graph condition, then also $\neg c$ is a graph condition. Furthermore, if c_i is a graph condition, then $\vee c_i$ and $\wedge c_i$ with index set $i \in I$ are graph conditions.

Atomic Graph Condition Atomic graph conditions (cf. Fig. 4a) are mapped to single view modules that only receive nodes of base graphs. Atomic graph conditions enable to express basic conditions on graphs, e. g., the existence of certain nodes and edges. According to our running example, the Generalization, BoundedAssociation, and UnboundedAssociation view modules implement atomic graph conditions.

Conjunction View modules with more than one input connector that receive nodes of view graphs implement conjunctions (cf. Fig. 4b). According to our running example, the Composite view module implements a conjunction of graph conditions for generalizations and associations.

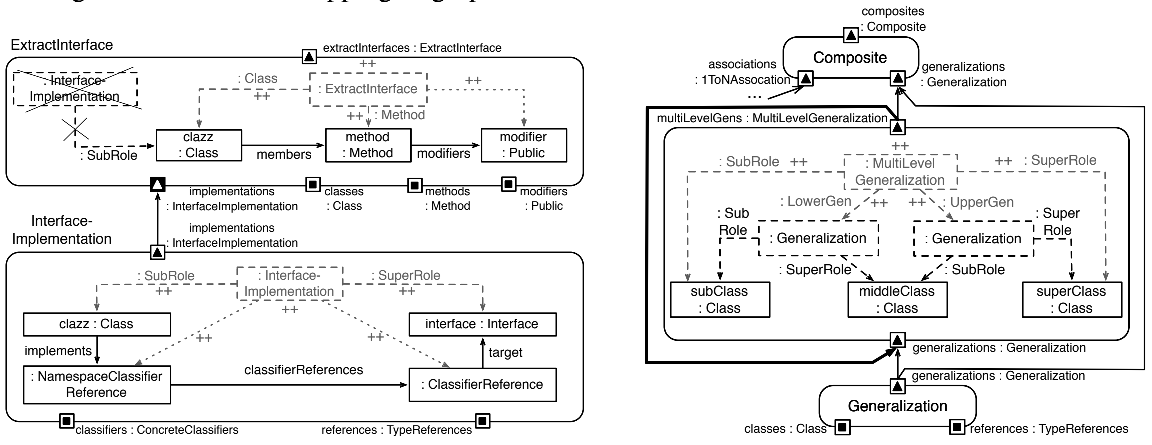
Disjunction View modules with an input connector, which receives nodes from more than one predecessor view module, implement disjunctions (cf. Fig. 4c). The input connector must specify the node super type of required nodes of view graphs. According to our running example, the Composite view module implements a disjunction of graph conditions for unbounded and bounded associations. Therefore, the Composite view module consists of an input connector with Association node type.

Simple NAC We refer to the term simple negative application conditions (NAC), when all negated graph



(a) Atomic condition (b) Conjunction (c) Disjunction (d) Simple NAC (e) Complex NAC

Figure 4: Schematic mapping of graph conditions to a discrimination network of view modules



(a) Complex NAC for ExtractInterface (b) Recursive definition of Multi-Level Generalization

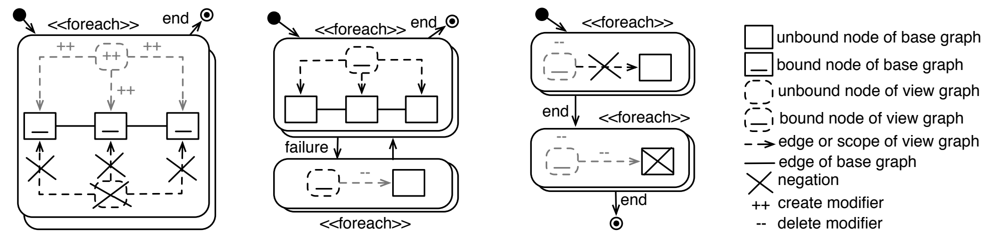
Figure 5: Complex NAC and recursion

patterns nodes are directly connected to non-negated graph pattern nodes, i. e., positive application conditions (PACs). Our approach maps simple NACs to negated graph pattern nodes that refer to base graphs within graph transformation rules of view modules (cf. Fig. 4d).

Complex NAC We refer to the term complex NAC, when at least one negated graph pattern node is not *directly* connected to a PAC. Our approach splits up complex NACs in two view modules (cf. Fig. 4e). The first view module searches for graph pattern matches for the negated part of the graph condition without negation. For example, instead of searching for graph pattern matches that satisfy graph condition $\neg c$, the first view module searches for graph pattern matches that satisfy graph condition c and creates nodes in view graphs that mark theses graph pattern matches, accordingly. Then, the second view module checks that no node in view graphs exists that satisfies graph condition c. Fig. 5a shows an example for mapping complex NACs to view modules. Fig. 5a shows two view modules that implement graph patterns for InterfaceImplementation and ExtractInterface. The InterfaceImplementation view module marks graph pattern matches for classes that implement an interface. The ExtractInterface view module marks graph pattern matches for classes that own public methods, but do *not* implement interfaces. The latter is denoted by the crossed out InterfaceImplementation node and SubRole edge.

4.2.2 Recursion

Recursion is mapped to cyclic dependencies of view modules, e. g., to search for graph patterns that employ path expressions. These cycles can consist of multiple view modules. In general, one view module that does not belong to the cycle itself describes the recursion start, while view modules within



(a) Create mode (b) Update mode (c) Delete mode (d) Legend
Figure 6: Schematic graph transformation rules for execution modes of view modules

the cycle describe the recursion step. According to our running example, the Composite view module has to consider multi-level generalizations as well to also detect variants of the Composite design pattern that employ multiple inheritance levels. Fig. 5b shows a recursive definition to find graph patterns matches for multi-level generalizations. The Generalization view module (cf. Fig. 3) describes the recursion start. The MultiLevelGeneralization view module in Fig. 5b describes the recursion step, because nodes of view graphs that mark graph pattern matches for multi-level generalizations can lead to additional matches for multi-level generalizations. Therefore, the MultiLevelGeneralization view module consists of a dependency between its own output and input connector (cf. bold line in Fig. 5b). Fig. 2a shows that the MultiLevelGeneralization node type is a specialization of the Generalization node type and, additionally, owns the LowerGen and UpperGen edge types that describe which reused nodes of type Generalization act as lower and upper generalization. Note that lower and upper generalizations can be multi-level generalizations as well. Fig. 5b shows the implementation of the MultiLevelGeneralization view module that checks whether two (multi-level) generalizations exist that have a class in common, which acts as superclass in one generalization and as subclass in another generalization. If yes, the Multi-LevelGeneralization view module creates a node that marks both Generalization nodes and the super- and subclass of the detected multi-level generalization.

5 Incremental Maintenance of Graph Database Views

In this section, we describe our maintenance algorithm for nodes of view graphs. Our incremental maintenance algorithm consists of maintenance phases that compute the search space for the execution of view modules. Each maintenance phase executes view module in analogous modes to a) create nodes of view graphs for new graph pattern matches, b) update existing nodes of view graphs that already mark graph pattern matches, and c) delete nodes of view graphs that do not mark graph pattern matches anymore. We describe our incremental maintenance algorithm bottom-up. Sec. 5.1 describes the view module execution modes. Sec. 5.2 describes how the search space for view modules is computed. Sec. 5.3 describes the maintenance phases that execute view modules with the computed search space.

5.1 View Module Execution Modes

A view module can be executed in three different modes called Update, Delete, and Create. In Create mode, view modules search for graph pattern matches, mark these matches with the help of nodes and edges in view graphs, and return created nodes of view graphs. In Update mode, view modules check whether nodes of view graphs still mark matches for satisfied graph patterns, set nodes of view graphs obsolete, if they do not mark matches for satisfied graph patterns anymore, and return revised nodes of view graphs. In Delete mode, view modules delete nodes of view graphs that were set obsolete during Update mode or consist of dangling edges due to deleted nodes of base graphs and view graphs.

Fig. 6 shows graph transformation rules hidden by view modules in terms of schematic story diagrams [10] to describe the behavior of view modules in each mode. Each view module that initially created a node of view graphs is responsible for its maintenance. Nodes of view graphs know which view module created them and view modules know which nodes of view graphs they created.

Create Mode In Create mode, view modules receive nodes of base graphs and view graphs according to their input connectors. These nodes are bound in the graph transformation rule as depicted by Fig. 6a. If these nodes satisfy the left-hand side of the graph transformation rule and the found graph pattern match is *not* already marked by a node of the same type in view graphs as depicted by the negated nodes, edges, and scopes, the graph transformation rule creates a node, edges, and scopes in view graphs that mark all nodes of the found graph pattern match as depicted by the node, edges, and scopes with create modifier. Thus, view modules mark graph pattern matches at most once.

Update Mode In Update mode, view modules receive nodes of view graphs that must be revised to check whether they still mark matches for satisfied graph patterns. These nodes are bound in the graph transformation rule as depicted by Fig. 6b. If these nodes of view graphs do not mark matches for satisfied graph patterns anymore, the graph transformation rule sets these nodes obsolete as depicted by the failure edge and removal of all edges that are used to mark the graph pattern match. Otherwise, the node, edges, and scopes of view graphs are preserved.

Delete Mode In Delete mode, view modules receive nodes of view graphs that are obsolete. These nodes are bound in the graph transformation rule as depicted by Fig. 6c. The graph transformation rule deletes nodes and their edges from view graphs, if they do not consist of edges or scopes anymore (cf. activity on top) or consist of dandling edges or scopes that do *not* mark a node anymore (cf. activity at the bottom).

5.2 Computation of View Module Input

Base and view graph changes are used to derive suspicious, obsolete, and missing nodes of view graphs.

Suspicious Nodes Nodes of view graphs are suspicious when they are connected to at least one modified node of base graphs, are connected to another suspicious node of view graphs, or view modules created new nodes in view graphs that dissatisfy complex NACs. A node is modified when an attribute value of the node changed or two nodes are modified when an edge is added or deleted that connects both nodes. Our approach uses modified nodes of base graphs to look up connected suspicious nodes of view graphs. Furthermore, when view modules create new nodes in view graphs, complex NACs implemented by dependent view modules may become dissatisfied. Therefore, our approach employs a reachability test to collect suspicious nodes of view graphs, when view modules create new nodes. The reachability test collects nodes of view graphs that are directly or indirectly reachable from created nodes in view graphs. The reachability test only traverses nodes, if they have the same node (sub-)type as input connectors of view modules that dependent on the view module that created new nodes. For example, when the InterfaceImplementation view module (cf. Fig. 5a) creates a new node of type InterfaceImplementation in view graphs, the reachability test looks up all reachable nodes of type ExtractInterface by traversing nodes of type Class, Method, and Public (cf. Fig. 5a). In Update mode, view modules use suspicious nodes of view graphs to check whether they still mark matches for satisfied graph patterns.

Obsolete Nodes A node of view graphs is *obsolete* when the node consists of at least one dangling edge or scope that is not connected to a node anymore. Our approach uses deleted nodes of base graphs to look up nodes of view graphs that became obsolete. In Delete mode, view modules delete obsolete nodes.

Missing Nodes A *missing* node of view graphs is a node that currently does not exist in view graphs, although it must exist due to changes of base graphs that result in new graph pattern matches. Our approach employs a reachability test that collects nodes of base graphs and view graphs that may result in new graph pattern matches. The reachability test collects all nodes that are directly or indirectly reachable from a) created and modified nodes of base graphs or b) are marked by created / were marked by deleted nodes of view graphs. The reachability test only collects nodes, if they have the same node (sub-)type as input connectors of the view module. For example, when a node of type Class is added to

```
procedure UPDATE(suspiciousNodes)
  obsoletes := ∅
  for node in suspiciousNodes do
    module := node.module
    module.update(node)
    if node is obsolete then
      obsoletes := obsoletes ∪ {node}
    else
      dependents := node.dependents
      obsoletes := obsoletes ∪ UPDATE(dependents)
    end if
  end for
  return obsoletes
end procedure
```

(a) Update phase

```
procedure DELETE(obsoleteNodes)
  changed := ∅
  for node in obsoleteNodes do
    module := node.module
    module.delete(node)
    changed := changed ∪ {previously marked nodes}
    dependents := node.dependents
    changed := changed ∪ DELETE(dependents)
  end for
  return changed
end procedure
```

(b) Delete phase

```
procedure CREATE(changedNodes)
  suspicious := ∅
  result := ∅
  while hasNextModule(result) do                    //handles recursion
    module := nextModule(result)                    //handles recursion
    candidates := reachabilityMissing(changedNodes,module)
    result := module.create(candidates)
    dependents := module.dependents
    suspicious := suspicious ∪ reachabilitySuspicious(result,dependents)
  end while
  return suspicious
end procedure
```

(c) Create phase

Figure 8: Execution of view module during maintenance phases

base graphs, the reachability test for the Generalization view module (cf. Fig. 3) collects all nodes of type Class and TypeReference that are reachable from the added node. We assume that view modules employ graph patterns that are connected graphs and edges between nodes can be traversed bidirectionally.

5.3 Maintenance Phases

Our maintenance algorithm employs the subsequent maintenance phases Update, Delete, and Create to maintain suspicious, obsolete, and missing nodes of view graphs. Fig. 7 shows the order of the maintenance phases. The algorithm passes suspicious nodes of view graphs to the Update phase, obsolete nodes of view graphs to the Delete phase, and added / modified nodes of base and view graphs to the Create phase. If the Create phase returns new suspicious nodes of view graphs, the algorithm executes an additional cycle of Update, Delete, and Create phases to revise these suspicious nodes.

```
procedure MAINTAIN(events)
  suspicious := ∅
  repeat
    suspicious:= suspicious ∪ suspiciousNodes(events)
    obsoletes := UPDATE(suspicious)
    obsoletes := obsoletes ∪ obsoletesNodes(events)
    changed := DELETE(obsoletes)
    changed := changed ∪ changedNodes(events)
    suspicious := CREATE(changed)
    events := ∅
  until suspicious = ∅
end procedure
```

Figure 7: Order of maintenance phases

Update Phase Fig. 8a shows pseudo code for the Update phase. For each suspicious node of view graphs, the algorithm looks up the responsible view module, passes the suspicious node to the view module, and executes the view module in Update mode. If the view module sets the node obsolete, the node is added to the set of obsolete nodes. Otherwise, all nodes that depend on the updated node are updated as well. Finally, the Update phase returns all collected obsolete nodes.

Delete Phase Fig. 8b shows pseudo code for the Delete phase. For each obsolete node of view graphs, the algorithm looks up the responsible view module, passes the obsolete node to the view module, and executes the view module in Delete mode. The view module returns all nodes that were previously marked by the deleted obsolete node. The algorithm considers the returned nodes as changed and collects them. Afterwards, all nodes that depend on the deleted node are removed as well. Finally, the algorithm returns all nodes that were previously marked by the deleted obsolete nodes of view graphs.

Create Phase Fig. 8c shows pseudo code for the Create phase. The algorithm iterates the network of view modules with respect to recursion cycles. The algorithm executes each view module in Create mode and passes candidate nodes, which may lead to new graph pattern matches (cf. Sec. 5.2), to view modules. The view module returns all created nodes of view graphs that mark new graph pattern matches. The algorithm uses the created nodes to determine nodes of view graphs that become suspicious (cf. Sec. 5.2). These suspicious nodes are collected and returned at the end of the Create phase.

5.3.1 Positive and Negative Application Conditions

Graph conditions must be mapped to our view modules as described in Sec. 4.2.1.

PACs Created and modified nodes of base and view graphs may satisfy PACs. Deleted and modified nodes of base and view graphs may dissatisfy PACs. The Update phase derives suspicious nodes of view graphs from modified nodes (cf. Sec. 5.2) and sets these suspicious nodes obsolete, if required. The Delete phase deletes obsolete nodes of view graphs that were set obsolete by the previous Update phase or are obsolete due to deleted nodes of base and view graphs. The Create phase uses created and modified nodes of base graphs as well as nodes of view graphs created and deleted by predecessor view modules to compute the candidate nodes for view modules. Thus, our algorithm supports PACs.

Simple NACs Deleted and modified nodes of base graphs may satisfy simple NACs. Nodes in base graphs that were connected to a deleted node of base graphs are considered as modified. Created and modified nodes of base graphs are used to compute the candidate nodes for view modules in the Create phase (cf. Sec. 5.2). Thus, the Create phase detects satisfied simple NACs. Created and modified nodes of base graphs may dissatisfy simple NACs. When a node of base graphs is added via an edge to another node of base graphs, both nodes are considered as modified. Modified nodes are used to derive suspicious nodes in view graphs (cf. Sec. 5.2). Thus, the Update phase detects dissatisfied simple NACs.

Complex NACs Created nodes of view graphs may dissatisfy complex NACs and, thus, may make other nodes of view graphs obsolete. When the Create phase creates new nodes in view graphs, our algorithm looks up suspicious nodes of view graphs (cf. Sec. 5.2). If such suspicious nodes exist, an additional Update, Delete, and Create sequence is employed to revise these suspicious nodes. Thus, dissatisfied complex NACs are detected during the Update phase. Deleted nodes of view graphs may satisfy complex NACs and, thus, cause missing nodes in view graphs. Nodes of base and view graphs that were connected to deleted nodes of view graphs are used to compute the candidate nodes for view modules to find missing nodes in view graphs (cf. Sec. 5.2). Thus, satisfied complex NACs are detected in the Create phase.

5.3.2 Recursion

When the discrimination network is acyclic, a topological sorting is sufficient to execute view modules in correct order. When the discrimination network consists of cyclic dependencies between view modules, an execution plan is generated that considers these cyclic dependencies when sorting view modules for execution. We refer to cyclic dependencies between view modules as *recursion cycle*. Recursion cycles of view modules are executed until the fix point view module reaches a fix point. A fix point view module is a view module that has at least one dependent view module that is *not* part of the recursion cycle or is connected to the termination of the network. A fix point view module reached a fix point when it did not update, delete, or create nodes of view graphs, because then dependent view modules *within* the recursion cycle are not impacted by the output of the fix point view module anymore.

During Create phase, nodes created by a fix point module are passed to dependent view modules *within* the recursion cycle, first. When the fix point view module reached its fix point, the nodes created by the fix point view module are passed to dependent view modules that do *not* belong to the recursion cycle. The recursion cycle terminates when the fix point view module has a fix point. During Create phase, recursion cycles terminate when the number of nodes in base graphs has an upper bound, nodes of view graphs mark at least one node in base graphs (i. e., nodes of view graphs that only mark nodes of view graphs are not permitted), and view modules in the recursion cycle do not create and delete the same kind of nodes in view graphs. Then, the number of nodes in view graphs has also an upper bound, because each node of base graphs is marked by at most one node of the same type in view graphs due to the NAC of the graph transformation rule in Create mode (cf. Fig. 6a). During Update and Delete phase, the edges and scopes between nodes of view graphs are used to revise dependent suspicious nodes and delete dependent obsolete nodes. Since the number of dependent nodes has an upper bound when the Create phase terminated, the revision and deletion of dependent nodes in view graphs terminates.

Projects	#Nodes in BG		#Nodes in VG		View Size (MB)		Execution Time for 100 revisions		Speedup
Apache...	Rev.1	Rev.100	Rev.1	Rev.100	Rev.1	Rev.100	Batch	Incremental	
Ant	12442	22071	1725	2927	1,2	2,1	0 h 9 min 14 s	0 min 21 s	26,38
Xerces	133858	191246	20050	25636	13,3	16,7	15 h 33 min 44 s	17 min 42 s	52,75

(a) Comparison of our batch and maintenance algorithms concerning execution time

Projects	Emulated Rete			Our Approach			Speedup
Apache...	Incremental	#Nodes in VG	View Size (MB)	Incremental	#Nodes in VG	View Size (MB)	
Ant	4,9 s	296 - 437	0,171 - 0,249	1,8 s	5 - 20	0,005 - 0,020	2,7
Xerces	43,5 s	1155 - 1764	0,657 - 1,003	13,9 s	2	0,003	3,1

(b) Comparison of emulated Rete network and our approach concerning memory footprint and execution time

Table 1: Overview of evaluation results (BG: base graph; VG: view graph)

6 Evaluation

In this section, we evaluate the performance of our incremental maintenance algorithm and compare the performance of ordinary Rete network structures [11] with generalized Gator network structures [15] for the incremental maintenance of view graphs. For evaluation purposes, we implemented a batch algorithm as reference algorithm and our incremental maintenance algorithm using the Eclipse Modeling Framework (EMF) and story diagrams [10] as operationalization of view modules. The batch maintenance algorithm [3] considers *all* nodes of view graphs created by a view module as suspicious, checks for *all* nodes of view graphs created by a view module whether they are obsolete, and uses *all* nodes that have a node (sub-)type as specified by input connectors of view modules to find missing nodes in view graphs. According to our running example, we recover software design patterns in ASGs of Java source code. We preprocessed the first 100 source code revisions of the Apache Ant and Xerces source code repositories to derive EMF models. Both algorithms result in exactly the same nodes in view graphs.

First, we compared the performance of the batch and incremental algorithm. We performed a view maintenance for each revision using either our batch or incremental algorithm. We used the history of the source code repositories to merge applied modifications into the ASGs using EMFCompare. We employed 49 view modules [3] to recover design patterns. We do not consider precision and recall of the recovered design patterns, because they are the same for both algorithms. Table 1a shows the number of nodes in base and view graphs for revision 1 and 100. For Ant 2,24% and for Xerces 0,82% of nodes in base graphs changed between two revisions in average. The computed candidate sets had a size of 10,36% for Ant and 4,96% for Xerces in comparison to the number of nodes passed to view modules during batch maintenance. In total, the incremental algorithm is approx. 26 times faster for Ant and approx. 53 times faster for Xerces than the batch algorithm.

Second, we compared the performance of Rete and Gator network structures. The runtime performance of a discrimination network depends on the network topology as stated by Varro et al. [30] for incremental graph pattern matching using Rete networks and Hanson et al. [15] for maintaining materialized views of *relational* databases using Gator networks. For our experiment, we used the Gator network presented in our running example and emulated an equivalent Rete network with 15 modules using our approach. Finding an optimal network structure is a non-trivial task. We followed optimization criteria from related work [2, 15, 30] such as matching to-one reference early in the network and using few network nodes to reduce memory footprint. We do not use the Rete based matcher [2] of EMF-IncQuery [1] to measure the performance of the Rete network, because we aim for comparing concepts instead of different technologies such as different employed graph pattern matchers. Table 1b shows in total the time required for incremental maintenance, number of nodes in view graphs, and view graph size for the first 100 revisions. Our approach is 2,7 times for Ant and 3,1 times for Xerces faster than the equivalent emulated Rete network. Our approach requires 2,92% - 8,03% for Ant and 0,30% - 0,45% for Xerces of memory in comparison to the equivalent emulated Rete network. The measurement proves that Gator networks *can* outperform Rete networks in time and space also for incremental graph pattern matching.

7 Related Work

We describe *discrimination networks* in Sec. 2. No research exists that employs Gator networks [15] for incremental graph pattern matching or view maintenance of *graph* databases, although the authors showed that optimized Gator networks *can* outperform Rete networks for view maintenance of *relational* databases. No research exists, which shows that Gator networks *can* outperform Rete networks for incremental graph pattern matching. In contrast to existing approaches for incremental graph pattern matching, our approach supports Gator networks with general network structures, while Rete network based approaches are limited to network nodes with at most two inputs. Thus, Rete networks require more network nodes to maintain matches for certain graph patterns and, thus, have to store more intermediate graph pattern matches than Gator networks. Thus, Rete networks require more comparison operations than Gator networks, when they update the state of the network. Thus, also for incremental graph pattern matching Gator networks *can* outperform Rete networks in space and time. In contrast to Rete and Gator networks, our approach supports cyclic network structures to enable recursion.

Database view maintenance is often performed incrementally. *Relational* databases employ an impact analysis [16], derive incremental maintenance queries that transfer views into a consistent state [23], or employ discrimination networks for view maintenance [15]. *Object-oriented* databases make use of object-oriented concepts and map objects to tables. For view maintenance, object-oriented databases re-write queries to make all tables explicit in view definitions to also consider inherited and inheriting tables [19]. *Graph* databases such as GRAS [17], GRACE [29], and Neo4j [26] do not provide concepts to define and maintain graph database views. Zhuge et al. [32] introduce the notion of graph-structured databases and employ delegate nodes that reference nodes in graphs to constitute views. We extended their delegate concept by adding node types to delegate nodes and edge types to effectively refer to nodes with certain roles in graph pattern matches. However, the authors limit their approach to tree-structured data and employ selection paths and conditions (i. e., no graph patterns) to define views.

Graph indexing approaches focus on path-based indexing (e. g. APEX [6]) for fast evaluation of path expressions and approaches that index frequent graph-structures for fast subgraph isomorphism tests (e. g. gIndex [31]). However, these approaches neither maintain graph pattern matches nor focus on the maintenance of such indexes. *Model search* approaches aim for the efficient retrieval of model elements. For example, Moogle [20] maps model search to text-based search. The maintenance of search indexes is not in scope of these approaches and they do not index and maintain graph pattern matches.

VIATRA2 [2] provides Rete network based matching for incremental model queries over EMF models in EMF-IncQuery [1], the incremental derivation of features (attributes and references) in EMF models [25], live model transformations to propagate changes of source models to target models [24], and the incremental derivation of view models [7]. In contrast to our approach, these approaches are limited to Rete networks. Model constraint evaluation approaches rewrite constraints based on model changes [5] to reduce computational complexity when re-evaluating model constraints or employ model profilers to keep track of model element that are traversed when evaluating model constraints to be aware of which modified model elements demand a re-evaluation of model constraints [8]. In our previous work [28], we maintain traceability links incrementally by employing localization rules to create and delete traceability links without any explicit procedure for updating suspicious traceability links. In this paper, we extended our previous approach to support general graph pattern matches of arbitrary domains by making use of Gator networks and an explicit `Update` phase for suspicious graph pattern matches. Ordinary Rete and Gator networks do not consist of an explicit `Update` phase and instead map updates to a sequence of delete and create steps. We extended our previous approach to support NACs during maintenance.

8 Conclusion and Future Work

In this paper, we present an enumeration mechanism that enables to mark and maintain graph pattern matches in graph data effectively and efficiently. We describe how our enumeration mechanism is used to define views over graph data using view modules. These view modules hide graph transformation rules and, therefore, the presented concepts can be easily transferred to arbitrary kinds of graph data and graph databases. Our incremental maintenance algorithm keeps enumerations of graph pattern matches consistent with graph pattern matches in graph data and supports complex NACs and recursion. Our evaluation shows that our incremental algorithm scales when the size of the graph data increases and that Gator networks *can* outperform Rete networks in time and space for incremental maintenance of graph pattern matches. As future work, we prove that our approach is as expressive as nested graph conditions.

References

[1] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *Model Driven Engineering Languages and Systems*, *LNCS* 6394, Springer, pp. 76–90, doi:10.1007/978-3-642-16145-2_6.

[2] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró & Gergely Varró (2008): *Incremental Pattern Matching in the VIATRA Model Transformation System*. In: *Proceedings of the 3rd International Workshop on Graph and Model Transformations*, GRaMoT '08, ACM, pp. 25–32, doi:10.1145/1402947.1402953.

[3] Thomas Beyhl & Holger Giese (2015): *Efficient and Scalable Graph View Maintenance for Deductive Graph Databases based on Generalized Discrimination Networks*. Technical Report, Hasso Plattner Institute at the University of Potsdam. Available at `http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-79535`.

[4] H. Bunke, T. Glauser & T.-H. Tran (1991): *An efficient implementation of graph grammars based on the RETE matching algorithm*. In: *Graph Grammars and Their Application to Computer Science*, *LNCS* 532, Springer, pp. 174–189, doi:10.1007/BFb0017389.

[5] Jordi Cabot & Ernest Teniente (2006): *Incremental Evaluation of OCL Constraints*. In: *Advanced Information Systems Engineering*, Springer, pp. 81–95, doi:10.1007/11767138_7.

[6] Chin-Wan Chung, Jun-Ki Min & Kyuseok Shim (2002): *APEX: An Adaptive Path Index for XML Data*. In: *Proceedings of the International Conference on Management of Data*, SIGMOD '02, ACM, pp. 121–132, doi:10.1145/564691.564706.

[7] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth & Dániel Varró (2014): *Query-driven Incremental Synchronization of View Models*. In: *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '14, ACM, pp. 31–38, doi:10.1145/2631675.2631677.

[8] Alexander Egyed (2006): *Instant Consistency Checking for the UML*. In: *Proceedings of the 28th International Conference on Software Engineering*, ACM, pp. 381–390, doi:10.1145/1134285.1134339.

[9] Hartmut Ehrig, Karsten Ehrig, Annegret Habel & Karl-Heinz Pennemann (2004): *Constraints and Application Conditions: From Graphs to High-Level Structures*. In: *International Conference on Graph Transformations*, *LNCS* 3256, Springer, pp. 287–303, doi:10.1007/978-3-540-30203-2_21.

[10] Thorsten Fischer, Jörg Niere, Lars Torunski & Albert Zündorf (2000): *Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java*. In: *Theory and Application of Graph Transformations*, Springer, pp. 296–309, doi:10.1007/978-3-540-46464-8_21.

[11] Charles L. Forgy (1982): *Rete: A Fast Algorithm for the Many Pattern/Many object Pattern Match Problem*. *Artificial Intelligence* 19(1), pp. 17–37, doi:10.1016/0004-3702(82)90020-0.

[12] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1994): *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[13] Michael R. Garey & David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Company.

[14] Eric N. Hanson (1996): *The Design and Implementation of the Ariel Active Database Rule System*. Transactions on Knowledge and Data Engineering 8(1), pp. 157–172, doi:10.1109/69.485644.

[15] Eric N. Hanson, Sreenath Bodagala & Ullas Chadaga (2002): *Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks*. Transactions on Knowledge and Data Engineering 14(2), pp. 261–280, doi:10.1109/69.991716.

[16] John V. Harrison & Suzanne W. Dietrich (1992): *Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach*. In: Workshop on Deductive Databases, JICSLP, pp. 56–65.

[17] Norbert Kiesel, Andy Schürr & Bernhard Westfechtel (1993): *GRAS, a graph-oriented database system for (software) engineering applications*. In: Proceeding of the 6th International Workshop on Computer-Aided Software Engineering, IEEE, pp. 272–286, doi:10.1109/CASE.1993.634829.

[18] Ho Soo Lee & Marshall I. Schor (1992): *Match Algorithms for Generalized Rete Networks*. Artificial Intelligence 54(2), pp. 249–274, doi:10.1016/0004-3702(92)90047-2.

[19] Jixue Liu, Millist Vincent & Mukesh Mohania (2000): *Maintaining Views in Object-Relational Databases*. In: Proceedings of the 9th International Conference on Information and Knowledge Management, CIKM '00, ACM, pp. 102–109, doi:10.1145/354756.354807.

[20] Daniel Lucrédio, Renata Fortes & Jon Whittle (2010): *MOOGLE: a metamodel-based model search engine*. Software & Systems Modeling 11(2), pp. 183–208, doi:10.1007/s10270-010-0167-7.

[21] Daniel P. Miranker (1987): *TREAT: A Better Match Algorithm for AI Production Systems*. In: Proceedings of the 6th National Conference on Artificial Intelligence, 1, AAAI Press, pp. 42–47.

[22] Jörg Niere, Jörg Wadsack & Lothar Wendehals (2003): *Handling large search space in pattern-based reverse engineering*. In: Proceedings of the 11th International Workshop on Program Comprehension, IEEE, pp. 274–279, doi:10.1109/WPC.2003.1199212.

[23] Xiaolei Qian & Gio Wiederhold (1991): *Incremental Recomputation of Active Relational Expressions*. Transactions on Knowledge and Data Engineering 3(3), pp. 337–341, doi:10.1109/69.91063.

[24] István Ráth, Gábor Bergmann, András Ökrös & Dániel Varró (2008): *Live Model Transformations Driven by Incremental Pattern Matching*. In: Proceedings of the 6th International Conference on Theory and Practice of Model Transformations, Springer, pp. 107–121, doi:10.1007/978-3-540-69927-9_8.

[25] István Ráth, Ábel Hegedüs & Dániel Varró (2012): *Derived Features for EMF by Integrating Advanced Model Queries*. In: Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA'12, Springer, pp. 102–117, doi:10.1007/978-3-642-31491-9_10.

[26] Ian Robinson, Jim Webber & Emil Eifrem (2015): *Graph Databases (Second Edition)*. O'Reilly Media.

[27] Marko A. Rodriguez & Peter Neubauer (2010): *The Graph Traversal Pattern*. CoRR Journal 1004(1001).

[28] Andreas Seibel, Stefan Neumann & Holger Giese (2010): *Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance*. Software & Systems Modeling 9(4), pp. 493–528, doi:10.1007/s10270-009-0146-z.

[29] Srinath Srinivasa & Martin Maier (2005): *LWI and Safari: A New Index Structure and Query Model for Graph Databases*. In: Proceedings of the 11th International Conference on Management of Data, Computer Society of India.

[30] Gergely Varró & Frederik Deckwerth (2013): *A Rete Network Construction Algorithm for Incremental Pattern Matching*. In: Theory and Practice of Model Transformations, LNCS 7909, Springer, pp. 125–140, doi:10.1007/978-3-642-38883-5_13.

[31] Xifeng Yan, Philip S. Yu & Jiawei Han (2005): *Graph Indexing Based on Discriminative Frequent Structure Analysis*. Transactions on Database Systems 30(4), pp. 960–993, doi:10.1145/1114244.1114248.

[32] Yue Zhuge & H. Garcia-Molina (1998): *Graph Structured Views and Their Incremental Maintenance*. In: Proceedings of the 14th International Conference on Data Engineering, IEEE, pp. 116–125, doi:10.1109/ICDE.1998.655767.