Efficient and Scalable Graph View Maintenance for Deductive Graph Databases based on Generalized Discrimination Networks

Thomas Beyhl, Holger Giese

Technische Berichte Nr. 99

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 99

Thomas Beyhl | Holger Giese

Efficient and Scalable Graph View Maintenance for Deductive Graph Databases based on Generalized Discrimination Networks

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2015
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-339-8

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-79535
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-79535

Graph databases provide a natural way of storing and querying graph data. In contrast to relational databases, queries over graph databases enable to refer directly to the graph structure of such graph data. For example, graph pattern matching can be employed to formulate queries over graph data.

However, as for relational databases running complex queries can be very time-consuming and ruin the interactivity with the database. One possible approach to deal with this performance issue is to employ database views that consist of pre-computed answers to common and often stated queries. But to ensure that database views yield consistent query results in comparison with the data from which they are derived, these database views must be updated before queries make use of these database views. Such a maintenance of database views must be performed efficiently, otherwise the effort to create and maintain views may not pay off in comparison to processing the queries directly on the data from which the database views are derived.

At the time of writing, graph databases do not support database views and are limited to graph indexes that index nodes and edges of the graph data for fast query evaluation, but do not enable to maintain pre-computed answers of complex queries over graph data. Moreover, the maintenance of database views in graph databases becomes even more challenging when negation and recursion have to be supported as in deductive relational databases.

In this technical report, we present an approach for the efficient and scalable incremental graph view maintenance for deductive graph databases. The main concept of our approach is a generalized discrimination network that enables to model nested graph conditions including negative application conditions and recursion, which specify the content of graph views derived from graph data stored by graph databases. The discrimination network enables to automatically derive generic maintenance rules using graph transformations for maintaining graph views in case the graph data from which the graph views are derived change. We evaluate our approach in terms of a case study using multiple data sets derived from open source projects.

Contents

1. Introduction

Nowadays, graph data is ubiquitous and browsing these graph data is an elementary task to work with graph data. For example, users in social networks and their relationships constitute a graph and a query that answers the cause of friendship for two companioned users is an interesting and also complex query. Another example is the domain of software engineering where abstract syntax graphs of source code and models are queried for, e.g., employed software design patterns as defined by Gamma et al. [23] to investigate software architectures or recommend refactorings as proposed by Fowler [22] to improve the source code. Also queries between graph data with different schemes are stated in practice. For example, searching for chains of traceability links between graphs that represent requirement documents (e.g., SysML requirement models [65]), abstract syntax graphs of models (e.g., UML class models [64]), and source code (e.g., Java source code).

In practice, graph databases provide a natural way of storing and querying graph data. One advantage of this fact is that queries that process graph data can refer directly to this graph structure. For example, graph pattern matching can be employed to formulate queries over graph data. However, graph pattern matching can be very time-consuming when the size of the graph data increases to a large number of nodes and edges. For example, subgraph isomorphism testing used for graph pattern matching is known to be NP-complete [18].

Furthermore, as for relational databases running complex queries always from scratch although only few nodes and edges of the data graphs in the graph database changed can be very inefficient. One possible approach to deal with this performance issue is to employ database views that consist of pre-computed answers to common and often stated queries. According to Gupta et al. [32] *"a view [...] defines a function from a set of base tables to a derived table"* for relational databases. But, to ensure that database views yield consistent query results in comparison with the data from which they are derived database views must be updated before queries make use of these database views. Gupta et al. [32] refer to *"the process of updating [...] views in response to changes to the underlying data [as] view maintenance"*. Such a maintenance of views must be performed efficiently, otherwise the effort to create and maintain such database views may not pay off in comparison to processing the queries directly on the data from which database views are derived. Therefore, often incremental view maintenance is employed that *"computes changes to a view in response to changes to the base relations"* [32] in case of relational databases.

Database views and view maintenance would be also beneficial for graph databases. However, current graph databases do not support database views and are limited to graph indexes that index nodes and edges of the graph data for fast query evaluation, but do not enable to maintain pre-computed answers of complex queries over graph data. Moreover, when views for graph databases have to be supported, several questions arise. What exactly is a view in graph databases (cf. [82])? Which nodes and edges are included in views? Is a view in graph databases a copy of subgraphs similar to copies of relations in relational databases? How are dependencies between nodes in separate views of graph databases represented?

In this technical report, we present an approach for the efficient and scalable incremental graph view maintenance for deductive graph databases. Our contribution is twofold. First, we present a view definition approach that enables to specify views for graph databases in terms of discrimination networks using graph patterns including positive application conditions, negative application conditions and recursion. Second, we describe batch and incremental maintenance procedures that maintain the content of derived views by (partially) re-executing the discrimination network. The advantage of our approach is that the discrimination network enables to automatically derive generic maintenance rules for maintaining graph views in case the graph data from which the graph views are derived changed. We evaluate our approach in terms of a case study using multiple data sets derived from open source projects.

1.1. State of the Art

View maintenance has been employed for relational databases, e.g. snapshot [14], deferred [13], and immediate [33] view maintenance. However, current graph databases such as Neo4J[1], AllegroGraph[2], and InfiniteGraph[3] do not provide a concept for view definition and maintenance. In the best case, they support graph indexes, which enable a fast query evaluation of nodes and edges that consist of certain properties such as types or attribute values, but do not enable to define and maintain views that keep ready answers for complex queries.

[1]`http://neo4j.com` (last access: April 28th 2015).
[2]`http://franz.com/agraph/allegrograph/` (last access: April 28th 2015).
[3]`http://www.infinitegraph.com` (last access: April 28th 2015).

For view maintenance of relational databases, discrimination networks such as RETE [21], TREAT [61], and Gator [37] have been used to enable incremental view maintenance. However, to the best of our knowledge no approach exists that employs discrimination networks for the incremental view maintenance of graph databases.

Graph query approaches such as model search approaches (e.g. Moogle [56]) create search indexes. However, they do not consider the maintenance of the search index when the indexed models change and do not enable arbitrary views.

Other approaches such as VIATRA 2 [6] and EMF-IncQuery [5] enable incremental graph pattern matching by mapping graph patterns to relational tuples for making use of RETE [21]. However, Hanson et al. [37] have shown that RETE networks are not optimal in all cases and generalized discrimination networks such as Gator networks [37] can perform better.

We give a detailed discussion of related work in Sec. 7.

1.2. Prerequisites

In this section, we introduce the terminology used in the remainder of this technical report. We adapt the terminology from related work about relational and graph databases.

According to Barcelo et al. [4] a *graph database* is a graph. However, for this technical report we extend this simple definition and refer to the term *graph database* as set of multiple (possibly independent) data graphs. Data graphs are complex structures that consist of nodes, edges between nodes, and attributes that belong to nodes and consist of certain attribute values. In our technical report, we adapt to the definition of Fan et al. [19].

Definition 1. A *data graph* is a directed graph $G = (N, E, f_A)$, where N is a finite set of nodes, $E \subseteq N \times N$ is a finite set of edges, in which $(n, n^{'})$ denotes an edge from node n to $n^{'}$, and $f_A(n)$ is a function such that for each node $n \in N$ $f_A(n)$ is a tuple $(A_1 = a_1, ..., A_m = a_m)$, where a_i is an attribute value, and A_i is referred to as an attribute of n, written as $n.A_i = a_i$.

We extend the definition of Fan et al. [19] and adapt a definition of Jouault et al. [44], who define that nodes and edges in the data graph consist of a type that are defined by a *reference graph*.

Definition 2. A data graph $G = (N, E, f_A)$ is associated with a *reference graph* $G_\omega = (N_\omega, E_\omega, \mu)$. The function $\mu : N \cup E \to N_\omega$ associates nodes and edges of G to nodes N_ω of G_ω.

Jouault et al. [44] also call the data graph a *model* and the reference graph a *metamodel*. Furthermore, a model must conform to its metamodel. In summary, we define graph databases as graph data that consist of multiple possibly independent data graphs with possibly different reference graphs.

Definition 3. A *graph database* is a set $G_{DB} = \{G_1, ..., G_n\}$ of possibly independent data graphs G_i with possibly different reference graphs G_{w_i}.

We distinguish data graphs stored by graph databases into two categories named database base graphs and database view graphs. A database *base graph* (base graph for short) (cf. [14]) is a data graph that conforms to a base reference graph and represents atomic data stored by graph databases. Base graphs are *not* derived from other data graphs.

Definition 4. A *base graph* is a data graph $B = (N_B, E_B, f_A)$ that conforms to a base reference graph $B_\omega = (N_{\omega_B}, E_{\omega_B}, \mu)$ with function μ associating nodes N_B and edges E_B of B to nodes N_{ω_B} of B_ω.

A database *view graph* (view graph for short) is a data graph that conforms to a view reference graph and is derived from base graphs and/or view graphs of a graph database (cf. [14]) with the help of a database view definition.

Definition 5. A *view graph* is a data graph $V = (N_V, E_V, f_A)$ that conforms to a view reference graph $V_\omega = (N_{\omega_V}, E_{\omega_V}, \mu)$ with function μ associating nodes N_V and edges E_V of V to nodes N_{ω_V} of V_ω. A view reference graph extends a base reference graph by additional nodes $N_{\omega_B} \subseteq N_{\omega_V}$ and edges $E_{\omega_B} \subseteq E_{\omega_V}$. Therefore, view graphs can consist of nodes and edges that do not conform to a node in the base reference graph.

A database *view definition* (view definition for short) describes how to derive graph data from other data graphs. Therefore, a view definition is also considered as *graph query* over a set of base graphs and/or view graphs for deriving new view graphs. Such view definitions often describe sub-queries that are required for common, complex and often stated queries by users of the graph database, later on. Note that a view definition can exploit at once multiple base and/or already derived view graphs specified by other view definitions.

Definition 6. A *view definition* d employs a *graph query* Q over a set of base graphs and/or view graphs $B_V = \{B_1, \ldots, B_n, V_1, \ldots, V_m\}$ to derive a new view graph V_{m+1}. We write $d : B_V \overset{Q}{\rightarrow} V_{m+1}$.

According to Barcelo et al. [4] *"graph patterns can naturally be viewed as [graph] queries"* for searching graph databases. Thus, *graph patterns* are employed to formulate graph queries, in practice. According to Fan et al. [19] graph patterns are graphs that define which kinds of nodes and edges belong to the subgraphs that database users wants to find in data graphs. We adapt the graph pattern definition of Fan et al. [19].

Definition 7. A *graph pattern* $P = (N_P, E_P, p_N)$ consists of a finite set of nodes N_P and finite set of edges E_P as defined for data graphs. The function p_N defines that for each node n in N_P $p_N(n)$ is a predicate of n. This predicate is defined as conjunction of atomic formulas $A\ op\ a$ with A denoting an attribute, a denoting an attribute value, and op referring to a comparison operator. The graph pattern P is associated with a reference graph $P_\omega = (N_\omega, E_\omega, \mu)$. The function $\mu : N_P \cup E_P \rightarrow N_\omega$ associates nodes N_P and edges E_P of P to nodes N_ω of P_ω.

Typically graph queries employ a graph pattern and return a set of subgraphs that match the employed graph pattern as query result. In practice, *graph pattern matching* is employed to answer graph queries in terms of graph patterns. Fan et al. [19] state that graph pattern matching *"is typically defined in terms of subgraph isomorphism"*. In case of graph databases *"it is to find all subgraphs [in the data graphs] that are isomorphic to the graph pattern"* [19].

Definition 8. A *graph query* Q employs a graph pattern P and returns all subgraphs that match graph pattern P as query result $Q(P) = \{M_1, ..., M_n\}$. We call M_i a match for graph pattern P over the set of base graphs and view graphs B_V in the graph database.

Graph query results in terms of matches for graph patterns can be either persisted as view graphs or computed on demand when a database user refers to this view graph when stating a graph query. We refer to the term *materialized view graph* when view graphs are persisted (cf. [32]) and use the term *virtual view graph* when the view graph is computed on demand, because the view graph is not persisted (cf. [14]). For that purpose, each view definition defines whether its query result is materialized or not.

Definition 9. A view definition $d \in D$ consists of a function $mat : D \rightarrow \{0, 1\}$ that describes whether its derived view graph is either materialized $mat(d) = 1$ or virtual $mat(d) = 0$.

Materialized view graphs are *consistent* with the base graphs from which they are derived when every graph query yields the same result when executed on the view graph and base graph (cf. [14]). Virtual view graphs are always consistent with the base graphs from which they are derived, because virtual view graphs are computed when the database user states a graph query that refers to this virtual view graph.

Definition 10. A view graph V is consistent with a base graph B from which view graph V is derived by a view definition that performs graph query Q, if and only if, each graph query Q over the derived view graph V yields the same query result as graph query Q over base graph B from which V is derived.

1.3. Running Example

An appropriate running example should provide large-scale graphs that enable multiple dependent and complex view definitions. Furthermore, along with large-scale graphs a real history of changes should be provided. Therefore, our running example deals with abstract syntax graphs derived from source code of open source projects and employs queries for software design pattern recovery. We use the version control history of the source code to derive real changes of the abstract syntax graphs. We have chosen abstract syntax graphs of source code, because these graphs are complex, consist of many kinds of nodes and edges, and can be easily derived from source code. We have chosen to employ the recovery of software design patterns as queries and views, respectively. As Gamma et al. [23] conveys certain software design patterns are suited for different software design challenges. The recovery of such software design patterns is a difficult task due to the number of different software design patterns, differences in programming languages and since even different implementation variants exist. Therefore, maintaining occurrences of such software design patterns in abstract syntax graphs in terms of view graphs can be beneficial when querying software design patterns in abstract syntax graphs. These view graphs are derived from base graphs (i.e. abstract syntax graphs of source code) with the help of view definitions.

In our running example, we setup a graph database that consists of abstract syntax graphs derived from source code as well as view graphs that contain graph pattern matches that describe the occurrences of software design patterns within these abstract syntax graphs. The right hand side of Fig. 1.1 depicts dependencies between different view definitions. The ellipses in Fig. 1.1 denote view definitions and the edges describe the dependencies between them. Each of these view definitions derives a view graph that contains markers for graph pattern matches, which can be reused by multiple dependent view definitions.

The left hand side of Fig. 1.1 shows that the `Composite` software design pattern is characterized by two structural properties. First, the `Composite` class is a specialization of the `Component` class. And second, the `Composite` class owns a to-many association with the `Component` class as target. Both structural properties are required pre-conditions for the recovery of `Composite` software design patterns. Thus, view definitions are required that describe how to derive view graphs about `Generalizations` and `One-To-N-Associations`. These view definitions are reused by the view definition that derives a view graph about the occurrences of `Composite` design patterns. Furthermore, view definitions that pre-compute knowledge about different kinds of software design patterns can be aggregated by a superior view definition that derives a view graph about the occurrences of `SoftwareDesignPatterns` in general. Afterwards, the derived view graphs keep ready answers for queries about employed software design patterns without time-consuming on-demand look-ups in the base graph.

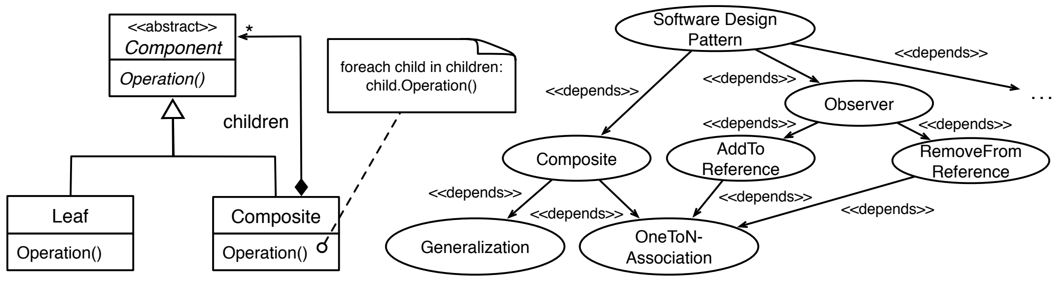


Figure 1.1.: Left: UML class model specification of the Composite pattern (cf. [23])
Right: Dependency graph about involved view definitions

However, the derived view graphs must be maintained before queries are answered to be consistent with their base graphs. For example, when a new inheritance relationship between two classes is added, the `Generalization` view graph must be updated by adding the new generalization, because the new inheritance relationship causes a new generalization between both classes. Analogously, when an inheritance relationship between two classes is removed, the `Generalization` view graph must be updated by removing the previously detected generalization, because the generalization does not exist anymore in the base graphs of this view graph. Furthermore, the inheritance relationship between both classes can change. In such cases, the view graph for generalizations need to be revised by checking whether the stored generalizations are still consistent due to the modification of the inheritance relationship in the abstract syntax graph and must be updated accordingly by preserving or deleting previously detected generalizations.

Note that changes to base graphs cause updates to view graphs that cause required updates to dependent view graphs as well. According to our running example, updates of the `Generalization` view graph may cause required updates to the `Composite` view graph. For example, generalizations that are added to the `Generalization` view graph may cause additions of Composite design patterns to the `Composite` view graph, while generalizations that are removed from the `Generalization` view graph may cause deletions of Composite design patterns from the `Composite` view graph. Furthermore, revised generalizations in `Generalization` view graphs as well as revised one-to-N associations in `One-to-N Association` view graphs need to trigger the revision of the `Composite` view graph, because stored Composite design patterns may become invalid due to changed generalizations or one-to-N associations.

We give a detailed description of the employed view maintenance algorithms including negative application conditions and recursion in Sec. 5.

1.4. Outline

The remainder of this technical report is organized as follows. In Chapter 2, we describe the needs of database users towards view maintenance for graph databases and derive requirements towards graph databases that enable such view maintenance. In Chapter 3, we introduce our approach and show how the derived requirements are addressed by our approach. In Chapter 4, we describe how view definitions can be created with our approach. In Chapter 5, we show how these view definitions enable to efficiently maintain derived view graphs. In Chapter 6, we present a case study to evaluate our approach. In Chapter 7, we compare our approach with related work and discuss the commonalities and differences. In Chapter 8, we conclude our presented approach and outline future work.

2. Needs and Requirements

In this chapter, we discuss the needs and requirements towards view maintenance for graph databases. We describe the needs of database user towards view maintenance for graph databases (Sec. 2.1). Afterwards, we derive requirements from these needs concerning graph databases that support view maintenance (Sec. 2.2).

2.1. Needs

In this section, we describe the needs database users have towards view maintenance for graph databases. We distinguish needs of database users into two categories. The first category deals with the expressiveness of the view definition language database users use to create views over base graphs and/or view graphs. The second category deals with the expectations database users have towards efficient view maintenance for graph databases.

2.1.1. Needs towards Expressiveness of View Definitions

When creating view definitions the database user wants to refer directly to the graph structure of the data. Therefore, view definitions should enable to formulate queries in terms of graph patterns (**N1**). Furthermore, view definitions may consist of sub-queries that are also required by dependent view definitions. Therefore, dependent view definitions should be able to reuse view definitions (**N2**). View definitions and dependencies between view definitions should enable to express nested graph conditions (**N3**) to enable database user to create appropriate view definitions of reasonable complexity. These nested graph conditions should include positive application conditions, negative application conditions and recursion to enable view graphs for deductive graph databases.

2.1.2. Needs towards Efficient View Maintenance

Database users should be capable of querying the database interactively using view graphs (**N4**). Furthermore, database users must retrieve the same query results when executing the query on base graphs and view graphs (**N5**). This means base graphs and view graphs must be consistent at the point in time when the query is stated. For that purpose, changes to base graphs must be propagated automatically to view graphs before queries are executed (**N6**). Note that also changes to view graphs as result of base graph changes, must be recursively propagated to dependent view graphs. However, the propagation of view graph changes to base graphs from which they were initially derived is not in the scope of this technical report. The propagation of base graph changes according to need **N6** should be possible without requiring database users to explicitly specify how to propagate changes from base graphs (resp. view graphs) to derived view graphs (**N7**).

2.2. Requirements

We distinguish requirements into two categories. The first category deals with requirements that address needs concerning the expressiveness of view definitions (see Sec. 2.2.1). The second category deals with requirements that address needs concerning the maintenance of view graphs (see Sec. 2.2.2).

2.2.1. Requirements on Expressiveness of View Definitions

Graph pattern matching is a natural way of querying graph data (cf. **N1**) and we assume for this technical report that queries over graph data are formulated in terms of graph patterns. Therefore, view definitions should enable to specify view definitions using graph patterns (**R1**).

View definitions should enable to create view graphs without copying and storing redundant information, which are also stored in base graphs, in view graphs. Thus, a lightweight mechanism is required to mark matches of graph patterns in base graphs (**R1a**).

Furthermore, queries stated in view definitions should be reusable by multiple dependent view definitions (cf. **N2**). Thus, a set of dependent view definitions should be definable (**R1b**) and dependent view definitions should be able to access nodes that play a certain role (i.e. have a certain semantic in the graph pattern match) in dependency view definitions efficiently without requiring to re-match these nodes (**R1c**). Moreover, to enable a reasonable complexity of queries formulated in terms of view definitions (cf. **N3**), interrelated view definitions should enable to formulate queries as nested graph conditions (**R2**) including positive and negative application conditions as well as recursion.

2.2.2. Requirements on Maintenance of View Graphs

The base graph and derived view graphs must be consistent when the user runs a query over view graphs (cf. **N5**). Therefore, the graph database must ensure the consistency of view graphs with their base graphs before queries are answered (**R3**). This consistency preservation can be either performed immediately when changes to base graphs occur (**R3a**) or can be deferred to the point in time when a query is stated that requires view graphs derived from changed base graphs (**R3b**). Snapshot view graphs (cf. [14]) are not in the scope of this technical report.

In general, required execution time and memory footprint introduced due to view maintenance should be minimal (**R4**). The time required to propagate changes from base graphs to view graphs should be independent of the sizes of base graphs (**R4a**) and, therefore, should scale with increasing sizes of base graphs (**R4b**) to enable interactive query session for database user (**N4**) also for large base graphs. This is especially important when deferred view graph maintenance (cf. **R3b**) is employed and the view graphs are updated when the database user states a query. Additional memory footprint required for materialized views should be proportional to the size of the view graphs (**R4c**).

The view definition must be sufficient to derive maintenance steps (**R5**) for propagating changes from base graphs to view graphs to preserve consistency between base graphs and view graphs (cf. **R3**). These maintenance steps should be optimal (**R5a**) in a sense that not more maintenance steps are performed than really required to keep base graphs and view graphs consistent. Especially, these maintenance steps should be independent of employed graph pattern matching technologies and, therefore, a generic algorithm of maintenance steps is required (**R5b**). Furthermore, these derived maintenance steps should support the complete expressiveness of view definitions (**R5c**).

3. Overview

In this chapter, we present the big picture of our approach. Fig. 3.1 depicts the components of our approach. Our approach consists of four main components: a graph database (see A in Fig. 3.1, Sec. 3.1), a view definition model (see B in Fig. 3.1, Sec. 3.2), a view maintenance engine (see C in Fig. 3.1, Sec. 3.3), and query engine (see D in Fig. 3.1, Sec. 3.4). In the following sections, we describe the role of each component by mapping the derived requirements to these components.

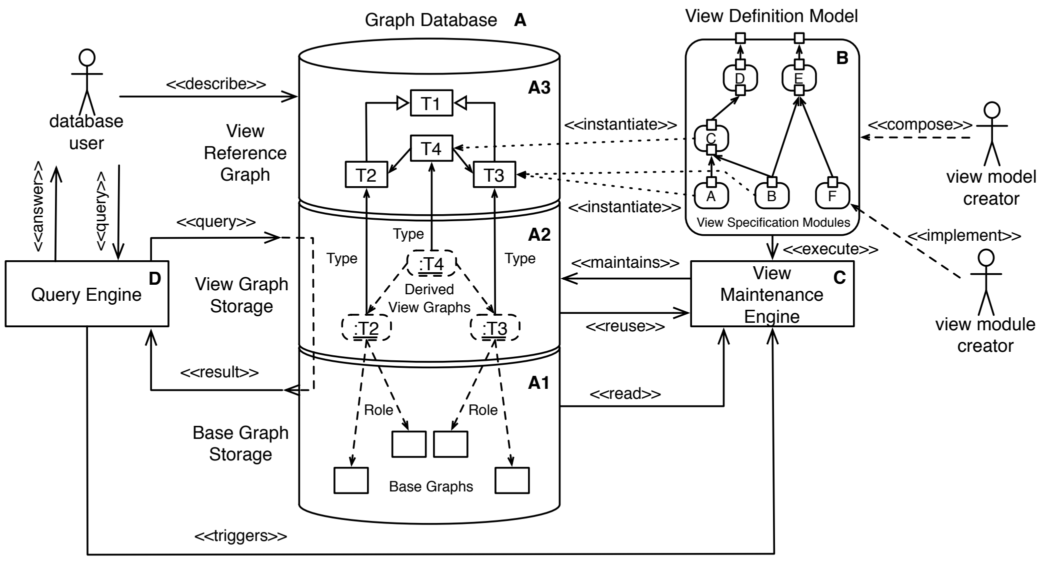


Figure 3.1.: Overview of view graph maintenance for graph databases

3.1. Graph Database

Our notion of a *graph database* consists of three parts: a base graph storage (see A1 in Fig. 3.1), a view graph storage (see A2 in Fig. 3.1), and view reference graph (see A3 in Fig. 3.1). According to definition 3 a graph database is a set of graphs that are distinguished into base graphs (see definition 4) and view graphs (see definition 5). Base graphs are stored by the *base graph storage*, while view graphs derived from these base graphs and other view graphs are stored by the *view graph storage*. View graphs consist of annotations that mark matches of graph patterns in base graphs and view graphs (cf. **R1a**) *without* copying subgraphs that match a graph pattern to view graphs. Furthermore, base graphs and view graphs consist of a reference graph (see definition 4 and 5). We neglect the reference graph of base graphs in Fig. 3.1. For example, the reference graph of source code describes that the abstract syntax graph consists of classes, attributes, references, methods, etc. The reference graph of view graphs is depicted as *view reference graph* on top of the graph database. The *view reference graph* describes which kinds of nodes and edges are part of derived view graphs.

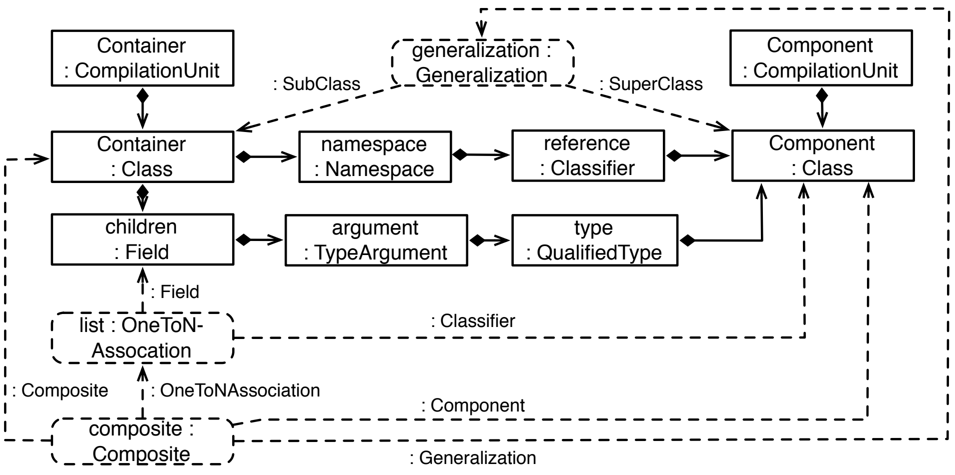


Figure 3.2.: View graph about Composite design patterns according to our running example

According to our running example, the view reference graph describes that view graphs about software design patterns consists of nodes that, e.g., describe the occurrences of Composite design patterns, and consists of edges that, e.g., describe which nodes represent the Container respectively Component class within the Composite software design pattern. Fig. 3.2 shows a view graph about Composite software design patterns. Solid rectangles and lines denote an excerpt of the base graph. Dashed rectangles and lines denote an excerpt of the derived view graph and depict annotations that mark matches of graph patterns in base graphs and/or view graphs. Fig. 3.2 depicts a detected Composite software design pattern in Java AWT. The `Composite` annotation (dashed rounded rectangle) denotes the detected Composite design pattern and depends on the `Generalization` annotation and `OneToN-Association` annotation referenced by roles (dashed lines). The `Generalization` annotation describes a specialization of the `Component` class as `Container` class. The `OneToN-Association` annotation describes that the `children` field is a to-many association with the `Component` class as target type.

3.2. View Definition

View graphs are derived with the help of a view definition model that consists of view definition modules. A *view definition module* (Sec. 3.2.1) specifies the graph patterns used to derive view graphs. A *view definition model* (Sec. 3.2.2) describes the interplay of multiple dependent view definition modules, i.e., how dependent view definition modules reuse view graphs derived by other view definition modules.

3.2.1. View Definition Module

A *view definition module* (view module for short) implements a graph query by enabling the view module creator to specify graph patterns (cf. **R1**). View modules employ graph pattern matching to find *all* matches of the specified graph pattern. As result, for each match of the graph pattern the view definition module creates an annotation that marks the match in base graphs and view graphs instead of copying matched subgraphs to view graphs (cf. **R4c**). The annotation references *all* nodes that participate in a certain match. Therefore, annotations are considered as markers for graph pattern matches. Thus, a view graph consists of a set of annotations.

According to our running example as depicted by Fig. 3.3, view modules exist that create view graphs about generalizations, multi-level generalizations, one-to-N associations, and Composite design patterns. For that purpose, the `Generalization` view module describes that it uses `Class` and `TypeReference` nodes in base graphs to create a view graph about generalizations. The `MultiLevelGeneralization` view module describes that it uses view graphs about generalizations as well as multi-level generalizations to derive view graphs about multi-level generalizations. Moreover, the `OnetoN-Association` module defines that it uses `Field` and `TypeReference` nodes to create a view graph about one-to-N associations. The `Composite` view module defines that it uses view graphs about `Generalizations` and `OnetoN-Associations` to create a view graph about Composite software design patterns.

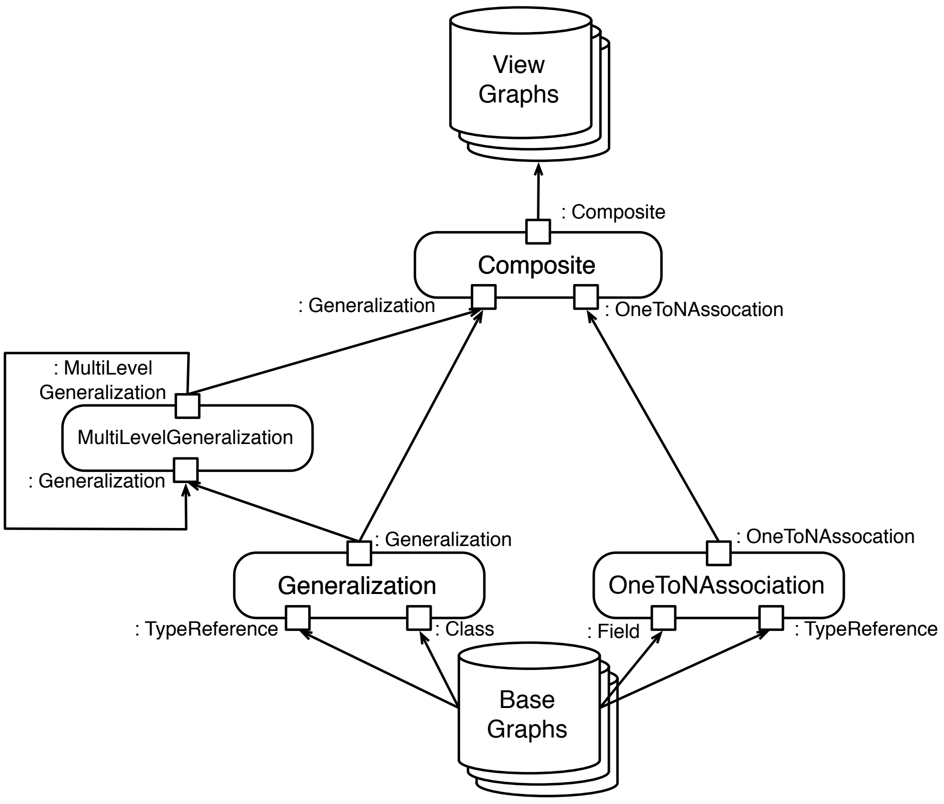


Figure 3.3.: Simplified view definition model according to our running example

3.2.2. View Definition Model

The *view definition model* (view model for short) describes the interplay of view modules, e.g., which view module uses which information derived by dependency view module (cf. **R1b**, **R1c**, and **R2**). Therefore, a view model is a directed cyclic graph of dependent view modules that is created by view model creators. Note that cyclic dependencies are permitted to support recursion.

According to our running example as depicted by Fig. 3.3, the view model describes that the `Composite` view module uses the view graphs created by the `Generalization`, `MultiLevelGeneralization` view module and `OneToN-Association` module. Note that a view module can reuse multiple view graphs of the same kind. For example, the `Composite` view module can use view graphs about generalizations and multi-level generalizations, because Composite software design patterns that use multi-level generalization are an implementation variant of the Composite design pattern as defined by Gamma et al. [23]. In our running example depicted by Fig. 3.3, single-level generalizations and multi-level generalizations are derived from base graphs by two separate modules. Note that derived multi-level generalizations are exploited recursively denoted by the dependency from the `MultiLevelGeneralization` module to the `MultiLevelGeneralization` module. In doing so, the `MultiLevelGeneralization` module is able to derive additional multi-level generalizations that are composed of other multi-level generalizations.

3.3. View Maintenance Engine

The *view maintenance engine* (view engine for short) is responsible to keep view graphs consistent with their base graphs (cf. **R3**). Initially, the view engine derives the view graphs from the base graphs and other view graphs from scratch. Afterwards, the view engine employs maintenance procedures to keep derived view graphs up-to-date with respect to their base graphs. For that purpose, the view engine takes the base graphs, already existing view graphs, and the view model with its view modules as input and adds, deletes, or updates annotations within view graphs. We distinguish three basic kinds of view maintenance procedures. All three maintenance procedures support the expressiveness of view models in terms of nested graph conditions and recursion (cf. **R2** and **R5c**).

In the following sections, we only outline the supported maintenance modes and give a complete description in Sec. 5. We distinguish naive batch mode (Sec. 3.3.1), batch mode with preservation (Sec. 3.3.2), and incremental mode (Sec. 3.3.3).

3.3.1. Naive Batch Maintenance

The naive batch maintenance procedure deletes all view graphs and creates them from scratch. This procedure perform a full recomputation of view graphs, but can be beneficial when the changes made to base graphs are large and, therefore, the additional overhead of the other maintenance procedures may not pay off. Note that annotations, which mark matches of graph patterns, change their identity when they are created from scratch and, thus complicate a post-processing of these annotations. Therefore, graph database users do not desire this naive batch maintenance.

3.3.2. Batch Maintenance with Preservation

The batch maintenance procedure with preservation processes all view graphs completely, but deletes annotations only when necessary. Thus, this maintenance procedure preserves annotations in view graphs that are still consistent with base graphs. The required maintenance steps are derived from the view modules (cf. **R5**). The maintenance procedure is independent of the employed graph pattern matching technology (cf. **R5b**), because the required maintenance steps are derived from the view module specification as shown by Fig. 3.3. In Sec. 5, we describe the concrete maintenance steps in detail.

3.3.3. Incremental Maintenance

The incremental maintenance procedure processes view graphs only partially. Which parts of the view graphs must be re-processed is derived from the modification information of base graphs. In general, only a small part of the view graphs needs to be re-processed, because heuristically the modified parts of base graphs are often much smaller than the complete base graphs. Thus, the incremental maintenance is in general much more efficient than batch maintenance (cf. **R4**). Therefore, less time is required to keep view graphs consistent with their base graphs (cf. **R4a**) in contrast to naive batch maintenance and batch maintenance with preservation. The required maintenance steps are derived from the view module specifications (cf. **R5**) and modification information of base graphs to avoid unnecessary maintenance steps (cf. **R5a**). The modification information of base graphs determine which view modules need to be re-executed to make impacted view graphs consistent with their base graphs. Similar to batch maintenance with preservation, the maintenance procedure is independent of the employed graph pattern matching technology (cf. **R5b**).

We distinguish two incremental modes to compute the input for view modules when maintaining view graphs. Both modes take the modification information of base graphs into account to derive artifacts and annotations that are required as input by view modules to make derived view graphs consistent with changed base graphs. In incremental black box mode, the artifacts and annotations required by a view module for maintenance are derived from the view module specification without taking the graph pattern specified within a view module into account. In incremental white box mode, the artifacts and annotations required by a view module for maintenance are derived from the graph pattern specified within the view module. When taking the concrete graph pattern into account the number of artifacts and annotations required as input for view modules can be decreased to reduce the effort of the view module to maintain the derived view graph. The number of required artifacts and annotations can be reduced, because edges between nodes in the graph pattern can be considered to compute relevant artifacts and annotations by only considering artifacts and annotations as relevant when they are reachable via edges that are part of the graph pattern specified within the view module, in contrast to incremental black box mode.

3.4. Query Engine

The *query engine* enables to query base graphs and view graphs, e.g., using a graph pattern matching language. The concrete query language is not in the scope of this technical report. The query engine is capable of triggering the view maintenance when deferred view maintenance (cf. **R3b**) is employed to make the view graphs required for answering certain queries consistent with their base graphs. In case immediate view maintenance (cf. **R3a**) is employed, the view maintenance engine immediately propagates the changes from base graphs to view graphs by (partially) re-executing the view model. Then, the query engine can answer queries without triggering view maintenance, because all changes to base graphs have been propagated to view graphs immediately.

4. View Definition Approach

In this section, we present our view definition approach that consists of a) a view reference graph that describes the kinds of interrelationships between nodes in base graphs and view graphs (Sec. 4.1), b) dependent view modules (Sec. 4.2) that compute matches for graph queries (Sec. 4.3), and c) a graph pattern language that enables to refer effectively to nodes that are already part of other view graphs (Sec. 4.4). Furthermore, we describe the syntax of derived view graphs (Sec. 4.5) and how nested conditions can be mapped to view models (Sec. 4.6). Finally, we discuss the fulfillment of the requirements that we describe in Sec. 2.2.

4.1. View Reference Graph

When creating view graphs over base graphs one must be aware of which kinds of nodes in base graphs and view graphs are processed to derive view graphs and which kind of information is represented within these view graphs. For that purpose, our approach enables to model a view reference graph (see definition 5) that describes which kinds of nodes in base graphs and view graphs are considered by view graphs that represent a certain kind of information (cf. **R1a**). Note that the view reference graph does not specify the graph query that defines the content of view graphs.

The left side of Fig. 4.1 shows our view reference graph metamodel, which defines the concepts of `ArtifactTypes`, `AnnotationTypes`, and `RoleTypes`, which can be used to describe view reference graphs.

An `ArtifactType` describes the type of a node in a base graph and consists of a name. We refer to a node in a base graph as artifact of a certain artifact type. Furthermore, a hierarchy of `ArtifactTypes` exists denoted by the `subTypes` (resp. `superType`) reference.

An `AnnotationType` describes the type of a node in a view graph and consists of a name. We refer to nodes in view graphs as annotations of a certain `AnnotationType`. Annotations mark graph pattern matches within base graphs and view graphs. Therefore, an annotation type describes which kind of information is represented by an annotation.

A RoleType describes which kind of artifact in a base graph or annotation in a view graph participates in an annotation. A RoleType describes in which role (resp. semantic) an artifact or annotation acts when marked by an annotation of a certain annotation type. Therefore, each RoleType references either an ArtifactType or AnnotationType to describe the type of the artifact or annotation that is annotated (cf. AnnotatedType in Fig. 4.1).

Note that subTypes (resp. superTypes) of AnnotationTypes exist and that RoleTypes are inherited to annotation sub types. Furthermore, annotation sub types can consist of additional role types.

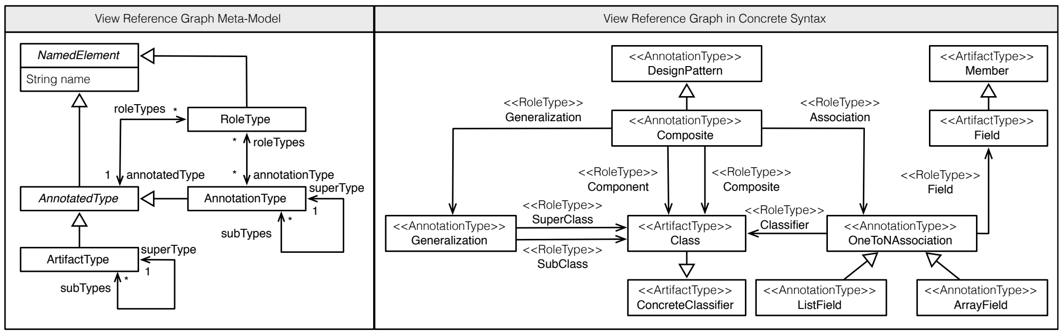


Figure 4.1.: View reference graph metamodel (left) and view reference graph in concrete syntax according to our running example (right)

The right side of Fig. 4.1 shows our running example in concrete syntax. As concrete syntax we use Unified Modeling Language (UML) class models with stereotypes. Artifact types, annotation types, and roles types are labeled with the stereotypes «ArtifactType», «AnnotationType», and «RoleType». According to our running example, abstract syntax graphs of source code contain among others ConcreteClassifiers such as Classes. Furthermore, such abstract syntax graphs contain Members of Classes such as Fields. The Composite annotation type consists of the role types Component and Composite, which reference the artifact type Class to express that a Composite annotation references one class that acts as component class and one class that acts as composite class. Furthermore, the Composite annotation type consists of the role types Generalization and Association, which reference the Generalization and OneToNAssociation annotation type to express that Composite annotations base on annotations about generalizations and one-to-N associations. The Generalization annotation type consists of the role types SuperClass and SubClass, which describe that generalization annotations reference one super class and one sub class. The OneToNAssociation annotation type consists of the role types Field and Classifier that describe which field acts as one-to-N association and which type this field has. The ListField and ArrayField annotation types are specializations of the OneToNAssociation annotation type and denote whether an association is implemented using a list or array data structure. The Field role type is inherited from the OneToNAssociation annotation type to the ListField and ArrayField annotation type.

4.2. View Modules

Our approach consists of view modules, which instantiate the view reference graph. View modules (modules for short) execute graph queries and store the query results in terms of annotations in view graphs. According to definition 8, graph queries are implemented as graph patterns. We do not make restrictions which graph pattern matching approach is employed within these modules.

The left side of Fig. 4.2 shows the view module metamodel. Each Module consists of at least two Connectors, i.e. one input connector and one output connector. We distinguish Connectors into AnnotationConnectors and ArtifactConnectors. ArtifactConnectors are input connectors, which consume artifacts of a certain ArtifactType from base graphs. Note that multiple ArtifactConnectors of different modules can consume artifacts of the same ArtifactType. AnnotationConnectors either consume or provide annotations of a certain AnnotationType. Therefore, we distinguish AnnotationConnectors into AnnotationInputConnectors and AnnotationOutputConnectors. AnnotationInputConnectors are input connectors and consume annotations. AnnotationOutputConnectors are output connectors and provide annotations. Note that multiple AnnotationConnectors of different modules can consume and provide annotations of the same AnnotationType. Furthermore, AnnotationInputConnectors can be negative (cf. isNegative attribute of AnnotationInputConnector class). A negative AnnotationInputConnector states that annotations of the annotation type as specified by the annotation input connector are used in negative manner within the view module.

The right side of Fig. 4.2 shows our running example in concrete syntax. Modules are depicted as rounded rectangles with the module name in the middle of the rectangle. Connectors are depicted as small rectangles and consist of a name and artifact type or annotation type separated by a colon. ArtifactConnectors are depicted as rectangles with a filled black rectangle in the middle. Annota-

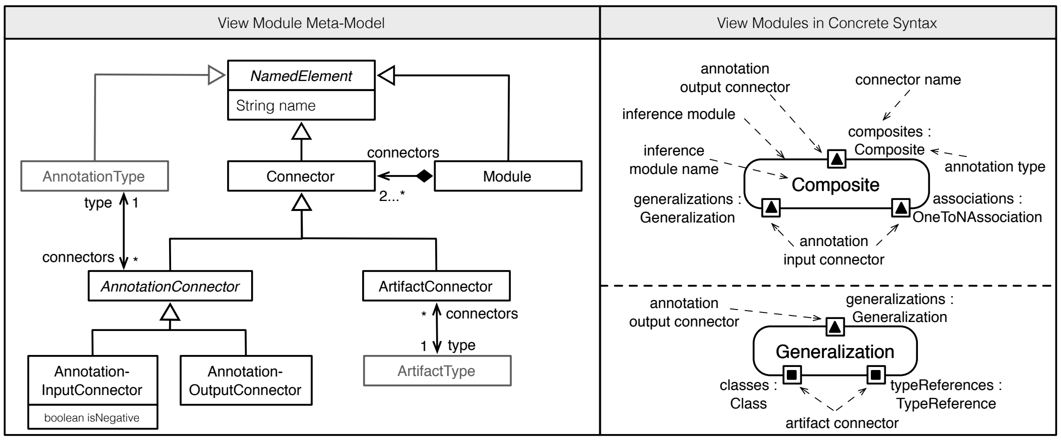


Figure 4.2.: View module metamodel (left) and view modules in concrete syntax according to our running example (right)

`tionConnectors` are depicted as rectangles with a filled triangle in the middle. `AnnotationInputConnectors` consist of a triangle that points to the module, while `AnnotationOutputConnectors` consist of a triangle that points away from the module.

The bottom right of Fig. 4.2 shows the `Generalization` module. The `Generalization` module consumes artifacts of the artifact types `Class` and `TypeReference` via the artifact connectors `classes` and `typeReferences` to produce annotations of the annotation type `Generalization` that are provided via the annotation output connector `generalizations`, afterwards. The top right of Fig. 4.2 shows the `Composite` module. The `Composite` module consumes annotations of annotation type `Generalization` and `OneToNAssociation` via the annotation input connectors `generalizations` and `associations` to produce annotations of annotation type `Composite` that are provided via the annotation output connector `composites`.

4.3. View Models

In our approach, view modules and dependencies between view modules constitute a directed cyclic graph and exchange results of graph queries in terms of annotations. The left side of Fig. 4.3 shows the view model metamodel. A `ViewModel` contains `Modules` (see Fig. 4.2), which are connected by `ModuleDependencies`. A `ModuleDependency` connects `AnnotationOutputConnectors` with `AnnotationInputConnectors` and, therefore, `ModuleDependencies` denote the exchange of annotations. Furthermore, a `ViewModel` consists of `ModelOutputConnectors`. A `ModelOutputConnector` provides the overall view graph computed by the `ViewModel`. For that purpose, `ModelDependencies` connect `AnnotationOutputConnectors` with `ModelOutputConnectors`.

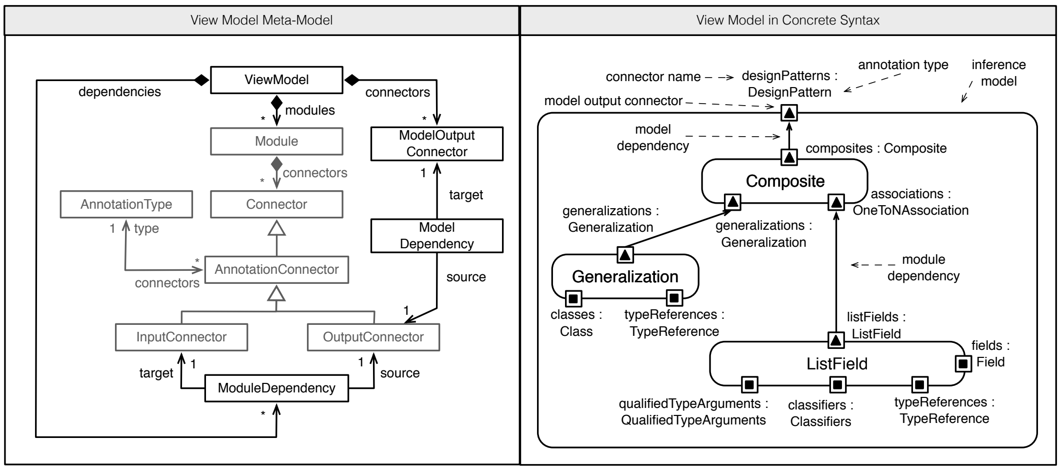


Figure 4.3.: View model metamodel (left) and view model in concrete syntax according to our running example (right)

The right side of Fig. 4.3 shows our running example in concrete syntax. `ViewModels` are depicted as rounded rectangles, which embed `Modules`. Similar to `AnnotationOutputConnectors` of `Modules`, each `ModelOutputConnector` is depicted as rectangle with a filled triangle pointing away from the `ViewModel` and is labeled with a name and annotation type separated by a colon. `ModelDependencies` and `ModuleDependencies` are depicted as solid lines from `AnnotationOutputConnectors` to `ModelOutputConnectors` and `ModuleOutputConnectors`, respectively. The arrow end denotes the dependent connector (resp. module). `ModuleDependencies` must connect `AnnotationOutputConnectors` and `AnnotationInputConnectors` of equal annotation (super) types. Well-formedness considerations of view models are not in the scope of this technical report.

4.4. View Query Language

Our approach provides a default graph pattern language to enable an easy implementation of graph queries in terms of graph patterns within modules. We refer to this graph pattern language as *view query language*. Note that also other graph pattern languages can be used.

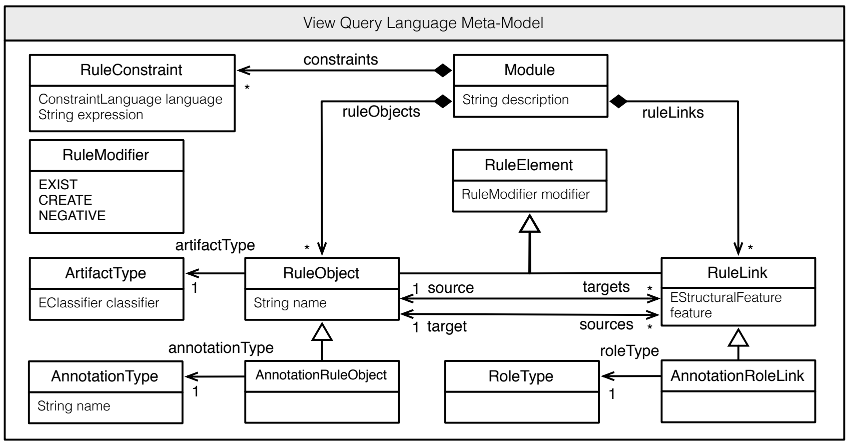


Figure 4.4.: View query metamodel

Fig. 4.4 shows the metamodel of our view query language. Our view query language consists of `RuleObjects` denoted as solid rectangles and `AnnotationRuleObjects` denoted as dashed rounded rectangles. `RuleObjects` match `Artifacts` in base graphs, while `AnnotationRuleObjects` match `Annotations` in view graphs. The label of `RuleObjects` and `AnnotationRuleObjects` consists of a unique name (omitted in Fig. 4.5) and `ArtifactType` or `AnnotationType` separated by a colon. Solid edges between `RuleObjects` depict `RuleLinks` and dashed lines between `AnnotationRuleObjects` and `RuleObjects` (resp. `AnnotationRuleObjects`) depict `AnnotationRoleLinks`. The label of `RuleLinks` denotes the name of an `EStructuralFeature` (reference), while the label of `AnnotationRoleLinks` consists of a unique name (omitted in Fig. 4.5) and role type name separated by a colon. `RuleObjects`, `AnnotationRuleObjects` and `AnnotationRoleLinks` consist of a `RuleModifier` that describes whether an `Artifact`, `Annotation`, or `Role` must already exist, must *not* exist, or will be created. `Annotations` and `Roles` that will be created are labeled by '++' (CREATE modifier). `Artifacts`, `Annotations`, and `Roles` that must *not* exist are crossed out (Negative modifier). Artifacts, Annotations, and Roles that must exist are not labeled (Exists modifier).

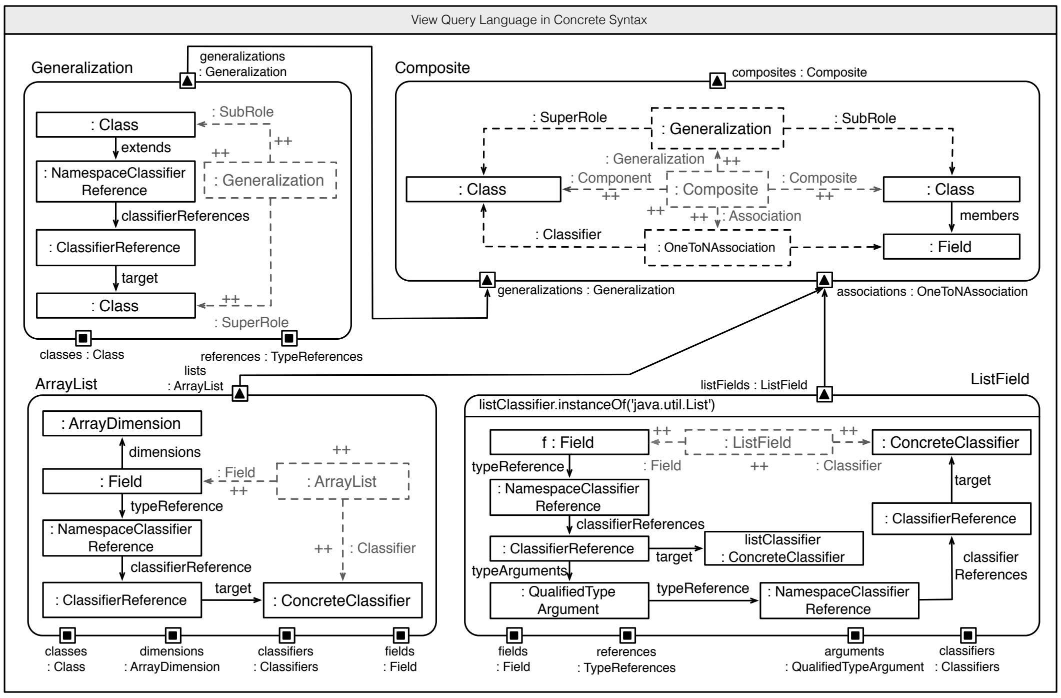


Figure 4.5.: Module implementations for our running example (simplified)

Fig. 4.5 shows (simplified) module implementations according to our running example using our view query language. According to our running example, the module Generalization on top left of Fig. 4.5 specifies the pattern graph that identifies an inheritance between two classes in an abstract syntax graph (ASG). A chain of NamespaceClassifierReference and ClassifierReference that references the super class of a class defines an inheritance between two classes. Namespace-ClassifierReferences and ClassifierReferences are subtypes of TypeReferences. Therefore, the module connectors specify that Classes and TypeReferences are required to identify inheritances in the data graph. When the generalization pattern matches, a Generalization annotation is created that references the subordinate class via the role of type SubRole and the superordinate class via the role of type SuperRole. The RuleObject of type NamespaceClassiferReference and ClassifierReference are not explicitly referenced by the generalization annotation via a role, because they do not act in a role with a certain semantic. However, implicit (invisible) roles between the matched artifacts of type `NamespaceClassiferReference` and `ClassifierReference` are created, because these artifacts are also part of the annotation and must trigger a revision of the `generalization` annotation when they are modified or deleted. As rule of thumb, each `RuleObject` and `AnnotationRuleObject` with Exists modifier that is not explicitly referenced by a role is implicitly referenced by an invisible role that belongs to the `AnnotationRuleObject` with Create modifier.

The modules `ArrayField` and `ListField` implement the graph patterns for identifying 1:N references. Both modules create annotations that reference the fields that act as 1:N references and the classifiers that act as the target classifiers of the reference. Both modules implement different graph patterns, but both identify `OneToNAssociations` that is the super type of the `ArrayField` and `ListField` annotation types (see Fig. 4.1). The `Composite` module specifies how to combine `Generalization` and `OneToNAssociation` annotations to detect `Composite` software design patterns in an ASG. The `Composite` module uses `Generalization` and `OneToNAssociation` annotations as context. The super class must be the target type of an 1:N reference and this 1:N reference must be a member of the sub class of this super class. Then, the super class acts in a `Component` role and the sub class acts in a `Composite` role. Furthermore, the `Generalization` annotation and `OneToNAssociation` are explicitly referenced via roles, since they are pre-conditions to identify `Composite` design patterns and one may want to access these annotations in dependent modules effectively by traversing these role.

Note that due to the view reference graph dependent modules do not need to know the concrete graph pattern implemented by dependency modules and, thus, modules can be considered as black boxes.

4.5. View Graphs

The annotations computed by modules and the dependencies between annotations via roles constitute a directed acyclic graph. The left side of Fig. 4.6 shows the view graph metamodel. The view graph metamodel depicts that `Annotations` have a certain `AnnotationType`. Annotations mark matches of graph patterns. Since `AnnotationTypes` consist of `RoleTypes` (see Fig. 4.1), each `Annotation` must consist of `Roles` of certain `RoleTypes`. Each `Role` annotates an `Artifact` or `Annotation` of the corresponding `ArtifactType` or `AnnotationType` as described by the `RoleType` in the view reference graph (see Fig. 4.1). Note that each `Module` knows which annotations it has created and that each annotation knows by which module it has been created.

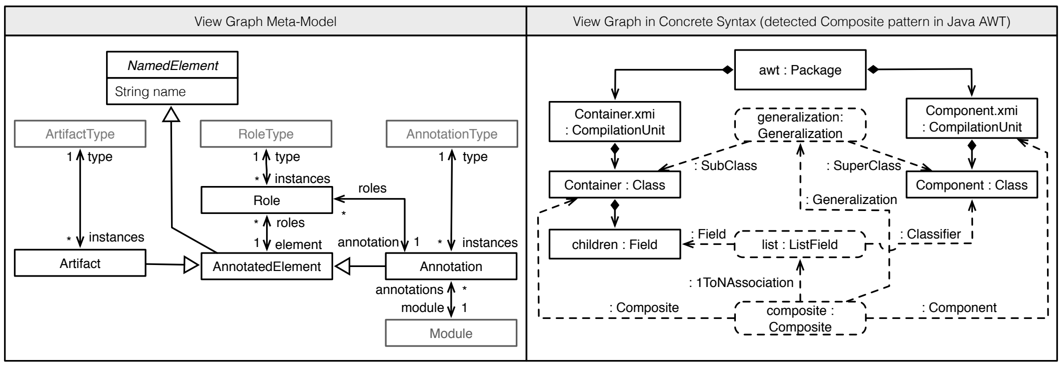


Figure 4.6.: View graph metamodel (left) and view graph in concrete syntax according to our running example (right)

The right side of Fig. 4.6 depicts the concrete syntax of annotations, roles, and artifacts according to our running example. Annotations are depicted as dashed rounded rectangles with the annotation name and annotation type in the middle of the rectangle separated by a colon. Roles are shown as dashed lines with the name of its RoleType attached. The arrow end of Roles denotes the annotated Artifact or Annotation. Solid rectangles depict Artifacts with the name and artifact type separated by a colon in the middle of the rectangle. Solid lines between Artifacts denote a containment hierarchy of Artifacts. The arrow denotes the child Artifact, while the rhombus denotes the parent Artifact. We neglect other references between Artifacts in base graphs, if they are not important for illustrating our running examples.

4.6. Mapping Nested Conditions to View Models

Graph conditions are graph patterns that, if satisfied, let the graph condition become true. We distinguish graph conditions into positive graph conditions and negative graph conditions. A positive graph condition becomes true if the graph pattern is satisfied. A negative graph condition becomes true if the graph pattern is *not* satisfied.

A nested graph condition is a boolean formula that consists of graph conditions. 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 (nested) graph conditions.

In the following sections, we describe how nested graph conditions can be mapped to view models. We describe the mapping of overlapping graph conditions (Sec. 4.6.1), conjunctions (Sec. 4.6.2), disjunctions (Sec. 4.6.3), positive and negative graph conditions (Sec. 4.6.4) as well as recursion (Sec. 4.6.5) to view models.

4.6.1. Mapping Overlapping Graph Conditions

Graph conditions are specified within view modules. A graph condition can be part of multiple more complex graph conditions. Therefore, graph conditions can overlap. The overlapping parts of graph conditions should be mapped to a single view module, because shared graph conditions should be matched only by one view module. Then, dependent view modules can reuse matches of these graph conditions to avoid unnecessary additional computations of matches for the same graph pattern. The reuse of matches for graph patterns is mapped to multiple outgoing module dependencies with different view modules as target.

Fig. 4.7 shows graph condition $C(x,y)$ with $x \in X$ and $y \in Y$. The graph condition $C(x,y)$ is specified as directed edge *toY* between artifacts of type X and Y. The graph condition $C(x,y)$ is reused by graph condition $A(C(x,y),z) \equiv C(x,y) \wedge A'(y,z)$ with $z \in Z$ and $B(C(x,y),w) \equiv C(x,y) \wedge B'(y,w)$ with $w \in W$. The graph condition $A'(y,z)$ is specified as directed edge *toZ* between artifacts of type Y and Z. The graph condition $B'(y,w)$ is specified as directed edge *toW* between artifacts of type Y and W.

4.6.2. Mapping Conjunctions

When at least two graph conditions are combined in terms of a conjunction, three general options exist to express this conjunction in view models. Fig. 4.8 shows three alternatives for encoding the graph condition $AB(x,y,z) \equiv A(x,y) \wedge B(y,z)$ with $x \in X$, $y \in Y$, and $z \in Z$. The graph condition $A(x,y)$ is specified as an edge *toY* between artifacts of type X and Y, while graph condition $B(y,z)$ is specified as an edge *toZ* between artifacts of type Y and Z. Either the conjunction is specified within one single view module AB (see Fig. 4.8 top left), is split up into two sequential view modules A and AB' (see Fig. 4.8 top right), or is split up into three view modules A, B, and AB'' (see Fig. 4.8 bottom). The single view module approach specifies the graph condition $AB(x,y,z)$ as one complete graph

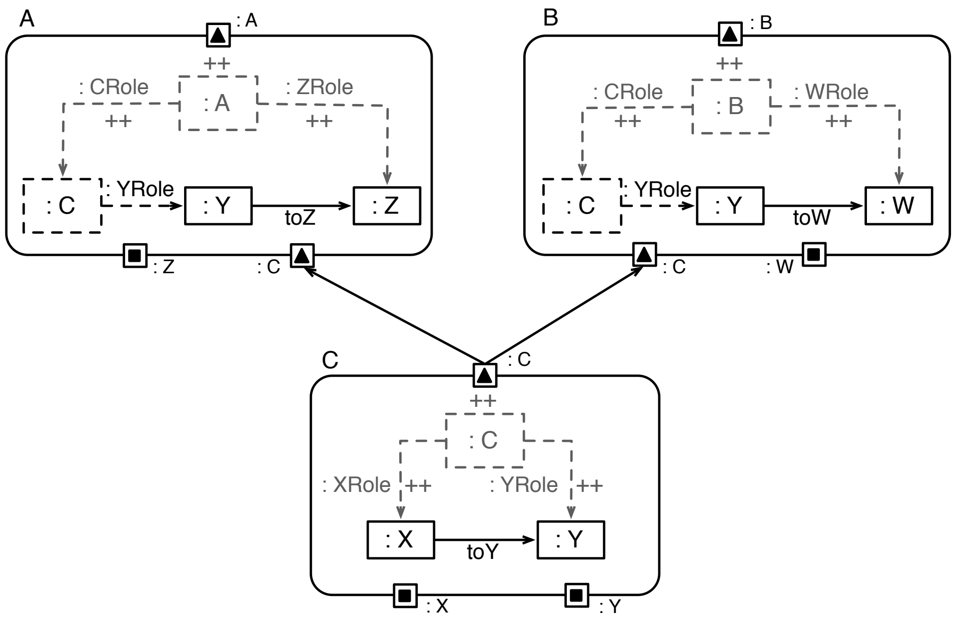


Figure 4.7.: Overlapping conditions: The graph conditions specified by view module A and B overlap in graph condition C

pattern. The sequential view module approach specifies the graph condition $A(x, y)$ and graph condition $AB'(A(x, y), z) \equiv AB$ in two separate modules. The actual conjunction of graph condition $A(x, y)$ and $B(y, z)$ is specified in the view module AB'. Note that this alternative could search for the graph condition $B(y, z)$ first and a view module BA could specify the conjunction of both graph conditions $BA(x, B(y, z)) \equiv AB$, afterwards. The alternative with the three view modules A, B and AB'' specifies the graph conditions $A(x, y)$ and $B(y, z)$ separately and the actual conjunction $AB''(A(x, y), B(y, z)) \equiv AB$ is specified by the view module AB''.

4.6.3. Mapping Disjunctions

When at least two graph conditions are combined in terms of a disjunction, one single mapping exists to express this disjunction in view models. Fig. 4.9 shows the mapping for the graph condition $AB(w, z) \equiv A(w, z) \vee B(w, z)$ with $w \in W$ and $z \in Z$. The graph condition $A(w, z)$ is specified as an edge toZ between artifacts of type $X \subseteq W$ and Z, while the graph condition $B(w, z)$ is specified as an edge toZ between artifacts of type $Y \subseteq W$ and Z. Thus, graph condition AB describes that

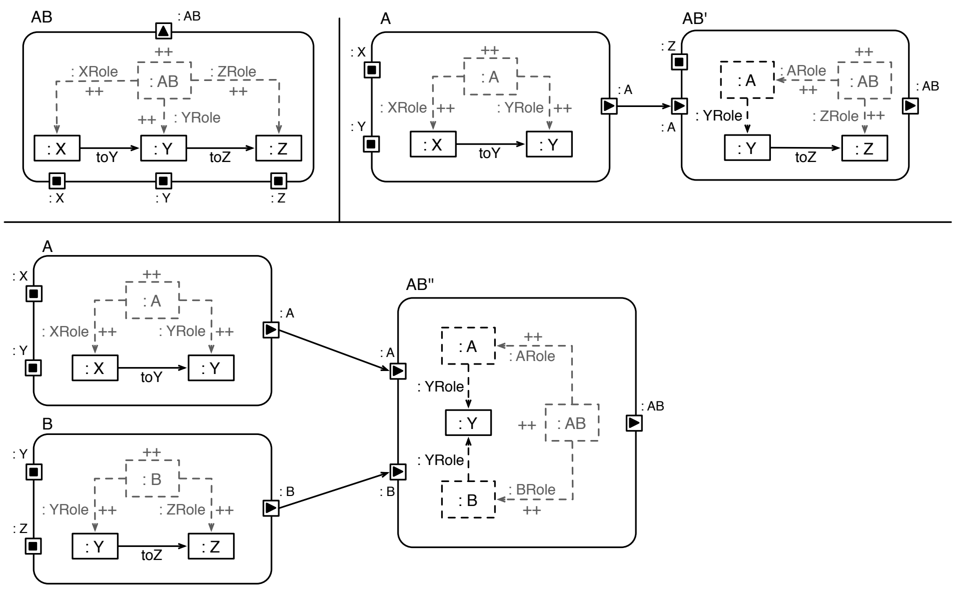


Figure 4.8.: Conjunction mapping: Single view module AB (top left), two split view modules A and AB' (top right), and three split view modules A, B and AB'' (bottom)

an artifact of type Z is reachable via a directed edge *toZ* from artifacts of type X or Y. Note that the artifact type W is the artifact super type of artifact type X and Y.

The view module A specifies graph condition $A(w, z)$, while the view module B specifies graph condition $B(w, z)$. The view module A (resp. B) provides annotations of type A (resp. B) that state that graph condition A (resp. B) is satisfied. Note that the annotation types A and B are annotation subtypes of annotation type C. The view module AB specifies the graph condition AB by describing that artifacts of type Z must consist of an annotation of type C, i.e. annotation type A or B.

4.6.4. Mapping Negative Graph Conditions

In this section, we describe how simple negative graph conditions and complex negative graph conditions are mapped to view models. We refer to the term simple NAC as graph condition that consists of one single `RuleObject` with negative modifier that must be connected to a PAC. We refer to the term complex NAC as graph condition that consists of more than one `RuleObject` with negative modifier that must be connected to a PAC. Note that complex NACs must be encoded as negated `AnnotationRuleObject` in our approach.

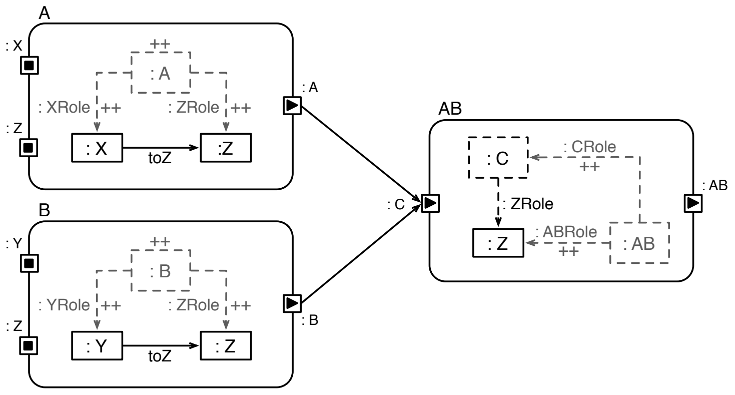


Figure 4.9.: Disjunction mapping: View modules AB encodes disjunction of graph condition A and B

Simple Negative Graph Conditions
Fig. 4.10 shows the graph condition $A(x, y)$ with $x \in X$ and $y \in Y$. Graph condition $A(x, y)$ specifies that an artifact of type X must *not* be connected to an artifact of type Y via edge *toY*. The annotation describes that an artifact of type X is *not* connected to an artifact of type Y via edge *toY*.

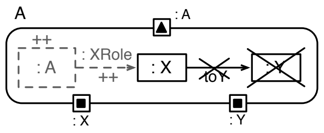


Figure 4.10.: Mapping of simple NACs: negation of artifact

Complex Negative Graph Conditions
Fig. 4.11 shows the graph condition $B(x) \equiv \exists x \neg \exists y, z : A(x, y, z)$ with a complex NAC denoted by the crossed out artifacts of type Y and Z and the edge *toZ* in between. The graph condition B specifies that artifacts of type X must *not* be connected to artifacts of type Y that are connected to artifacts of type Z via an edge *toZ*.

First, in our approach all negations are removed from the complex NAC and the resulting PAC is maintained within view graphs. In our example, the graph

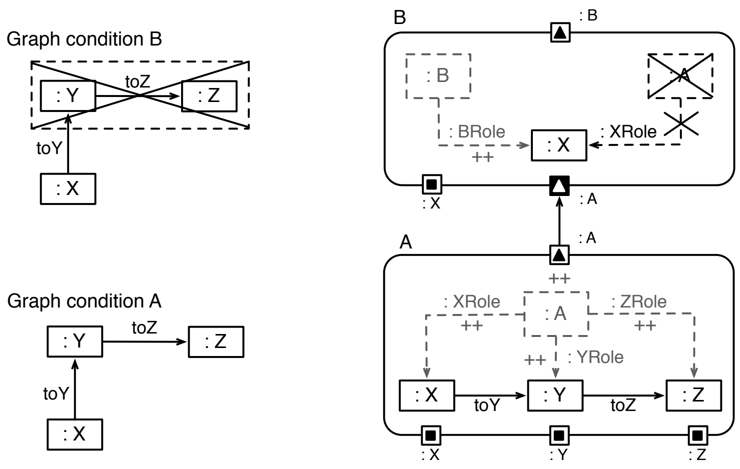


Figure 4.11.: Mapping of complex NACs: negation of graph condition

condition $A(x,y,z)$ with $x \in X$, $y \in Y$, and $z \in Z$ is maintained by view module A. Graph condition A describes that artifacts of type X are connected to artifacts of type Y via edge *toY* and that these artifacts of type Y are connected to artifacts of type Z via edge *toZ*. Afterwards, the `AnnotationRuleObject` that represents matches for this PAC is negated to describe that the PAC must not be fulfilled for certain artifacts in base graphs or annotations in view graphs. Note that the dependent view module B specifies in terms of a negated annotation input connector (denoted by inverted color scheme) that it uses annotations of type A in negative manner. In our running example, view module B specifies that artifacts of type X must *not* be part of annotations of type A denoted by the negated `AnnotationRuleObject` named A and negated `AnnotationRoleLink` named *XRole*. Thus, artifacts of type X are not part of matches as specified by view module A and, therefore, are not connected to artifacts of type Y via edge *toY* that are connected to artifacts of type Z via edge *toZ*.

4.6.5. Recursion

Recursive graph conditions are mapped to rule dependencies that constitute cycles in view models. Furthermore, an annotation sub type is required that describes recursion steps.

Fig. 4.12 shows the mapping of the graph condition $A(A(\ldots A(B(x,x),x)\ldots),x)$ with $x \in X$. The graph condition matches every path of artifacts of type X that are connected by a direct edge *toX*. The view module B defines the start of the recursion by matching the graph condition $B(x, x)$ that is specified as direct edge *toX* between artifacts of type X. The view module A defines the recursion step by matching graph condition $A(A(\ldots, x), x)$, i.e. for every path of artifacts of type X that are connected by a direct edge *toX* the graph condition matches if there is an additional edge *toX* targeting an artifact of type X. Note that annotation type B is the annotation super type of annotation type A as depicted by Fig. 4.12. Therefore, view module A consumes annotations of type B and its sub types such as annotation type A. Moreover, annotations of type A consist of a role of type *BRole* that enables to reference annotations of type B as part of the match.

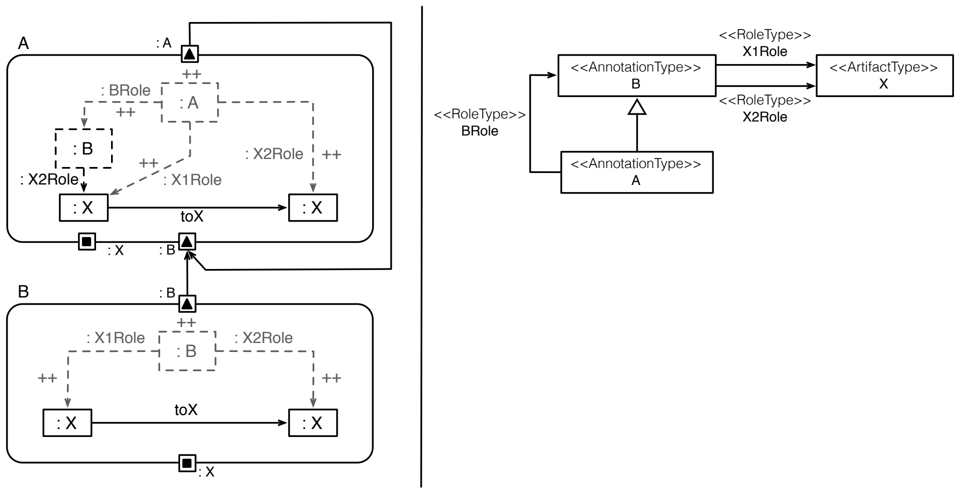


Figure 4.12.: Mapping of recursion: creation of annotations that may lead to the creation of annotations of the same type

4.7. Discussion

In this section, we discuss which concepts of our view definition approach fulfill which requirements. Our view query language (see Sec. 4.4) enables to specify graph queries in terms of graph patterns (cf. **R1**). The view module connectors and dependencies (see Sec. 4.2 and Sec. 4.3) enable the reuse of already derived view graphs (cf. **R1b**) and the nesting of positive and negative graph patterns including recursion (cf. **R2**). Furthermore, role types (see Sec. 4.1) enable to refer to artifacts and annotations referenced by annotations stored by other view graphs (cf. **R1c**).

It is up to the view module and model creators to find an appropriate decomposition of graph patterns into view modules. For example, it is a good view model design when shared sub-graph patterns are outsourced to single view modules that are reused by multiple dependent view modules, because then re-matching the same graph sub-patterns is avoided, reuse of matches is enabled (cf. **R1b**) and, therefore, also leads to a maintenance performance improvement. However, this may increase the memory footprint, because an additional view graph is required to maintain annotations that represent matches for these graph sub-patterns. Finding an optimal view model concerning memory footprint and query response time is not in the scope of this paper.

5. Efficient and Scalable View Graph Maintenance

In this chapter, we describe the view graph maintenance procedures in detail. We aim at maintaining graph query results, i.e. graph pattern matches, in terms of annotations efficiently and scalable. We refer to the maintenance of annotations as creation of missing annotations, revision of suspicious annotations, and deletion of obsolete annotations when base graphs change. This maintenance must be performed efficiently (cf. **R4a**), i.e. only impacted annotations should be revised instead of all annotations. Furthermore, this maintenance of annotations must scale (cf. **R4b**), i.e. the time required to perform the maintenance should not depend on the size of base graphs.

We distinguish between different maintenance modes. In batch mode the complete base graph is processed. We distinguish the batch mode into naive batch mode and batch mode with preservation. In naive batch mode (Sec. 5.2), when base graphs change all annotations stored in view graphs are deleted and created from scratch again without maintaining already existing annotations. In contrast to naive batch mode, in batch mode with preservation (Sec. 5.3) *all* annotations stored in view graphs are revised before the complete base graph is processed from scratch again. This revision of annotations preserves the identity of annotations in contrast to naive batch mode. The incremental mode exploits change information of base graphs for creating missing annotations, revising suspicious annotations, and deleting obsolete annotations. We distinguish the incremental mode into incremental black box mode (Sec. 5.4) and incremental white box mode (Sec. 5.5). Both incremental modes differ in the manner how they compute which artifacts and annotations are provided to view modules when base graphs and view graphs change. While the incremental black box mode only considers the types of artifact and annotation connectors, the incremental white box mode considers the graph pattern specified within view modules to provide a narrowed set of relevant artifacts and annotations to view modules for view graph maintenance.

The following sections are organized as follows. We describe how view models are traversed (Sec. 5.1). Afterwards, we describe the naive batch mode (Sec. 5.2), batch mode with preservation (Sec. 5.3), incremental black box mode (Sec. 5.4), and incremental white box mode (Sec. 5.5). For each maintenance mode, we consider positive application conditions (PACs), negative application conditions (NACs), and recursion individually.

5.1. Traversing View Models

In all maintenance modes, the view model is traversed bottom-up with respect to cycles of modules. In general, a view module can be executed when its dependency modules except for dependency view modules that are part of a recursion cycle have been executed before. When iterating over the set of view modules, view modules without annotation input connectors and view modules whose dependency view modules except for dependency view modules that are part of a recursion cycle have been executed already can be executed.

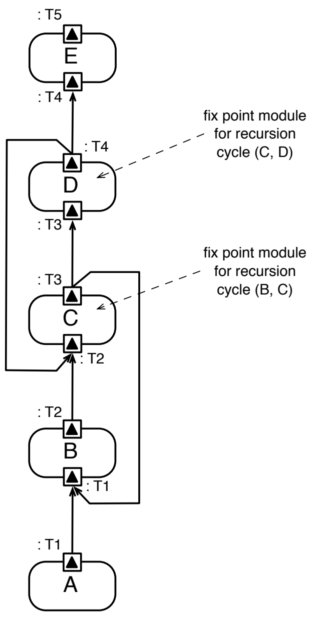

: T5
E
: T4
: T4
D
: T3
: T3
C
: T2
: T2
B
: T1
: T1
A
fix point module for recursion cycle (C, D)
fix point module for recursion cycle (B, C)

Figure 5.1.: Example of recursion cycles and fixpoint modules

If a view module is a *fixpoint view module* that created, deleted or revised annotations, an additional iteration of the recursion cycle must be performed. A fixpoint view module is a view module that is part of a certain recursion cycle and has dependent view modules that are *not* part of this recursion cycle. For example, Fig. 5.1 shows the recursion cycles C-D and B-C. Annotation type T2 is the super annotation type of annotation type T4. Annotation type T1 is the super annotation type of annotation type T3. The module C is a fixpoint module, because it is part of the recursion cycle B-C and has module D as dependent module that is not part of the recursion cycle B-C. Furthermore, the module D is a fixpoint module, because it is part of the recursion cycle C-D and has module E as dependent module that is not part of the recursion cycle C-D. If a fixpoint module did not create, delete, or update annotations, the next dependent module that is not part of the recursion cycle is executed. When a fixpoint module did not create, delete, or update annotations the *fixpoint* of the recursion cycle is reached. When a fixpoint module created, deleted, or updated annotations, the fixpoint is *not* reached and the modules in the recursion cycle are executed again, because created, deleted, and updated annotations require an additional revision of existing view graphs. Note that the execution of a recursion cycle can trigger the execution of embedded recursion cycles. For example, when the recursion cycle with module C and D is executed also the recursion cycle with the module B and C is executed when module C creates, deletes or updates annotations.

5.2. Naive Batch Maintenance

The naive batch mode processes complete base graphs and does not preserve already created and still valid annotations, because always all already created annotations are deleted from view graphs before the view model is executed again to create view graphs from scratch again (see Fig. 5.2). A description of the algorithm for naive batch maintenance in terms of pseudocode can be found in Sec. B.1.

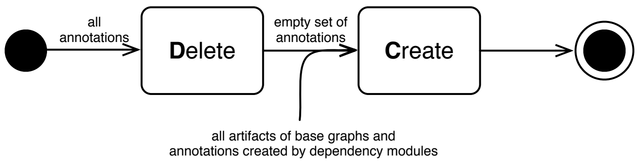

all annotations
Delete
empty set of annotations
Create
all artifacts of base graphs and annotations created by dependency modules

Figure 5.2.: Activity diagram about maintenance steps in naive batch mode

Positive Application Conditions First, the naive batch mode deletes all already existing annotations from all view graphs. Afterwards, the naive batch mode executes all view modules as described in Sec. 5.1 to ensure that all annotations are available when required by dependent view modules. For executing a view module, i.e. running a graph query, the naive batch mode determines which artifacts and annotations have to be passed to the view module by taking the artifact and annotation input connectors into account. *All* artifacts of the artifact types (and subtypes) as specified by artifact connectors of the module are passed to the module. This requires an efficient lookup of artifacts of a certain artifact type in base graphs that can be performed efficiently in our approach by traversing the `instances` reference between `ArtifactType` and `Artifact` (see Fig. 4.6). Furthermore, *all* annotations created by dependency view modules are passed via annotation input connectors to the view module. Annotations created by a certain view module can be looked up efficiently by traversing the `annotations` reference between `Module` and `Annotation` (see Fig. 4.6).

Negative Application Conditions Since the naive batch mode deletes all annotations stored in view graphs and creates all annotations from scratch, also NACs are evaluated from scratch. Therefore, the naive batch mode supports graph patterns with simple and complex NACs, intuitively.

Recursion The naive batch maintenance mode supports recursion, because according to our description how view models are executed (see Sec. 5.1), cycles of view modules are executed until the fixpoint view module reaches a fixpoint. However, the naive batch maintenance mode does not consider the deletion of *single* annotations and, thus, no recursion cycles are supported that need to delete annotations created during a previous iteration of the recursion cycle.

5.3. Batch Maintenance with Preservation

The batch mode with preservation is an extension of the naive batch mode. The batch mode with preservation employs additional maintenance steps to revise *all* already existing annotations stored in view graphs and to delete *only* obsolete annotations from view graphs. Fig. 5.3 shows the general maintenance procedure. The batch mode with preservation employs the maintenance step order: Update, Delete, and Create. A description of the algorithm for batch maintenance with preservation in terms of pseudocode can be found in Sec. B.2.

An annotation is obsolete, if at least one role of the annotation does not reference an artifact or annotation anymore. An annotation becomes obsolete when either an artifact or annotation is removed from the roles of the annotation or is set obsolete by the UPDATE step, because the graph pattern match represented by the annotation does not exist anymore. The DELETE step deletes obsolete annotations. Afterwards, the complete base graph is processed from scratch again to create missing annotations during the CREATE step. Missing annotations are annotations in view graphs that must exist due to changes of base graphs that lead to additional graph pattern matches. Thus, these additional graph pattern matches must be represented in terms of annotations in view graphs to keep derived view graphs consistent with these base graphs.

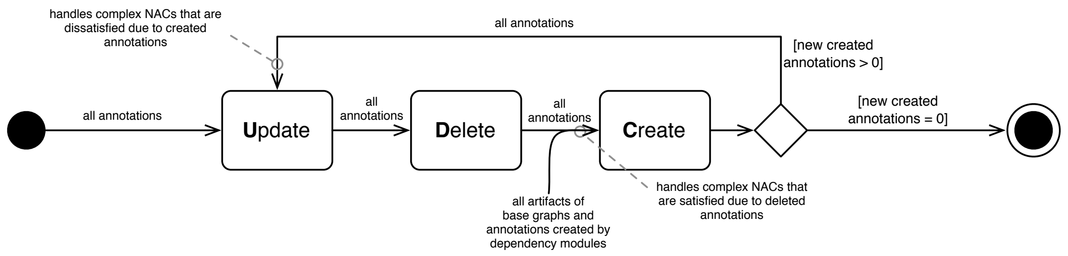

handles complex NACs that are dissatisfied due to created annotations
all annotations
[new created annotations > 0]
all annotations
Update
all annotations
Delete
all annotations
Create
[new created annotations = 0]
all artifacts of base graphs and annotations created by dependency modules
handles complex NACs that are satisfied due to deleted annotations

Figure 5.3.: Activity diagram about maintenance steps in batch mode with preservation

Positive Application Conditions The UPDATE step revises *all* already existing annotations by checking whether the matches for the graph pattern marked by these annotations still exist . For that purpose, the same module that initially created the annotation is responsible to perform the UPDATE step, because only this module is aware of the graph pattern that led to this match in terms of an annotation. If this match still exists, the annotation is preserved. Otherwise, the annotation is set obsolete by removing all annotated elements, i.e. artifacts and annotations, from the roles of the annotation. The DELETE step deletes *all* obsolete annotations by checking whether at least one role of an annotation does not reference an artifact or annotation. An annotation becomes obsolete when an artifact or annotation that is part of an annotation is deleted or the UPDATE step sets an annotation obsolete. The module that initially created the annotation is responsible for its maintenance and, therefore, deletes the annotation. Analogously to the naive batch maintenance mode, the Create step processes the complete base graph and creates all missing annotations.

The maintenance order Update, Delete, and Create guarantees that view graphs are consistent with its base graphs, because the Update step may set annotations obsolete before the Delete step is executed and the Create step does not create annotations that rely on not revised or obsolete annotations. This maintenance order leads to annotations that keep their identity and, therefore, changes in view graphs can be recognized easily.

Negative Application Conditions We distinguish the consideration of simple NACs and complex NACs. A simple NAC only consists of a single `RuleObject` with negative modifier and must be connected to a PAC. A complex NAC can consist of more than one connected `RuleObject` with negative modifier and must be connected to a PAC as well. Note that complex NACs must be specified as negated `AnnotationRuleObjects`, i.e. a view module must create annotations for matches of the non-negated graph pattern and dependent modules must use the created annotations in negated manner according to our graph condition mapping presented in Sec. 4.6.

A simple NAC can become true, when an artifact in the base graph is deleted. A simple NAC can become false, when an artifact is added to the base graph.

The Update step of batch mode with preservation considers *simple* NACs for the case an artifact was added to base graphs, because it checks *all* matches that led to annotations again and, therefore, detects simple NACs that are not fulfilled anymore due to the added artifact and sets the corresponding annotation obsolete. Thus, the subsequent Delete step deletes obsolete annotations.

The Create step of batch mode with preservation considers *simple* NACs for the case an artifact was removed from base graphs, because it searches for all matches by processing the complete base graphs. Therefore, the Create step also finds matches for graph patterns when *simple* NACs become true due to deleted artifacts.

A complex NAC can become true, when a PAC (possibly with NACs) becomes false represented by an annotation that is deleted from a view graph. A complex NAC can become false, when a PAC (possibly with NACs) becomes true represented by an annotation that is added to a view graph.

In batch mode with preservation, the Delete step removes annotations that are obsolete. Thus, the subsequent Create step can match negated `Annotation-RuleObjects`, i.e. complex NACs, which become true due to annotations that were removed during the previous Delete step. Therefore, the batch mode with preservation supports the case that a complex NAC becomes true, when a PAC (possibly with NACs) becomes false.

The case that a complex NAC becomes false, when a PAC (possibly with NACs) becomes true is handled by an *additional* UPDATE-DELETE-CREATE cycle (UDC cycle for short), because created annotations may lead to the case that complex NACs are not fulfilled anymore. Therefore, after the CREATE step an additional UPDATE step is performed that checks for all annotations in the view graphs whether an created annotation dissatisfies a complex NAC encoded as negated `AnnotationRuleObject`. When this UPDATE step sets annotations obsolete, the subsequent DELETE step removes these obsolete annotations afterwards. Since the DELETE step removes annotations, complex NACs may become satisfied. Therefore, the subsequent CREATE step may create additional annotations again. The UDC cycle is executed as long as the CREATE step creates new annotations.

Note, that this additional UDC cycle is only required when dependent view modules use annotations in negated manner (i.e., implement complex NACs), because only in this case created annotations may let become NACs false what requires an additional UPDATE step to detect annotations that become obsolete. Therefore, in terms of an optimization input connectors should specify whether they use annotations in positive or negative manner (see Sec. 4.2). Then, unnecessary UDC-cycles can be avoided when dependent view modules do not use annotations in negated manner, because created annotations cannot invalidate already existing annotations when no NACs are employed within the graph pattern of dependent view modules.

Recursion The batch maintenance mode with preservation supports recursion concerning missing annotations similar to the naive batch maintenance mode. In contrast to naive batch maintenance mode, recursion cycles are supported that need to delete annotations created during a previous iteration of the recursion cycle, because the batch maintenance mode with preservation supports the deletion of single annotations. The batch maintenance mode with preservation supports recursion for obsolete annotations, because the continuous execution of view modules in the recursion cycle until a fixpoint is reached (i.e., no annotations are deleted anymore by the fixpoint module) ensures that also annotations are deleted that become obsolete due to annotations deleted during the previous execution of the recursion cycle. The batch maintenance mode with preservation supports recursion also for annotations that are revised during UPDATE steps, because as long as the execution of the fixpoint module returns revised annotations an additional recursion cycle is performed that revises annotations that depend on the previously revised annotations.

5.4. Incremental Black Box Maintenance

In contrast to batch mode, the incremental mode uses change information of base graphs to partially reprocess the base graphs and view graphs. Analogous to batch mode with preservation, the incremental mode employs the maintenance step order: Update, Delete, and Create. Fig. 5.4 shows the general maintenance procedure. A description of the algorithm for incremental maintenance in terms of pseudocode can be found in Sec. B.3.

Created, modified, and deleted artifacts can lead to suspicious, obsolete, and missing annotations. We describe each case separately for positive and negative application conditions as well as recursion in the following sections.

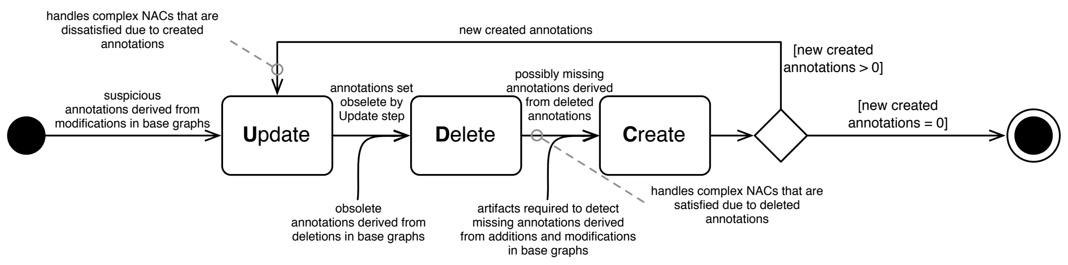

handles complex NACs that are dissatisfied due to created annotations
new created annotations
[new created annotations > 0]
suspicious annotations derived from modifications in base graphs
Update
annotations set obselete by Update step
Delete
possibly missing annotations derived from deleted annotations
Create
[new created annotations = 0]
obsolete annotations derived from deletions in base graphs
artifacts required to detect missing annotations derived from additions and modifications in base graphs
handles complex NACs that are satisfied due to deleted annotations

Figure 5.4.: Activity diagram about maintenance steps in incremental mode

5.4.1. Suspicious Annotations

Suspicious annotations are annotations that reference modified artifacts or other suspicious annotations via roles.

Positive Application Conditions Annotations are suspicious, because the modification of artifacts may lead to the fact that matches that led to annotations in view graphs do not exist anymore. An artifact is considered as modified, when attribute values of this artifact changed, when another artifact was added to or removed from a reference that is owned by this artifact, or this artifact is added to or removed from a reference owned by another artifact. Suspicious annotations that need to be revised are determined using captured modification events of base graphs. Suspicious annotations can be determined by traversing all roles in which a modified artifact acts (see `roles` reference between `AnnotatedElement` and `Role` in Fig. 4.6) to the annotation that owns the role (see `annotation` reference between

`Role` and `Annotation` in Fig. 4.6). In contrast to batch mode with preservation, the Update step only revises *suspicious* annotations in view graphs. The view module that initially created the annotation is determined. For that purpose, each annotation knows by which module it was created (see `module` reference between `Annotation` and `Module` in Fig. 4.6). Afterwards, the module checks whether the match of the graph pattern represented by the annotation still exists. If the match still exists, the annotation is preserved. Furthermore, all dependent annotations must be recursively revised as well, because dependent modules may define additional constraints for attribute values of artifacts matched by dependency modules. If the match does not exist anymore, the annotation is set obsolete by removing all artifacts and annotations from the roles of the annotation. Obsolete annotations are deleted during the Delete step as described in Sec. 5.4.2.

Negative Application Conditions The Update step considers simple NACs, because when an artifact is added to or removed from a reference, then the artifact that owns this reference as well as the artifact that is added to or removed from the reference are considered as modified and attached annotations are considered as suspicious and will be revised. The Update step for simple NACs is the same as for batch mode with preservation, but in contrast to batch mode with preservation only *suspicious* annotations are revised. Thus, annotations that are attached to the modified artifact are revised. Annotations that are not attached to modified artifacts do not have to be revised, because the graph pattern match represented by these annotations is not impacted by the modification.

Similar to the Update step of the batch mode with preservation, annotations that represent matches with complex NACs may become false due to created annotations that represent matches for PACs (possibly with NACs). Therefore, also created annotations can make annotations suspicious. But in contrast to the batch mode with preservation, created annotations that represent the matches of graph patterns with PACs (possibly with NACs) trigger the revision of suspicious annotations and, if necessary, the deletion of annotations that may let become other complex NACs true. Thus, when the Create step creates new annotations an additional UDC-cycle is performed to revise annotations that became suspicious due to these created annotations (see Sec. 5.4.3). Complex NACs that become true due to deleted annotations are detected by the subsequent Create step (see Sec. 5.4.3).

Recursion The incremental maintenance mode supports recursion for suspicious annotations, because the dependencies between annotations are used to revise dependent suspicious annotations recursively as well.

5.4.2. Obsolete Annotations

Obsolete annotations are annotations with at least one role that does not reference an artifact or annotation. Annotations are obsolete when the artifacts or dependency annotations that led to these annotations in view graphs do not exist anymore or annotations have been set obsolete by the previous UPDATE step, because the graph pattern matches represented by these annotations do not exist anymore. Therefore, obsolete annotations must be deleted.

Positive Application Conditions For PACs, all annotations are determined that are obsolete using the captured modification events. Obsolete annotations can be determined by traversing via roles in which deleted artifacts acted to annotations that are obsolete. For that purpose, the `roles` reference between `AnnotatedElement` and `Role` and the `annotation` reference between `Role` and `Annotation` is used (see Fig. 4.6). The DELETE step deletes all obsolete annotations derived from modification events of base graphs. The module that initially created the annotation is also responsible for deleting this annotation. Then, all dependent annotations of the deleted annotation become obsolete as well. Thus, all dependent annotations are recursively deleted as well by the responsible modules that initially created these annotations.

Negative Application Conditions The DELETE step considers simple NACs, because annotations that represent matches of graph patterns with simple NACs that are not fulfilled anymore are set obsolete by the previous UPDATE step and, thus, deleted during the DELETE step. Simple NACs that become true are handled by the subsequent CREATE step.

The deletion of obsolete annotations may let complex NACs become true. Complex NACs that become true due to the deletion of obsolete annotations that represented the matches of PACs (possibly with NACs) are detected by the subsequent CREATE step.

Recursion The incremental maintenance mode supports recursion for obsolete annotations, because the dependencies between annotations are used to delete dependent annotations recursively as well.

5.4.3. Missing Annotations

Artifacts that were created, deleted or modified can lead to missing annotations that need to be created. Missing annotations are annotations that must exist due to changes of base graphs to keep derived view graphs consistent.

Positive Application Conditions The CREATE step creates all annotations that must exist due to created, modified, or deleted artifacts. The captured modification events of base graphs are used to determine all created and modified artifacts. Note that it is sufficient to only consider created and modified artifacts here when only supporting simple NACs, because the creation and deletion of an artifact always implies that at least one artifact will be considered as modified when the created or deleted artifact is added to or removed from a reference. Then, the artifact that owns the reference as well as the artifact that is added to or removed from the reference are considered as modified. The CREATE step executes the modules as described in Sec. 5.1. Each module requires a set of relevant artifacts and annotations that need to be processed to create all missing annotations. We refer to this set of relevant artifacts and annotations as *scope*. To compute the scope for a module we assume that the graph pattern matched by the module is a connected graph, i.e. there is a path between every pair of nodes in the graph pattern. The scope for a module is a transitive closure, which consists of the artifacts and annotations that are a) reachable from created and modified artifacts and annotations and b) have an artifact type or annotation type as specified by the artifact and annotation input connectors. The artifact types referenced by annotation types must be considered as well as the annotation types and artifact types referenced via role types of annotation types (nested annotation types). For modules without annotation input connector the computation starts with the created and modified artifacts. For modules with annotation input connectors the computation starts with the created and modified artifacts as well as annotations created or revised by dependency modules during the current maintenance cycle. Note that only the module connectors are used for scope computation to ensure the black box property of modules.

According to our running example, the `Generalization` module requires artifacts of type `Class` and `TypeReference` (see Fig. 4.5). When we assume that a class B was added as subclass to class A in the abstract syntax graph, then a creation event for class B exists (see Fig. 5.5). The scope computation starts with class B and checks whether an artifact of type `Class` or `TypeReference` is directly reachable via a reference of class B. We assume that uni-directional references are traversable bi-directional and we consider this issue as an implementation detail that is not in the scope of this technical report. In our case a node of type `NamespaceClassifierReference` (subtype of `TypeReference`) exists and is added to the scope.

Furthermore, the node of type `NamespaceClassifierReference` references a node of type `ClassifierReference` (subtype of `TypeReference`) that is added to the scope as well. From the node of type `ClassifierReference` a class A of type `Class` is reachable and, therefore, is added to the scope. This procedure continues until the set of nodes in the scope does not change anymore. Thus, the scope consists of the classes A and B, and the namespace classifier reference and classifier reference that constitute the generalization between both classes. The other classes C and D cannot be part of a match with class B, because they are not reachable via nodes of type `Class`, `NamespaceClassifierReference`, and `ClassifierReference`. Therefore, the classes C and D are not included in the scope that is passed to the view module.

For modules with annotation input connectors also annotations are added to the scope when they are attached to relevant artifacts and consist of a relevant annotation type as specified by the annotation input connector.

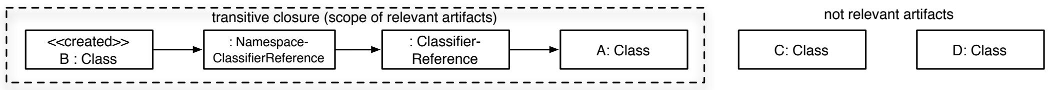


Figure 5.5.: Scope of relevant artifacts (dashed rectangle denotes scope of added class B for `Generalization` module)

Negative Application Conditions The deletion of an artifact may let become a simple NAC true. Thus, an annotation may be missing in view graphs. To keep view graphs with their base graphs consistent, this missing annotation must be created. The deletion of an artifact from a reference leads to a modified artifact that owns this reference and the removed artifact is considered as modified as well. The algorithm for scope computation introduced for PACs above considers modified artifacts as input. Therefore, simple NACs are supported by the Create step.

The creation of an annotation in view graphs may let become complex NACs encoded as negated `AnnotationRuleObject` false. Similar to batch mode with preservation, an additional UDC cycle is employed to detect annotations that must not exist anymore in view graphs due to created annotations that dissatisfy complex NACs. The Update step sets annotations that must not exist anymore obsolete and during the subsequent Delete step the responsible modules delete these obsolete annotations.

To perform also the additional UDC cycle efficiently, the scope for the additional UPDATE step is computed as follows. The newly created annotations from the previous CREATE step are used to determine the *suspicious* annotations for the subsequent additional UPDATE step. A newly created annotation references artifacts and annotations that are considered as input for the scope computation. Beginning at these start artifacts and annotations, all references and roles are traversed to find artifacts and annotations that have a type as specified by input connectors and also the output connector of the dependent module in contrast to the PAC case. When an annotation with a type as specified by the output connector of the dependent module is traversed, the annotation is added to the set of suspicious annotations. This annotation is suspicious, because it is reachable from the annotation that was created in the previous CREATE step and, thus, may dissatisfy a complex NAC implemented by dependent view modules.

Recursion The incremental maintenance procedure supports recursion for missing annotations similar to the batch maintenance mode with preservation.

5.5. Incremental White Box Maintenance

In general, the incremental white box maintenance works the same way as the incremental black box mode (Sec. 5.4). However, the main difference between the incremental black box mode and incremental white box mode is the manner how the scope is computed during the CREATE maintenance step. While the incremental black box mode only considers the types of artifact and annotation connectors, the incremental white box takes the graph pattern specified within view modules into account. When taking the specified graph pattern into account, only artifacts and annotations are added to the scope that are reachable via a reference or role that is used within the graph pattern.

According to our running example, the `Generalization` module (see Fig. 4.5) requires artifacts of type `Class` and `TypeReference`. The artifact type `TypeReference` is the artifact super type of `NamespaceClassifierReference` and `ClassifierReference`. Fig. 5.6 shows a base graph in terms of an abstract syntax graph that represents a generalization and interface implementation. The `extends` reference represents the generalization of class A by class B and the `implements` reference represents the implementation of interface I by class A. The top part of Fig. 5.6 shows the computed scope when incremental black box mode is employed. The bottom part of Fig. 5.6 shows the computed scope when incremental white box mode is employed.

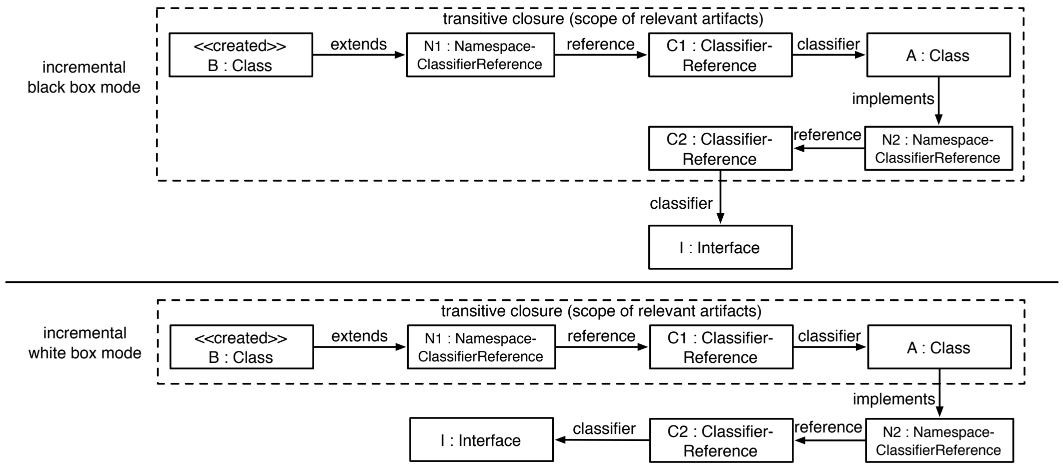

incremental black box mode
transitive closure (scope of relevant artifacts)
<<created>> B : Class
extends
N1 : Namespace-ClassifierReference
reference
C1 : Classifier-Reference
classifier
A : Class
implements
C2 : Classifier-Reference
reference
N2 : Namespace-ClassifierReference
classifier
I : Interface
incremental white box mode
transitive closure (scope of relevant artifacts)
<<created>> B : Class
extends
N1 : Namespace-ClassifierReference
reference
C1 : Classifier-Reference
classifier
A : Class
implements
I : Interface
classifier
C2 : Classifier-Reference
reference
N2 : Namespace-ClassifierReference

Figure 5.6.: Comparison of computed scope for incremental black box mode (top) and incremental white box mode (bottom)

According to our running example, we consider the `Generalization` view module as depicted by Fig. 4.5. The graph pattern specified by the `Generalization` module shows that the crucial property to detect an inheritance relationship between two classes is the `extends` reference owned by a sub class. Note that the `implements` reference owned by the class that implements an interface is *not* relevant to detect an inheritance between two classes.

When employing the scope computation of the incremental black box mode, also the `implements` reference is traversed to collect all reachable artifacts that are of artifact (sub) type `Class` and `TypeReference`, because the incremental black mode only takes the types of artifacts and annotation connectors into account.

When employing the scope computation of the incremental white box mode, the `implements` reference is *not* traversed, because the scope computation takes the actual graph pattern into account and determines that all artifacts that are *only* reachable via `implements` references cannot be part of graph pattern matches that represent inheritances between classes. Thus, the incremental white box mode narrows the scope of artifacts and annotations that can be part of graph pattern matches due to modifications of base graphs. Note that the support of recursion and negative application conditions is not impacted by the narrowing of the scope.

5.6. Discussion

The batch mode with and without preservation does not consider change information of base graphs. Thus, they are inefficient when annotations need to be revised in case base graphs change. For naive batch mode all nodes in the base graph (#N) are processed (O(#N)). For batch mode *with* preservation *all* already existing annotations (#A) are revised and all nodes in the base graph (#N) are processed afterwards. This revision of annotations preserves the identity of annotations in contrast to the naive batch mode, but requires slightly more effort (O(#N + #A)). In contrast to the batch mode, the incremental mode is efficient, because only modified parts of base graphs are reprocessed. Therefore, the time required to perform the incremental mode mainly depends on the number of nodes referenced by captured modification events (#ΔN). In general, the number of nodes referenced by captured modification events (#$\Delta N << N$) is heuristically much smaller than the number of all nodes in base graphs. Thus, in general the effort required for incremental mode (O(#ΔN)) is much smaller than the effort required for batch mode (O(#N) or O(#N + #A), respectively). But, the actual time required for incremental mode also depends on the modules that are re-executed, because modules may have different computational complexities. The incremental mode preserves the identity of annotations as well.

The scope computed during the CREATE step is not optimal for incremental black box mode, because modules are considered as black boxes and, therefore, the implemented pattern graph is unknown. Thus, one cannot know whether the artifacts and annotations contained by the computed scope will lead to a match and are worth to be processed or not. The incremental white box mode narrows the computed scope, but is also not optimal, because additional attribute constraints are not considered. But, from a theoretical perspective the computed closure for incremental white box mode (#$scope_{white}(\Delta N)$) is in general smaller than the closure computed by the incremental black box mode (#$scope_{white}(\Delta N) <$ #$scope_{black}(\Delta N)$).

6. Evaluation

In this chapter, we evaluate the performance of our maintenance modes by measuring the time required to perform the maintenance in comparison to our other maintenance modes. A memory footprint comparison is not required, because all view maintenance strategies create the same view graphs of equal size. However, a measurement of the view graph memory footprint is required to evaluate the additional memory required by view graphs. The view graph size is independent from the view maintenance strategies. We describe our evaluation setup (Sec. 6.1), present the evaluation results (Sec. 6.2), discuss the evaluation results (Sec. 6.3), and comment on the validity of our evaluation (Sec. 6.4).

6.1. Evaluation Setup

In the scope of our evaluation is the comparison of our different view maintenance modes for graph databases. For that purpose, we require large-scale base graphs, arbitrary modifications of these base graphs, and several (dependent) view modules that derive view graphs from these base graphs.

We selected to use abstract syntax graphs (ASGs) of Java source code as large-scale graph data, because arbitrary open source Java project exists that consist of a source code version control repository that contain a history of real changes. Furthermore, these abstract syntax graphs embody information about employed software design pattern as described by Gamma et al. [23] as well as information about where in the source code refactorings should be employed as described by Fowler et al. [22].

In the following sections, we report on how we derived large-scale base graphs and use the change history of open source projects to modify these base graphs (Sec. 6.1.1). Furthermore, we sketch the employed view model (Sec. 6.1.2).

6.1.1. Graph Data and Graph Changes

For our evaluation, we used graph data examples derived from open source Java projects. For that purpose, we parsed Java source code into Java models using Jamopp[1] (Jamopp models for short). These Jamopp models represent the abstract syntax graph of Java source code in an UML model-like manner.

Since parsing Java source code for deriving Jamopp models is a time-consuming task, we pre-processed the Java source code to derive Jamopp models beforehand. We stored the Jamopp models as XMI models with resolved cross references between different Jamopp models. Note, that some cross-references might not be resolved due to missing Java libraries required by the parsed Java source code during the transformation process. Furthermore, we only consider the main development branch. We consider the *trunk* directory as main development branch in Subversion repositories.

Furthermore, we require modification deltas of artifacts (i.e. Jamopp models) to evaluate our incremental maintenance modes. For that, we exploited the version history of open source projects and pre-processed the Java source code to derive Jamopp models for each version control revision. When processing the modification delta of Jamopp models, an explicit adaptation of Jamopp models within the base graph storage is performed by employing difference and merge algorithms. We used EMFCompare to apply modification deltas to Jamopp models stored in the base graph storage. The technical realization is not in the scope of this technical report. For each selected open source project, we processed the first 100 revisions. Note that not all revisions include changes to the main development branch and, therefore, some revisions are skipped in our measurements. When all changes of one revision are applied to the base graphs, we immediately maintained the materialized view graphs using one of our maintenance modes.

Table 6.1 gives an overview of the open source projects used for evaluation. The table shows the number of artifacts for revision 1 and revision 100 per data set as well as the average modification size in terms of atomic change events[2].

Apache Ant *"Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other."*[3]

[1] `http://www.jamopp.org` (last access: 15th June 2015).
[2] An atomic change event is a change event that cannot be decomposed further.
[3] `http://ant.apache.org` (last access: 13th May 2015).

Table 6.1.: Overview of employed graph data sets

Projects	**Artifacts**		**Avg. Modification Size**	**Annotations**		**Index Size (MB)**	
Apache...	Rev.1	Rev. 100		Rev. 1	Rev. 100	Rev. 1	Rev. 100
Ant	12442	22071	492 (2,23% - 3,95%)	1725	2927	1,2	2,1
Commons IO	59423	65487	1567 (2,39% - 2,64%)	4229	4963	2,6	3,1
Xerces	133858	191246	1577 (0,85% - 1,21%)	20050	25636	13,3	16,7

Apache Commons IO *"Commons IO is a library of utilities to assist with developing IO functionality."*[4]

Apache Xerces *"Xerces2 Java is a library for parsing, validating and manipulating XML documents."*[5]

6.1.2. Software Design Pattern View Model

For our evaluation, we composed a view model with view modules that derive view graphs, which consists of information about employed software design patterns. Fig. 6.1 shows the complexity of the view model. The view model consists of 49 view modules. The concrete implementation of the view modules is not in scope of this technical report, but can be found in the appendix C. 15 of 49 view modules are high-level inference modules, i.e. modules that recover software design patterns. 16 of 49 inference modules are low-level inference modules, i.e. modules that do not depend on other inference modules. 18 of 49 inference modules are intermediate inference modules, i.e. modules that depend on other inference modules, but recover only software design sub-patterns required for recovering software design patterns.

6.2. Evaluation Results

Table 6.2 gives an overview about the total execution times required to process the changes of the first hundred revisions for each open source project in different maintenance modes. The speedup of the incremental modes in comparison to batch mode with preservation is given in brackets.

[4]`http://commons.apache.org/proper/commons-io/` (last access: 13th May 2015).
[5]`http://xerces.apache.org` (last access: 13th May 2015).

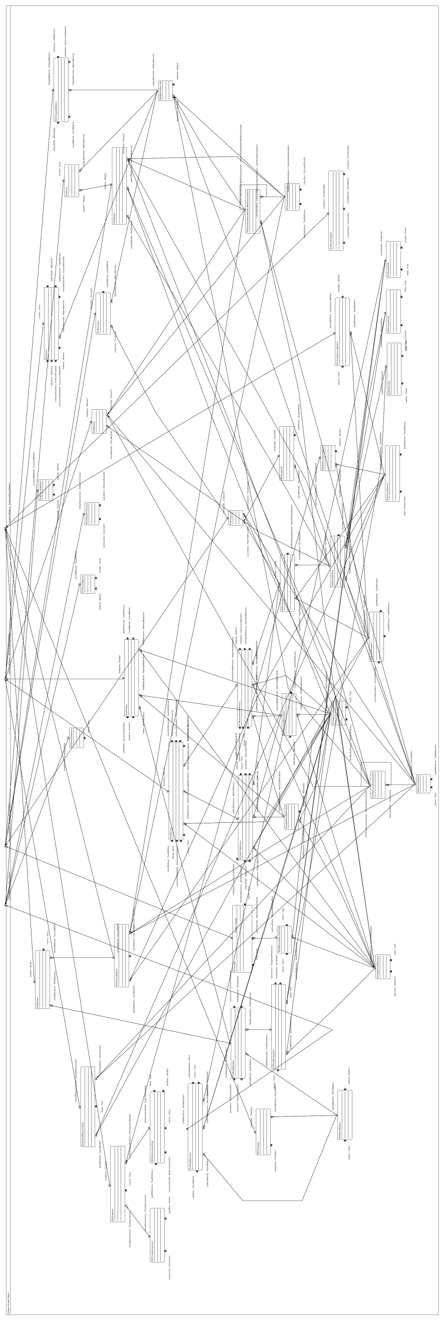


Figure 6.1.: Sketched view model used for evaluation

Table 6.2.: Overview of execution times for subsequent application of changes from revision 1 to 100 (speedup in brackets in comparison to batch mode with preservation)

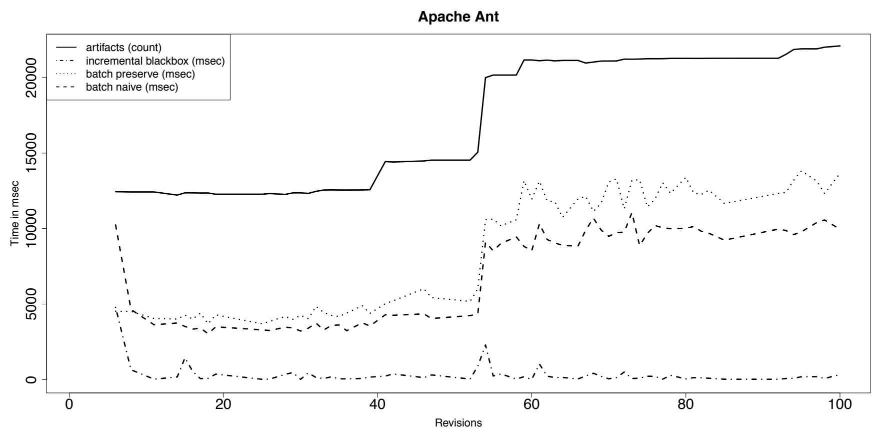
Open Source Project	Batch Naive	Batch Preserve	Incremental Black	Incremental White
Apache Ant	0 h 7 min 28 s	0 h 9 min 14 s	0 min 21 s (26,52)	0 min 22 s (24,92)
Apache Commons IO	0 h 23 min 19 s	0 h 28 min 48 s	1 min 23 s (20,95)	1 min 22 s (21,18)
Apache Xerces	14 h 27 min 3 s	15 h 33 min 44 s	17 min 42 s (52,74)	16 min 13 s (57,57)

In general, the evaluation results show that the batch maintenance mode with preservation is slower than the naive batch maintenance mode due to the fact that the preservation of annotations requires more effort as discussed in Sec. 5.6. For example, the naive batch mode required 7 minutes 28 seconds and the batch mode with preservation required 9 minutes and 14 seconds for the open source project Apache Ant.

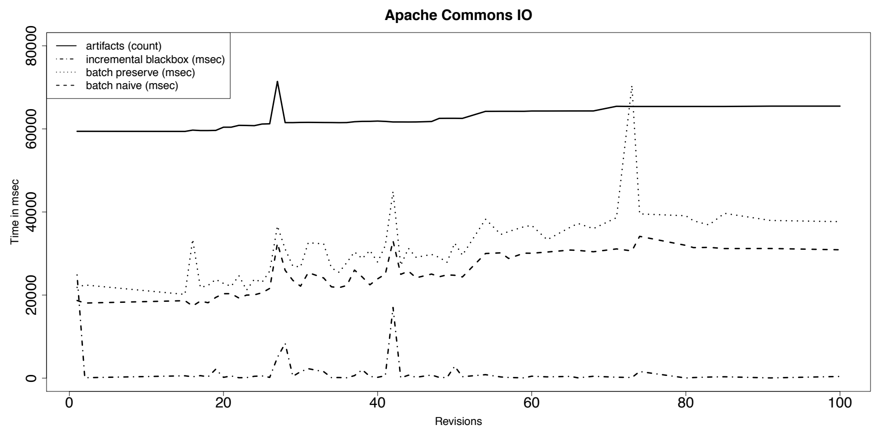
Furthermore, the evaluation results show that the incremental maintenance mode is significantly faster than the naive batch maintenance mode and batch mode with preservation due to the fact that the incremental maintenance mode only processes changed parts of base graphs as discussed in Sec. 5.6. For example, the incremental maintenance mode last 21 seconds when incremental black box mode is employed and 22 seconds when incremental white box mode is employed for the open source project Apache Ant. In comparison with the batch maintenance mode with preservation, the incremental maintenance modes have a speedup of approx. 25 for the open source project Apache Ant. For larger base graphs as for the open source project Apache Xerces, a speedup of approx. 55 for the incremental modes can be observed. We use the batch maintenance mode with preservation as reference measurement, because only the batch maintenance mode with preservation and the incremental maintenance modes preserve annotations.

Fig. 6.2 shows the required execution time per revision and maintenance mode graphically. The solid line denotes the number of artifacts in the base graphs. The dashed line denotes the execution time required by naive batch maintenance mode. The dotted line denotes the execution time required by batch maintenance mode with preservation. The dash-dotted line denotes the execution time required by the incremental black box maintenance mode. We neglect to plot the execution times for incremental white box maintenance mode, because the plots are very similar to the plots of the incremental black box maintenance mode.

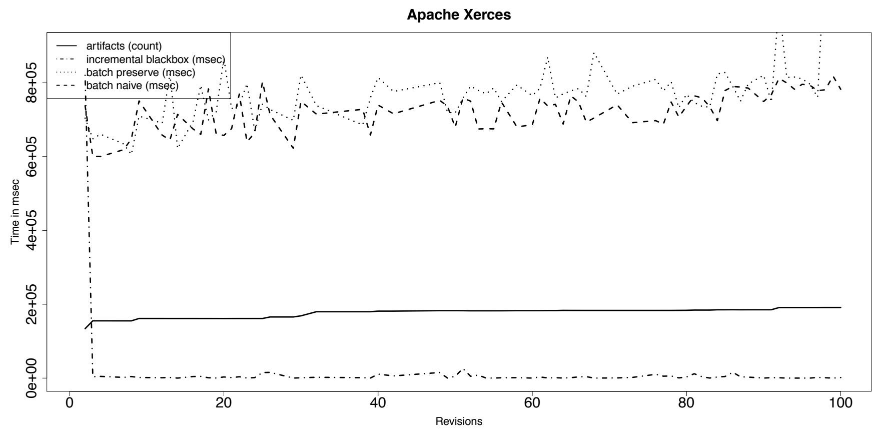
Table 6.3 shows the computed average scope size, i.e. the number of artifacts and annotations in the scope, and the average time required to compute the scope. Note that the depicted values consider the sum over all view modules. For example



(a) Apache Ant



(b) Apache Commons IO



(c) Apache Xerces

Figure 6.2.: Execution time in msec for revision 1 to 100

Table 6.3.: Overview of scope size and computation time (in brackets percentage share in comparison to batch mode with preservation)

Open Source Project	Avg. Scope Size (count)			Avg. Scope Computation Time		
	Preserve	**Inc Black**	**Inc White**	**Preserve**	**Inc Black**	**Inc White**
Apache Ant	85179	8823 (10,36%)	7344 (8,62%)	43 ms	30 ms	73 ms
Apache Commons IO	297950	32734 (10,99%)	24831 (8,33%)	190 ms	128 ms	216 ms
Apache Xerces	801026	44968 (5,61%)	39620 (4,95%)	1971 ms	157 ms	405 ms

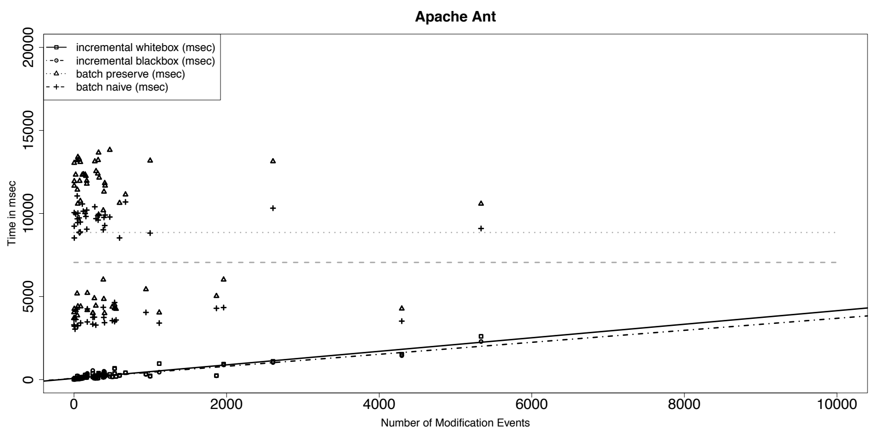
for the Apache Ant project, the number of artifacts and annotations that are part of the scope (cf. avg. scope size) is approx. only 10 % for incremental black box mode and approx. 9 % for incremental white box mode in comparison to batch mode with preservation. Furthermore, the computation of the narrowed scope in incremental white box mode takes in general approx. two times longer than the scope computation in incremental black box mode.

Fig. 6.3 depicts the time required for view graph maintenance related to the number of modifications. The plots show that the time required for *incremental* view maintenance correlates with the number of modification events in a linear manner. The time required for *incremental* view maintenance increases when the number of modification events increases. Note that the gray dashed and dotted lines depict the arithmetic mean of the time required for *batch* view maintenance in terms of a reference line, because no correlation between the number of modification events and the time required for view maintenance exists.

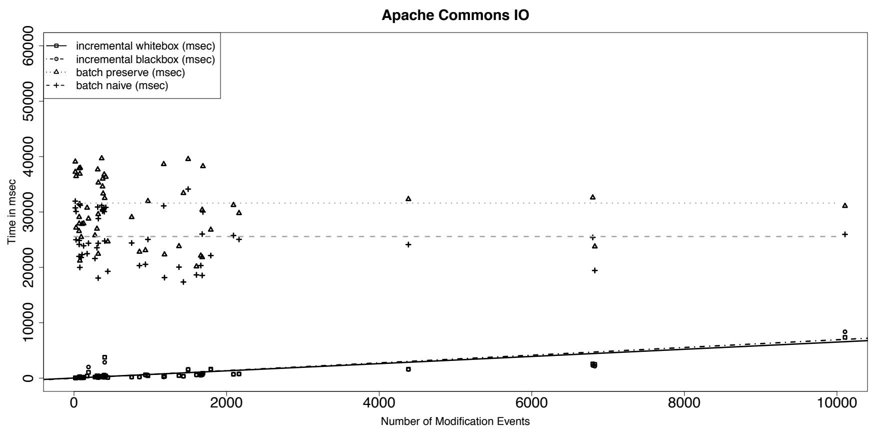
Fig. 6.4 depicts the memory footprint required for storing annotations (i.e., marking matches in base graphs and view graphs). For measuring the memory footprint we used the open source tool Java Agent for Memory Measurements[6] (JAMM). JAMM uses Java's instrumentation API to compute an approximation of object sizes. We counted the object size of annotations, roles owned by annotations as well as artifacts that reference objects in base graphs. We did *not* count the objects in base graphs that are referenced by annotations, because these objects belong to the base graph and are not part of view graphs.

Table 6.4 shows the index sizes for revision 1 and 100. For the open source project Apache Ant, approx. 3000 annotations exists for revision 100. These annotations lead to an index size of approx. 2,1 mega bytes (MB) required for storing annotations, their roles, and artifacts that reference the object in base graphs.

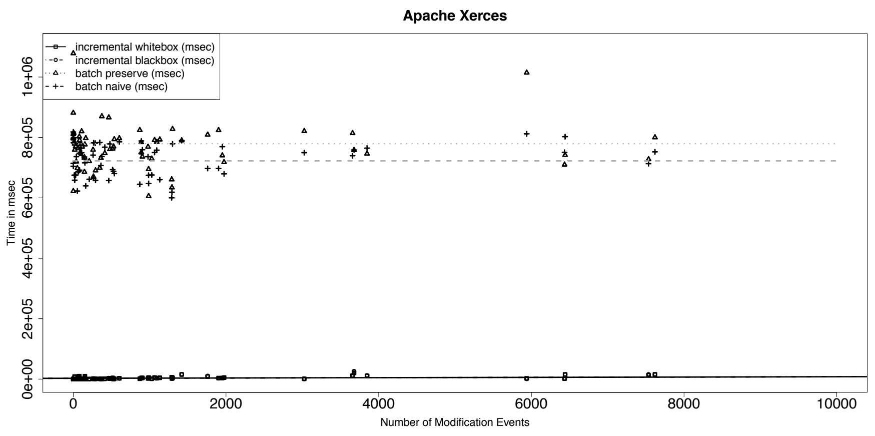
[6]`https://github.com/jbellis/jamm` (last access: 10th July 2015).



(a) Apache Ant

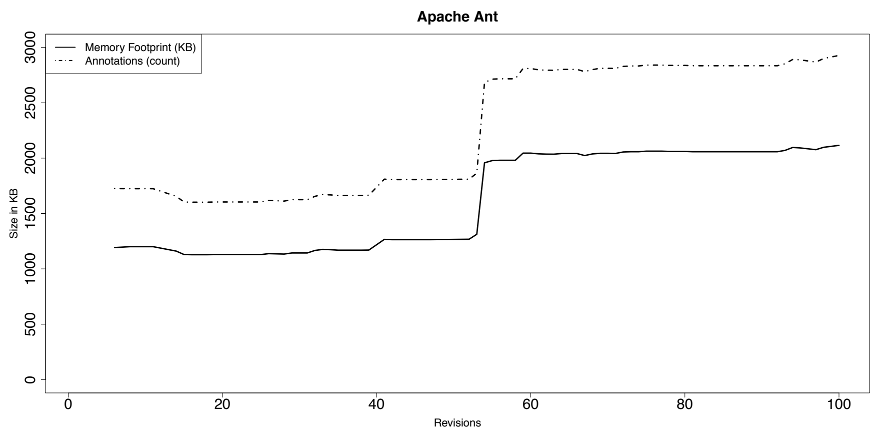


(b) Apache Commons IO

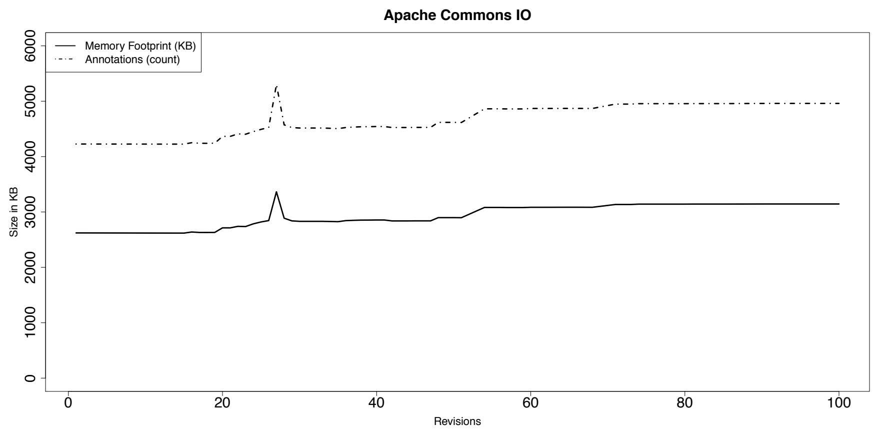


(c) Apache Xerces

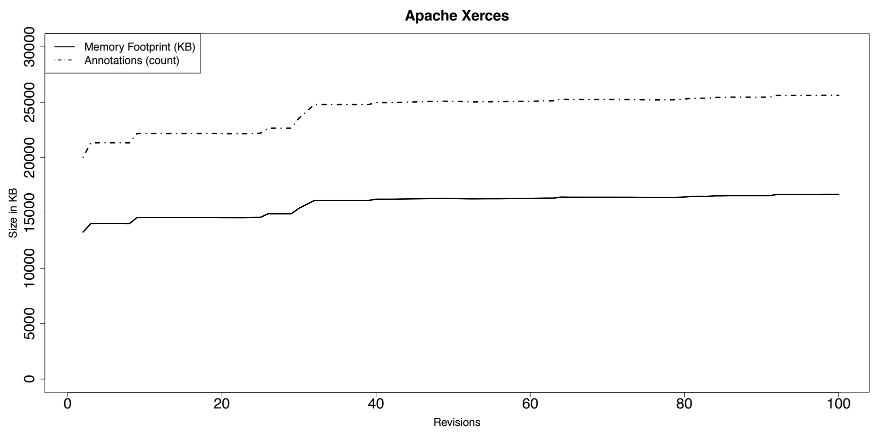
Figure 6.3.: Correlation between number of modifications and execution time



(a) Apache Ant



(b) Apache Commons IO



(c) Apache Xerces

Figure 6.4.: Number of annotations and size of memory footprint in kilo bytes

Table 6.4.: Overview of index sizes per data set

Open Source Project	Annotations		Index Size (MB)	
	Rev. 1	Rev. 100	Rev. 1	Rev. 100
Apache Ant	1725	2927	1,2	2,1
Apache Commons IO	4229	4963	2,6	3,1
Apache Xerces	20050	25636	13,3	16,7

6.3. Evaluation Discussion

The actual speedup of the incremental maintenance mode in comparison with the batch maintenance mode with preservation depends on the actual base graph modifications as well as employed view model. The performance of the incremental maintenance mode bases on the heuristic that the number of modified artifacts in base graphs is low between two consecutive revisions. Table 6.2 shows the average modification size of base graphs for used open source projects. Table 6.2 shows that the heuristic works especially well for large-scale base graphs such as Apache Xerces, because in general the average modification size decreases when the size of base graphs increases. This fact is underlined by the observed speedup increase with growing base graph sizes.

Furthermore, the concrete artifacts included by the modification delta a) have an impact which concrete modules have to be re-executed and b) have an impact on the size of the scope that is passed to the modules that must be re-executed. Note that view modules have different complexities and, therefore, the concrete modules that have to be re-executed per UDC cycle impact the performance of the maintenance.

The evaluation results show that the number of artifacts and annotations computed by the incremental white box mode is smaller than for incremental black box mode. Furthermore, the evaluation results show that the time required by incremental white box mode to compute the scope is greater than for incremental black box mode. Due to this observation, the additional time required by incremental white box mode may not pay off in comparison to incremental black box mode, because it is more expensive to narrow the scope before passing the scope to view modules than passing an approximated scope that is less expensive to compute as in incremental black box mode.

6.4. Threats to Validity

We performed each execution time measurement only once, because the differences of the execution times are large and significant enough that additional measurements will not change the overall observation. Moreover, additional measurements would increase the overall time required to perform the evaluation. This is practically infeasible, especially for large-scale base graphs. However, unexpected Java garbage collections may impact the overall execution times negatively. To avoid that the garbage collection is triggered unnecessarily, we increased the Java heap space to 256 GB of main memory.

We pre-processed the Java source code to derive abstract syntax graphs in terms of Jamopp models, because parsing large-scale Java source code on demand is an expensive operation that may exceed the overall time required for the actual maintenance of view graphs. However, due to our pre-processing some parts of the abstract syntax graphs are skipped, e.g. cross references to third-party libraries.

Therefore, fewer annotations and, therefore, smaller view graphs are created and maintained. Thus, less maintenance effort is required, because less annotations are contained by view graphs. But, note that we do not aim at improving the precision and recall of annotations that describe employed software design patterns.

Furthermore, all view graphs created and maintained by our different maintenance modes contain equal annotations. The implementation of our maintenance modes leads to equal view graphs.

7. Related Work

In this chapter, we describe related work and finally discuss this related work by comparing it with our approach. We identified several research areas that are related to our approach. We describe the related work for each research area separately. First, we describe existing approaches for discrimination networks (Sec. 7.1). Afterwards, we describe how discrimination networks among other things are used for view maintenance (Sec. 7.2) of relational databases (Sec. 7.2.1), object-oriented databases (Sec. 7.2.2), and graph databases (Sec. 7.2.3). Furthermore, we describe which graph indexing approaches (Sec. 7.3) and graph querying approaches (Sec. 7.4) exist that are independent from graph databases and are related to our approach. Moreover, we describe incremental processing approaches from the research area of Model-Driven Engineering (Sec. 7.5).

7.1. Discrimination Networks

In recent related work, discrimination networks are used for condition testing. For example, this condition testing is employed in production systems [21], active database management systems [35], and view maintenance [37]. Different kinds of discrimination networks exist that are presented in detail in the following sections. We introduce the different kinds of discrimination networks in historical order. We describe Rete networks (Sec. 7.1.1), TREAT networks (Sec. 7.1.2), and Gator networks (Sec. 7.1.3).

7.1.1. Rete Network

Forgy [21] introduced the *Rete Match Algorithm* (Rete algorithm for short) to find all objects that match each pattern from a large collection of patterns. Originally, the algorithm was developed for executing production rules with the help of production systems interpreters. A production rule consists of a left and right hand side. The left hand side specifies conditions that must be fulfilled by working memory elements to perform the sequence of actions on the right hand side of the production rule. Since the set of working memory elements changes over time, e.g. due to the actions performed by a production rule or working elements that are added, removed, or modified by users of the production system, the conditions on the left hand side of production rules must be tested efficiently to enable a high execution speed for production systems. Efficient testing means that the condition testing is only performed when the working memory changes can impact the satisfaction of the left hand side of production rules. The Rete algorithm addresses this performance issue by introducing *Rete networks* that avoid iterating over working memory between cycles of working memory changes.

In general, the Rete algorithm executes an indexing function that *"is represented as a network of simple feature recognizers"* [21], i.e. single condition tests. The results of condition tests can be combined to more complex conditions tested by subsequent condition tests. Furthermore, each condition test stores which working memory elements pass the condition test. Note that a single condition test can combine at most two results of antecedent condition tests. When working memory elements are added to (resp. removed from) working memory the Rete algorithm finds all conditions satisfied (resp. dissatisfied) by the added (resp. removed) working memory elements and updates the list of working memory elements associated with each condition in the Rete network by adding (resp. removing) the working element to (resp. from) the list of working memory elements that pass the condition test.

Rete networks consist of four kinds of nodes: root nodes, terminal nodes, one-input nodes, and two-input nodes. The root node distributes tokens. Tokens are descriptions of working memory element changes and consist of a tag and a list of working memory elements. The tag describes whether a working memory element has been added (positive token) to or removed (negative token) from the working memory. When a working memory element has been modified, two tokens are distributed via the root node to the Rete network, which denote that a working memory element has been removed and subsequently added to the working memory. One-input nodes follow the root node and perform a single condition test over working memory elements as defined on the left hand side of production rules. If the working memory element passes the condition test, a positive token is passed to subsequent nodes in the Rete network. If the working memory element does not pass the condition, a negative token is passed to the subsequent nodes. Two-input nodes follow one-input/two-inputs nodes and join the incoming working memory elements that passed the two antecedent condition tests. For that purpose, two-input nodes store the working memory elements that are passed from the antecedent left and right Rete (sub-)network in their internal memory. When a positive token arrives at a two-input node, the token is added to the left or right internal memory. When a negative token arrives at a two-input node, the token is removed from the left or right internal memory. When a left (resp. right) incoming token is positive, the two-input node checks whether the token can be joined with tokens stored in the internal right (resp. left) memory and, if yes, pass a positive token to subsequent nodes. When a left (resp. right) incoming token is negative, the joins do not exist anymore and a negative token is passed to subsequent nodes. Terminal nodes follow one-input/two-inputs nodes and store all working memory elements that satisfy the left hand side of production rules. The working memory element associated with a positive incoming token is added as instantiation to the so called conflict set. The working memory element associated with a negative incoming token is removed from the conflict set. Afterwards, the production system interpreter selects a production rule whose left hand side is satisfied by an instantiation from the conflict set and executes the sequence of actions from the right hand side of the production rule. These actions can lead to working memory changes that lead to new tokens processed by the Rete network as described above. The original network interpreter is implemented in a Pascal-like language. For implementation details, we refer to [21].

Schor et al. [70] present several improvements to the Rete Matching Algorithm. The most crucial improvement by Schor et al. concerning efficiency is an efficient support of modified working memory elements. Instead of mapping modifications of working elements to subsequent positive and negative tokens that *"causes an excessive retriggering of rules"* [70], the improvements by Schor et al. [70] avoid such re-triggerings if existing instantiations continue to exist and only performs such re-triggerings in terms of subsequent positive and negative tokens if the previous test result of a modified working memory element differs from the current test result.

Bunke et al. [9] transferred the concepts of the Rete Matching Algorithm to the efficient implementation of graph grammars. The Rete network is derived from the left hand sides of graph grammar rules, i.e. productions, and consists of five different types of network nodes. These network nodes are distinguished into root nodes, node checkers, edge checkers, subgraph checkers and production nodes. The root node is connected to node checkers and edge checkers and distributes incoming nodes and egdes of the underlying graph (i.e. graph nodes and graph edges) to these checkers. Graph nodes are sent to node-checkers, while graph edges are sent to edge-checkers. Node checkers and edge checkers conform one-input nodes of original Rete networks. Node checkers check whether the passed graph node has a certain label. A graph node that successfully passes the node checker is transmitted to a subsequent production node. Production nodes conform terminal nodes in original Rete networks. Edge checkers check whether the source and target graph nodes of the passed graph edge have certain labels and that the graph edge itself has a certain label. A graph edge that successfully passes the edge checker is transmitted to (multiple) subsequent subgraph checkers and possibly a production node if the checked subgraph corresponds to a left hand side of graph grammar rule. A subgraph checker has two incoming edges and conforms two-input nodes in original Rete networks. *"Any graph [...] that results from the combination of [left and right incoming subgraphs] is sent to all direct successors of the [...] subgraph checker"* [9].

In general, Rete networks are either left-associative or right-associative concerning their two-input nodes, but not both at a time. *"In an ordinary Rete [network] [...] a join node never has another join node as its right-hand side [(resp. left-hand side)] input"* [54]. However, when generalizing Rete networks by removing the restrictions on the associativities of join nodes, i.e. mixing left-associative and right-associative join nodes in terms of reconvergent join nodes, the original Rete algorithm produces duplicated or missing matches. Lee et al. [54] provide counter examples. The reason for this flaw is that the original Rete algorithm employs a depth-first processing in all cases. But, Lee et al. [54] employ a topological sorting of nodes in the Rete network and, furthermore, distinguish between right and left distribution of tokens to ensure that the results of all intermediate join nodes are available when required. Furthermore, Lee et al. [54] introduce stop and resume nodes as extension of ordinary Rete networks to delay match processing to a later point in time (possibly on demand).

7.1.2. TREAT Network

TREAT is an alternative discrimination network approach for Rete networks. TREAT was originally introduced by Miranker et al. [59, 61] to overcome the primary disadvantages of Rete networks. These disadvantages are high memory footprint due to two-input nodes (i.e., join nodes) that store intermediate states, similar sequences of required network maintenance steps when a working memory element is removed that are as expensive as when adding a working memory element, and shared network parts make parallel computation difficult.

In comparison to Rete networks, TREAT networks only employ one-input nodes that perform condition testing for working memory elements. The working memory elements that pass the condition test are stored in subsequent alpha-memories. A TREAT network does not consist of two-input nodes and beta-memories that store join results of two-input nodes. Instead, TREAT networks compute join results on demand based on change information about working memory elements called constrained search for instantiations. For that purpose, an alpha-memory is partitioned into old memory (already processed working memory elements), new-delete memory (new deleted working memory elements), and new-add memory (new added working memory elements). The search for instantiations takes place between the old memories and new memories. Since TREAT networks do not consist of two-input nodes no join results are stored in TREAT networks. Therefore, TREAT networks do not have to recompute join results when working memory elements are deleted in contrast to Rete networks and, thus, outperforms Rete networks during deletions. But, TREAT networks require additional effort when working memory elements are added to compute join results in contrast to Rete. The evaluation presented by Miranker [59] shows that the extra number of comparisons required by TREAT networks for added working memory elements does not exceed the number of comparisons required by Rete networks for deleted working memory elements in some cases.

Miranker et al. [60] extend the TREAT approach by a lazy matching procedure for production systems (e.g., RETE and TREAT). Their approach employs a best-first (meaning recency-first) search for instantiations by suspending searches for instantiations and proceeding with the search for instantiations that may exist due to added, removed or modified working memory elements as result of previous rule firing due to found instantiations. A stack is employed to keep track of previous instantiations for resuming the search for instantiations.

Hanson [34] enhanced TREAT called Ariel TREAT (A-TREAT for short) in terms of speed up and storage reduction. Note that the following improvements can be transferred to Rete networks as well. A-TREAT employs an *"interval binary search tree to efficiently test conditions that specify closed intervals [...], open intervals [...], or points [...]"* [34] to speed up rule processing. Ordinary alpha-memories store working memory elements that passed the antecedent selection condition test performed by one-input nodes. This leads to duplicated data that is already stored in the underlying data corpus. The situation becomes even more problematic when the selectivity of the condition test is low, because a lot of working memory elements pass the condition test and will be duplicated. For this purpose, Hanson proposes the concept of virtual alpha-memory, *"which contains a predicate describing the contents of the node rather than the qualifying data itself"* [34]. Virtual alpha-memories consist of a predicate and an identifier of the relation on which the predicate is defined. Virtual alpha-memories enable to reduce the required storage, but require additional effort when the selection condition test has to be performed, e.g. when joining working memory elements.

7.1.3. Gator Network

Hanson et al. proposed the discrimination network called Gator network for rule condition testing and view maintenance of relational databases in a series of technical reports [39, 38, 36] that are summarized in [37]. Gator networks are generalized

Rete and TREAT networks. In contrast to Rete and TREAT networks, Gator networks enable an arbitrary network structure. Especially, Gator networks allow join nodes (resp. two-input nodes in Rete) with more than two inputs. Therefore, Rete and TREAT networks are considered as the extremes of Gator network structures. Input nodes with more than two inputs employ an internal join order plan to join incoming tuples in a fixed manner, because the join order plan is determined when the network is initially created.

Hanson et al. [37] state that the performance of Gator networks relies on the network structure. For that purpose, the authors present a cost model for Gator networks that is used to predict the performance of a certain network structure. Their *"cost functions estimate the expense to propagate tokens through a Gator network, assuming a frequency of token arrivals at different nodes determined by the frequency statistics, relation cardinality, attribute cardinality, selection and join predicate selectivity"* [37]. With this cost prediction generated Gator networks are optimized by employing a combination of state-space search techniques. The authors show in their evaluation that *optimized* Gator networks can outperform Rete and TREAT networks.

7.2. Database View Maintenance

In the literature, different approaches exist for view maintenance of databases. In general, we distinguish between view maintenance for relational databases (Sec. 7.2.1), object-oriented databases (7.2.2), and graph databases (Sec. 7.2.3). In the following sections, we focus on the maintenance of materialized views, i.e. views that are stored in the database. Furthermore, we only consider changes to base tables/graphs that must be propagated to view tables/graphs to keep derived views consistent with their base tables/graphs. We do not consider changes to view tables/graphs that are propagated back to base views/graphs.

7.2.1. View Maintenance for Relational Databases

Shmueli et al. [73] present an approach for immediate view maintenance of relational databases. The approach of Shmueli et al. [73] bases on a good-bad marking scheme. This good-bad marking scheme enables to determine whether an inserted or deleted tuple has an impact to views. Views are considered as joins of two relations (and relations resulting from joins) in the database. Good tuples are tuples that have an impact on the view (i.e., join result), while bad tuples have no impact on views. The approach works for acylic databases, but can be transferred to cyclic databases by transforming the cyclic schema into a tree [28]. Tuples in leaf relations are always good. When tuples are inserted to or deleted from intermediate nodes it must be checked whether they are good or bad. A tuple is good if the tuple can be joined with at least one tuple in every child relation. Otherwise, the tuple is bad. Good tuples must be checked for *"compatibility above"* [28], i.e., whether currently bad tuples on the path to the root node become good. Shmueli also describes an optimization of this approach that employs good-counters and up-pointers that enable a more efficient look up of relevant tuples during maintenance [73].

Blakeley et al. [7] present an approach for efficient and immediate updates of materialized views of relational databases to keep materialized views consistent with their base tables. The basic idea of their approach is to determine whether a certain change to a base table is relevant or irrelevant for derived view tables, i.e. whether this change has an impact on the consistency between derived view tables and their base tables. For that purpose, their approach determines whether selection conditions become satisfiable or unsatisfiable concerning the current database state and employ a differential update that determines which tuples must be inserted and deleted to keep view tables consistent with base tables. The authors discuss differential update algorithms in relational algebra for select views, project views, and join views. Finally, they combine these algorithms in a common select-project-join view algorithm in relational algebra.

Ceri et al. [11] present an incremental view maintenance approach for materialized views of relational databases. Their main concept is to derive maintenance rules from view definitions for materialized views in terms of production rules that are applied when certain changes to base tables occur. These production rules propagate changes from base tables to view tables. Their approach consists of two basic steps: view analysis and rule generation. During view analysis the view definitions are analyzed to determine whether an efficient incremental maintenance can be employed. If yes, for each view definition maintenance rules are generated for insertion and deletion of tuples to base tables. Updates of tuples are mapped to insert and delete maintenance rules. Since view definitions can make use of several operators, the authors discuss how these operators require different maintenance rules. We refer to [11] for a detailed discussion. For example, comparison operators are supported as well set operators such as union and intersection. Furthermore, nested positive and negative subqueries are support, but only for a single nesting level. The authors do not give a performance evaluation.

Qian et al. [66] present an approach for deriving incremental relational expressions from ordinary relational expressions that are ordinarily used for view definitions. With the help of these incremental relational expressions changes to base tables can be propagated incrementally to derived view tables. The authors present an algorithm that derives incremental relational expressions from ordinary relational expressions with the help of equivalence-preserving transformation rules.

The algorithm applies these transformation *"to factor out the incremental changes in relations from the relational expressions"* [66] taking as input the ordinary relational expressions and incremental changes to base tables. The algorithm result is an incremental relational expression that is used together with incremental changes to base relations to compute the tuples that must be added to or removed from derived view tables to keep them conform with their base tables. Griffin et al. [29] extend the algorithm of Qian et al. [66] to guarantee minimality, i.e. no unnecessary tuples are generated in the change sets.

Harrison et al. [40] present an approach for incremental view maintenance of relational deductive databases that supports negation and recursion. The authors contribute the Propagation Filtration algorithm (PF algorithm for short). The algorithm consists of two phases: propagation phase and filtration phase. During the propagation phase, candidate rules are determined and executed in a constrained manner to determine an approximation of tuples that may impact derived view tables. Candidate rules (i.e., view definitions) are rules that depend directly or indirectly on base relations. When executing the candidate rule, the rule is *"constrained using the bindings contained within the set of updates for the relation"* [40]. For that purpose, the candidate rule body (i.e., the actual query) must be known. The execution result of all candidate rules is considered as approximation of potentially affected tuples in view tables. During the filtration phase, for each tuple in the set of approximated tuples is checked whether the tuple is a relevant or irrelevant update to the view. For this purpose, for added tuples is checked whether they are provable on the old database state (incl. views) and for removed tuples is checked whether they are provable on the new database state. Tuples in the approximation that are not provable are actual tuples that must be propagated to views. Note that the PF algorithm supports recursion and negation.

Gupta et al. [33] present two approaches for incremental view maintenance of relational databases. Their approach aims at supporting duplicates efficiently. Duplicates are considered as tuples that can be derived multiple times from base tables. Their "counting" algorithm maintains the number of alternative derivations for each derived tuple. If the number of alternative derivations drops to zero, the derived tuple is deleted, since no alternative derivation exists. Otherwise, the derived tuple is preserved. They also present the DRed algorithm that overestimates the set of derived tuples that must be deleted due to changes of base tables. Afterwards, the DRed algorithm searches for alternative derivations and creates new tuples in view tables, if these alternative derivations exist. Note that for both algorithms delta rules are derived from view definitions to apply these procedures.

Gupta et al. [31] present also another approach about using only partial information to update materialized views. They motivate their research with the fact that in certain scenarios materialized views and base relations reside on different computing nodes. But, it should be still possible to update materialized views when, e.g. base relations are not available and instead only the view definition, the current state of the materialized view and the update delta is available. The authors consider different scenarios to demonstrate their approach. The basic contribution of their approach is how to infer whether a certain update delta can contribute to the current state of the materialized view based on boolean expressions that are derived from information available in certain scenarios. Furthermore, with their approach it can be inferred whether a certain tuple must be added or removed from the materialized view without accessing base relations. Their approach works for select-project-join (SPJ for short) views.

Ross et al. [69] present an approach for incremental view maintenance and integrity constraint checking for relational databases. They aim at cost reduction for maintaining materialized views incrementally. The basic idea of their approach is to maintain additional views that support the incremental maintenance of a specific view (resp. constraint) defined by database users. The authors argue that although maintaining such additional views the effort for incremental maintenance of a specific view (resp. constraint) can be reduced. For this purpose, the authors introduce the notion of update tracks. An update track is a set of nodes and edges between these nodes that are affected by transactions of a certain type. Multiple equivalent update tracks can exist that lead to the same evaluation result. The authors contribute an algorithm that finds the cheapest update track in a directed acyclic graph (DAG) of expression nodes that are derived from expression trees (i.e. view definitions) with help of equivalence rules. Expression nodes have expressions as children that evaluate to the same result. However, the presented algorithm is expensive. Therefore, the authors outline three heuristics to find an optimal update track. They propose to a) use a single expression tree instead of an expression DAG, b) materialize results of expression nodes that would be expensive to compute incrementally if not materialized, and c) employ a Greedy algorithm to find an optimal update track.

Colby et al. [13] present an approach for deferred incremental view maintenance for relational databases. The authors aim at minimizing view downtime (due to maintenance) while obtaining the per-transaction overhead also low. The authors reach this goal by decoupling base logs that record changes to base tables from differential tables that store changes that must be applied to derived view tables to keep base tables and view tables consistent. Changes stored in base logs are propagated to differential tables every k time units, while view updates are propagated from differential tables to view tables every $m > k$ time units. Since only the base logs must be updated by each transaction the per-transaction overhead is minimized, while the view maintenance downtime is also minimized, because required incremental changes to view tables are derived periodically from base logs and stored in differential tables and can be directly applied with low overhead when consistent view tables are required. However, a performance evaluation is not given by the authors.

Colby et al. [14] extend their former approach [13] by enabling the support of multiple view maintenance policies at once. Their research is motivated by the fact that previous research focused on single view maintenance approaches in isolation. However, combining multiple view maintenance policies in a graph of dependent materialized views can be beneficial in practice. But, supporting multiple view maintenance policies at the same time yields consistency issues that have to be considered when maintaining view tables to keep them consistent with their base tables. For this purpose, the authors introduce the notion of view groups and view dependency graphs. Furthermore, they specify properties (e.g., *"A query on a view in a viewgroup can be answered without querying any other view group"* [14]) and rules (e.g., *"An immediate view cannot have a deferred view or a snapshot view as a parent."* [14]) for view groups to define which combinations of view maintenance policies are meaningful and valid. The authors consider a) update to base tables, b) queries over deferred views and c) explicit/periodic refresh of a viewgroup containing snapshot views. Note that the addition and deletion of views is not considered. The authors provide update operation for these three cases. In case a), the update operation updates all immediate views and auxiliary tables required by dependent views. In case b), deferred views are recursively refreshed, if required at all. In case c), all snapshot views in a viewgroup are updated and all dependent views are updated in a breadth-first manner.

Mistry et al. [62] present an approach for optimizing view maintenance of relational databases when multiple views are involved. The authors exploit query expressions common to different view definitions to reduce maintenance costs. Their approach consists of a query expression DAG as used in query execution optimization that is enhanced with additional equivalence nodes that correspond to differentials, i.e. changes made to relations. This enhanced query expression DAG is used to compute the maintenance costs of possible execution paths in the DAG. The maintenance cost is used by a greedy algorithm for computing how to integrate additional materialized views with best plans for view maintenance by selecting views with best benefit.

The research about active relational databases and production rule triggering is related to view maintenance, because the tuples associated with each node in a Rete network [21], TREAT network [61], and Gator network [37] can be considered as intermediate views. Especially, the tuples associated with production nodes can be considered as equivalent to view tables. Alternatively, the actions (resp. right-hand side) of production rules can be used to trigger modification of base tables when certain events in base tables occur as, e.g., described by Ceri et al. [11]. For example, Hanson et al. [37] states that discrimination networks can be used for incremental view maintenance of relational databases. In case of Hanson et al. [37] Gator networks are employed, but also Rete networks [21] and TREAT networks [61] can be employed instead, because they are extremes of Gator networks (see Sec. 7.1). Note that discrimination networks have been only used in the context of relational databases for incremental view maintenance.

7.2.2. View Maintenance for Object Databases

Gluche et al. [25] present an approach for incremental maintenance of views in object-oriented databases using Object Query Language (OQL). They propose a *"framework in which view definitions and updates are understood as mappings between algebraic domains that represent collection type constructors such as sets, bags, and lists"* [25]. The algebraic properties of these mappings are used to generate incremental update plans. The authors show that OQL is appropriate as view definition language for object-oriented databases, because a large subset of OQL may be translated to such mappings.

Kuno et al. [53] present an incremental maintenance approach for materialized object views called MultiView. In contrast to view maintenance in relational databases, the approach presented by Kuno et al. [53] make use of object-oriented concepts that enable to incrementally maintain views called virtual classes derived from base classes. Virtual classes are defined with the help of object algebra operators. In general, the maintenance procedures consider two basic dependencies called membership-dependency and value-dependency. For computing maintenance steps concerning membership-dependencies the generalization hierarchy of base and virtual classes is taken into account. For computing maintenance steps concerning value-dependencies a register service is provided that enables virtual classes to register for change notifications of certain class properties as used in view definitions. Note that additional virtual classes can extend virtual classes (i.e., views) and updates to base and virtual classes are propagated through the chain of virtual classes.

Liu et al. [55] present a view maintenance approach for object-relational databases using Object Relational SQL (QR-SQL), i.e. objects are mapped to tables. In contrast to relational databases, object inheritance and object references must be considered when maintaining views in object-relational databases, because changes to tables that are not explicitly specified in view definitions can occur. The employed maintenance procedure applies a step-wise query rewriting by, e.g., making referenced tables explicit in view definitions and considering inherited and inheriting tables in terms of triggers for view maintenance.

Akhtar et al. [1] present an approach for the incremental maintenance of materialiazed views in object-oriented databases for view definitions specified in Object Query Language (OQL). The basic idea of their approach is to analyze the view definitions at an algebraic level to derive information which kinds of modification events can render the materialized view inconsistent to the base relations from which the view has been derived. These kinds of events are used to derive incremental maintenance plans that are applied to keep derived materialized views up-to-date. Which actions are performed by an incremental maintenance plan is discussed by the authors in detail and depends on the concrete view definition and modification events. To keep track which objects in base relations contribute to which derived views, auxiliary views are derived and maintained that contain the object identifiers of objects that contribute to derived views.

7.2.3. View Maintenance for Graph Databases

Kiesel et al. [50, 51] introduce *GRAS* as a graph-oriented software engineering database system. *GRAS* is a graph storage that uses the graph rewriting language *PROGRES* [71] as query language. The data model of *GRAS* consists of an explicit type layer for nodes and edges. Graphs in *GRAS* are attributed graphs. The authors state that a graph database system *"should support the incremental computation of derived data"* [51]. Unfortunately, no explicit definition of the term *derived data* is given and we assume that derived data are computed attribute values, because GRAS consists of an explicit index storage for supporting the indexing of attribute values. However, we found no hint that GRAS support views over graph data and, therefore, does not provide concepts for maintaining view graphs.

Zhuge et al. [82] introduce the notion of *graph-structured databases* along with the notion of virtual and materialized views for graph structured databases. Their graph data model consists of atomic types and set types. Atomic types are leaf of the graph (e.g. integer or string values), while set types are intermediate nodes that consist of children. Materialized views employ delegates that reference the object in base graphs and other materialized views, respectively. Note that edges are not contained by materialized views and are retrieved from base graphs. View definition queries specify views in terms of selection paths and selection conditions. When updates to base graphs occur, the materialized view is immediately maintained. The maintenance procedure determines for the inserted, deleted, or modified nodes in the base graph whether it must be added to or removed from the derived views by re-evaluating the selection path and condition for the given nodes. Note that materialized views are considered itself as graph structured databases and, thus, can be handled the same way. However, the authors do not comment on this issue concerning change propagation from base graphs to other views graphs with multiple view graphs in between. Furthermore, the authors limit their algorithm to tree structured databases and selection paths and conditions, instead of graph structured databases and selection path expressions with wild cards. Moreover, their approach requires the view definition specification. Arbitrary graph patterns for defining views are not considered.

Balsters [3] proposes to use Object Constraint Language (OCL) as view definition language for UML classes to define derived classes. However, Balsters argues that OCL is not relational complete as relational algebra due to the missing natural join operator in OCL. Thus, he proposes a natural join operation for OCL. The maintenance of derived classes is not considered by the author.

Angles [2] presents a comparison of graph database models by focusing on current graph database implementations used in practice such as Neo4J[1], AllegroGraph[2], InfiniteGraph[3] and others (see [2]). In summary, most graph databases are based on simple and attributed graphs and rather provide an application programming interface (API) for current programming languages for interaction with the graph database than providing a standard query language. From the perspective of view maintenance, current graph databases support only indexes. These indexes are rather simple indexes *"over a property for all nodes that have a given label"* [75] as in Neo4j than complex graph queries specified by view definitions as known from relational databases. Maintenance of materialized views based on view definitions are not considered by Angles' comparison [2]. We examined the manuals of the graph database implementations mentioned above and conclude that materialized views based on view definitions are not supported by current graph database implementations.

Srinivasa et al. [74] propose the graph database system GRACE that enables search for similar graphs and substructures. GRACE employs three kinds of indexes for querying: an attribute value index, a graph location index, and a label walk index. The label walk index is a path-based indexing approach that enables to search for similar graphs and substructures. However, the authors do not comment on how the index is maintained in case the graphs in the graph database are modified. GRACE employs a new invented language called Safari for graph retrieval.

Jouili et al. [47] present a graph database benchmark that they use afterwards for a performance comparison of current graph databases. However, it remains also an

[1]`http://neo4j.com` (last access: April 28th 2015).
[2]`http://franz.com/agraph/allegrograph/` (last access: April 28th 2015).
[3]`http://www.infinitegraph.com` (last access: April 28th 2015).

open question which impact indexes have on the performance and whether there are differences between index performances of different graph databases. Graph views are not considered.

Khurana et al. [49] present a system for snapshot retrieval of historical graph data. Snapshots are one kind of database views that, in contrast to materialized views and virtual views, are copies of a certain database state. Since naive copies are very inefficient concerning storage consumption, because of redundant stored partitions, the authors present how to use delta information between snapshots derived from modification events to reduce storage consumption. Moreover, their system enables to define auxiliary indexes that are captured along with snapshots. These auxiliary indexes store user-defined index information. It seems to be the responsibility of the user to specify the actual indexing function, differential function, and functions that exploit delta information during retrieval, because the paper lacks a detailed description of this issue. The authors present this idea using the example of subgraph pattern matching over a set of historical graph data snapshots.

7.3. Graph Indexing

Goldman et al. [26] propose *DataGuides*. A DataGuide enables to derive a dynamic schema for semi-structured and graph-structured databases. Furthermore, this dynamic schema is maintained when edges are added to or removed from the database. DataGuides are graphs that summarize the structure of semi-structured databases. A node in a DataGuide represents nodes that are reachable via the same path starting at the root node in the database. Edges between nodes in a DataGuide represent edges between nodes with the same label in a database. Thus, DataGuides can be considered as path index and, therefore, besides serving as dynamic structure to facilitate the formulation of queries over semi-structured data, DataGuides are also used to accelerate query evaluation. Note, that path queries with wildcards are not supported. Also note that DataGuides are no views, because path query results are not materialized as it is the case for database views.

Messmer et al. [57] present a graph indexing approach that enables to retrieve apriori known graphs stored in graph databases based on graph and subgraph isomorphism more efficient than scanning the database sequentially on demand. Their approach bases on a decision tree that indexes permutations of adjacency matrices in terms of the row-column elements of these permutations. These permutations represent mappings between two graphs and, therefore, the problem of finding graph and subgraph isomorphisms is mapped to finding these permutations. Since the decision tree consists of redundant subtrees when realized naively, the concept of redirecting branches is introduced to aggregate redundant subtrees to one common subtree in the decision tree. However, the graphs that need to be indexed must be known apriori and a maintenance of the decision tree when graphs are added to, deleted from, or modified within the graph database is not considered.

Milo et al. [58] present an index structure called *T-Index* for graph-structured data. T-Index aims at supporting the evaluation of path expressions without scanning the graph data sequentially. T-Indexes are indexes that support the evaluation of path expressions that conform to a certain path expression template. The main concept of T-Index is that nodes in the graph data are grouped together in terms of equivalence classes in the index structure in a manner that nodes in the same equivalence class are reachable via the same path in the graph data. For that purpose, T-Index constructs a non-deterministic automaton whose states are equivalence classes and edges are transitions between objects in these equivalence classes. Note that the maintenance of the index is not in the scope of their paper. Furthermore, the T-index cannot be considered as view, because the index facilitates to evaluate path expressions that conform to a certain template faster, but these results are not materialized as for database views.

Cooper et al. [15] propose a path-based indexing approach called *Fabric* for semi-structured data such as XML documents. Their approach bases on Patricia Tries that are organized in layers for efficient lookup of queries. For indexing and querying the semi-structured data, queries are translated into prefix strings that are indexed with the help of Patricia Tries. The authors distinguish two kinds of queries: raw paths and refined paths. Raw paths are paths from the root element to leafs in terms of prefixed strings, while refined paths are special paths that are added to the index manually and reference directly the answer for certain queries. Thus, refined paths can be considered as a simple kind of a view. However, how the index and especially refined paths are maintained when the semi-structured data changes is not in the scope of the presented approach.

Chung et al. [12] propose the adaptive indexing approach *APEX* for graph-structured documents such as XML documents. APEX is a path-based indexing approach that enables to adapt the employed index structure concerning often stated queries. For that purpose, APEX employs index structures that complement each other. A graph-structured index summarizes the structural information of the indexed graph data, e.g. which node is referenced by other nodes. A hash tree enables fast access to nodes in the graph-structured index that contribute to the answers of stated queries. This hash tree is adapted when the query workload changes and other queries become frequent. Frequent queries are detected by a data mining approach for frequent label paths. Note that the adaption is triggered by changing query workloads and that the index is not maintained when the indexed graph data changes.

Kaushik et al. [48] propose to exploit local similarity of nodes in graph-structured data to reduce index sizes. They present the path-based indexing approach *A(k)-Index* as generalization of the 1-Index [58]. The main idea of their approach is to group index elements representing similar nodes (resp. node structures) of the graph-structured data together in the index in contrast to other indexing approaches. The main concept of their approach is the notion of bisimilarity and k-bisimilarity. The authors state, *"if two nodes u and v are k-bisimilar, then the set of [paths of length k] into these [data] nodes is identical"* [48]. Thus, these nodes can be grouped together in the index to reduce the index size. Subsequently, the retrieved candidate sets are validated to remove false positives especially for short paths. Note that the maintenance of the index is left for future work by the authors.

Yan et al. [78, 79] present a graph indexing graph called *gIndex* that uses frequent graph structures contained by graphs in the graph database as indexing feature. Thus, in contrast to other graph indexing approaches, *gIndex* does not employ a path-based indexing approach. The authors propose to use discriminative graph fragments for indexing, since these frequents reduce the search space better than other fragments and are in general stable what enables an incremental maintenance of the index. Their approach consists of an index construction and query processing step. The index enumerates graph structures and each indexed graph structure consists of a list of graphs in the graph database that contain the indexed graph structure. The query step computes a set of candidate graphs for a given query and, afterwards, checks whether the query is a real isomorphic subgraph of the candidate graph. To compute the most useful graph structure for indexing, a discrimination ratio is computed to decide whether a given graph structure is more discriminative than other graph structures. The authors consider the insertion and deletion of graphs to (resp. from) the graph database. However, they do not consider the modification of graphs already contained by the graph database. For the insertion and deletion of graphs, the discriminative graph fragments are identified within the graph and added (resp. removed) from the list of associated graphs for the discriminative feature in the index.

Srinivasa et al. [74] propose the label walk index that is a path-based indexing approach in the graph database system GRACE. The label walk index is a prefix tree that indexes sequences of node labels, i.e. walks (also called paths). Each index element consists of a list of identifiers of graphs that contain the substructure associated with the indexed path and the number of occurrences in these graphs. However, the authors do not address how the index is maintained in case indexed graphs change.

Yan et al. [80] present the graph indexing approach *Grafil* that aims at retrieving graphs stored in a graph database that are similar to the stated query graph. Their approach consists of two matrices called feature-graph matrix and edge-graph matrix. The feature-graph matrix stores the number of occurrences of features (i.e., subgraphs) per graph contained by the graph database. The feature-graph matrix is created during index construction and maintained when graphs are added to or removed from the graph database by adding or removing columns that represent these graphs from the matrix. The edge-feature matrix stores which features contain which edges. The edge-feature matrix is computed when the query is stated and is used to select features that contain the edges of relaxed query graphs. Query graphs are relaxed by removing edges. The resulting set of features is used to lookup graphs that contain these features. The resulting graph candidate set is verified by existing feature similarity measures and, if required, potential candidates are removed from the set of graphs that contain substructures similar to the stated query graph.

He et al. [41] present a graph indexing approach called *Closure-Tree* (C-tree for short). The main concept of their approach is an index tree that consists of nodes that represent graph closures. Graph closures *"capture the structural information of each graph"* [41] represented by child nodes in the tree, i.e. graph closures are bounding containers for a certain set of graphs. Leaf nodes in the index tree are graphs in the database. When querying for graphs in the graph database, the C-tree is traversed based on pseudo subgraph isomorphism as approximation of subgraph isomorphism to prune candidate graphs and derive a candidate answer set. Afterwards, subgraph isomorphism tests are performed on the remaining graphs in the candidate answer set. The index is maintained when graphs are inserted to and deleted from the graph database. However, the modification of graphs already contained by the graph database is not considered. Note that only subgraph queries and similarity queries are supported for efficient look up of graphs in the graph database that are subgraph isomorphic or similar to the query graph.

Williams et al. [77] present an approach for indexing graphs to improve the performance of subgraph isomorphism tests and similarity queries for graphs. The basic idea of their approach is to use graph decomposition graphs that enumerate all connected and induced subgraphs of a given graph. Each node (resp. induced subgraph) in the graph decomposition is hashed using the canonical code of the graph and the hash value is added to an index. The graph decomposition graphs of all graphs in the graph database are merged to one single graph database decomposition graph. A lookup function is used to look up nodes (resp. induced subgraphs) in the graph database decomposition graph. The lookup procedure consists of two steps. The first step computes the hash value of the query (graph) using its canonical code. The result of the first step is a set of candidate matches with their canonical codes for the query. In the second step the candidate matches are verified by comparing the canonical codes of candidates with the canonical code of the query. If the canonical code is exactly the same, then the candidate is a isomorphic match for the query. Note that the second step is required, because the hash function does not guarantee uniqueness and the index size is limited to a certain size that is set a priori. However, the index is constructed once and the authors do not comment on how the index is maintained when the graph database content changes.

Zhang et al. [81] present the graph indexing approach *TreePi* that uses frequent trees as index features instead of frequent subgraphs. During index construction frequent trees in graphs of a graph database are mined that are indexed using their canonical form. During query processing the query (graph) is partitioned into trees. These trees are used to retrieve all graphs in terms of a candidate set that contain these trees. This candidate set is pruned by selecting those trees, which fulfill a center distance constraint that is implied by the query. Afterwards, the remaining trees are used to reconstruct the query, if possible, to verify the result. This verification step replaces the expensive subgraph isomorphism test of other indexing approaches. Note that the described optimizations can only be employed, because subtrees are used instead of subgraphs and, therefore, the center of trees can be determined easily in contrast to graphs.

7.4. Graph Querying

In this section, we present existing graph querying approaches. We distinguish between model search (Sec. 7.4.1) and graph search (Sec. 7.4.2). While model search deals with the retrieval of models expressed in certain modeling languages (e.g., in Model-Driven Software Engineering), graph search deals with the retrieval of graph structures in a broader sense independently from certain modeling languages and the domain of software engineering.

7.4.1. Model Search

In this section, we present the state of the art in model search. In practice, a model repository *"provides storage facilities for models"* [45] in terms of an artifact storage. On top of such a model repository current model search engines create a search index by employing one single monolithic approach that is dedicated to the search task that needs to be supported. A query engine browses this search index, when a query is stated. Afterwards, the artifacts referenced by the retrieved search index results are delivered as query result. Queries can be expressed in terms of keywords, OCL expressions or patterns (query-by-example).

The approach of Gomes et al. [27] enables the case-based retrieval of UML models that are similar to a currently developed target design by means of similarity metrics. Their approach exploits synonyms of words (e.g. class names) and semantic relations between words to create the search index. Their approach focus on supporting UML packages, classes and interfaces and does not support other modeling languages.

MoScript [52] is a domain-specific language for querying and manipulating model repositories. MoScript employs OCL expressions to retrieve logical models from a mega-model, which capture physical models and their relationships [43]. These logical models need to be dereferenced to obtain the corresponding physical models. Thereby, the mega-model itself acts in terms of a search index. MoScript focuses on querying models with the help of OCL expressions, but MoScript does not consider views for the retrieval of models that fulfill certain properties.

The approach of Bozzon et al. [8] enables to search for models about applications developed by employing Model-Driven Engineering (MDE) practices. Their approach exploits the inner structure of models and distinguishes between a content processing flow to create a search index and a query processing flow to answer queries. The content processing flow extracts meaningful information to create the search index by extracting global meta data, splitting projects into smaller units, mining information to generate the search index, and performing a linguistic normalization. The query flow enables a keyword-based and context-based (query-by-example) search for models using the search index created by the content processing flow. However, their approach does not provide means to model the content processing flow, i.e. the generation of the search index individually, e.g. in terms of views.

Moogle [56] is a model search engine, which exploits meta-models of models. Therefore, *Moogle* enables to search for meta-information, since it is aware of the internal structure of models. In Moogle model search is mapped to text search using the full-text search engine Apache SOLR[4]. Therefore, Moogle creates model descriptors that conform to the schema of Apache SOLR. Thus, these model descriptors constitute the search index. However, also Moogle's procedure to create the search index is immutable and does not support different kinds of indexes in term of views.

[4]`https://lucene.apache.org/solr/` (last access: August 13th 2015).

7.4.2. Graph Search

Fan et al. [18] investigated incremental algorithms for graph pattern matching using graph simulation, bounded simulation, and subgraph isomorphism. They present several algorithms for incremental graph pattern matching. However, the algorithms of Fan et al. [18] require the concrete pattern graphs as input and do not assume black boxes.

Fan et al. [20] discuss how graph pattern queries can be answered using views without accessing base graphs. The authors provide algorithms that enable to a) determine whether a query can be answered with a given set of views without accessing the base graph, b) compute answers for a query efficiently given a set of views, and c) determine a minimal set of views which should be used for answering a query. The authors describe algorithms for the challenges described above using graph simulation and bounded graph simulation. The authors do not consider view maintenance.

7.5. Incremental Model-Driven Engineering

In the following sections, we summarize related work from the model-driven engineering domain that use incremental processing approaches for incremental graph pattern matching (Sec. 7.5.1), incremental model transformation (Sec. 7.5.2), and incremental model constraint evaluation (Sec. 7.5.3).

7.5.1. Incremental Graph Pattern Matching

Bergmann et al. [6] present an incremental graph pattern matching approach for the VIATRA2 framework. The authors adapted the concept of RETE networks [21] to pre-compute all matches for graph patterns with the result that matches are available in constant time (on condition that the RETE network processed all changes to models) when required to perform graph transformations. Their approach maps (partial) graph patterns to nodes in the RETE network and each node in the RETE network *"stores the set of tuples that conform to the pattern"* [6]. Thus, tuples are associated with matches for graph patterns. Note that their approach is limited to the topology of RETE networks and does not enable topologies of Gator networks [37].

The approach of Bergmann et al. [6] has been adapted in several other contexts. For example, EMF-IncQuery [5] uses the incremental graph pattern matching provided by the Viatra2 framework to answer model queries over EMF models instantly after all matches have been updated by the RETE network.

The performance of incremental pattern matchers that base on RETE networks depends on the topology of the employed RETE network as stated by Varró et al. [76] and Hanson et al. [37]. For this purpose, Varró et al. [76] present an approach for RETE network construction for incremental graph pattern matching. The approach employs a state-space search by taking the distance to the final topology (called unification point) and cost of a (partial) network topology into account. The benefit of their approach is that the cost function is customizable.

Furthermore, EMF-IncQuery has been used by Ráth et al. [68] to incrementally compute derived features in EMF models. Derived features are attributes and references whose values are non-persistent and, therefore, are computed from persistent features or other derived features. Model queries are used to compute values of derived features. They integrated EMF-IncQuery into the model code generated by EMF.

Giese et al. [24] present an approach for incremental model synchronization based on Triple Graph Grammars (TGG). Their algorithm enables incremental model synchronization due to explicit dependencies between correspondence nodes that reflect the execution order of TGG rules. When modification information of the synchronized models are available the synchronization starts at the correspondence node that is connected to modified model elements and traverses the DAG of correspondence nodes in a breadth-first search. Their algorithm reverts transformation steps when model elements have been deleted and applies transformation steps when model elements have been added. In case only attribute values have been changed, these attribute values are propagated between the synchronized models. Negative application conditions are not considered by their approach.

7.5.2. Model Transformation

Hearnden et al. [42] aim at incremental model transformation (also called live transformations) using SLD resolution trees as used for the execution of logic programs that represent the trace of a transformation execution that is used for change propagation between source and target models. Changes to the source model of the transformation leads to an update of the resolution tree and, therefore, these changes can be mapped to changes in the target model.

The approach of Ráth et al. [67] uses the RETE-based matcher of the Viatra2 framework (see Sec. 7.5.1) to enable live transformations by determining the match set that is impacted by a change.

Debreceni et al. [16] present how EMF-IncQuery and its RETE-based incremental graph pattern matcher is used for deriving view models from source models and incrementally synchronizing these view models upon source model changes. Their approach bases on a) annotated model queries that enable to specify that view models can dependent on other view models, b) traceability links that enable to lookup previously derived model elements in view models, and c) a notification mechanism that provides match set deltas for view synchronization when source models change.

Jouault et al. [46] present an incremental execution algorithm for live transformations in ATL. They propagate creations and deletions of source model elements to the target model by executing only those parts of the transformation that are impacted by the change. For that purpose, they employ a change tracking for models to keep track of modified model elements.

7.5.3. Model Constraint Evaluation

Egyed [17] presents an approach for incremental re-evaluation of model constraints based on model changes. His approach employs a model profiler that keeps track of model elements that have been traversed for evaluating a model constraint. Thus, changed model elements can be traced back to model constraints that must be re-evaluated due to these changes to model elements without requiring manual annotations of model constraints.

The approach of Groher et al. [30] extends the approach of Egyed [17] by focusing on changeable model constraints that trigger the incremental re-evaluation of these constraints.

The approach of Cabot et al. [10] rewrites OCL constraints based on model changes to lower computational complexity of the constraints for re-evaluation. For that purpose, their approach must be aware of the concrete OCL constraints and does not assume black box constraints.

In our former work, Seibel et al. [72] present a context-aware and modification-aware incremental approach for the maintenance of traceability links by employing maintenance steps in terms of creation and deletion rules. These rules can be considered as constraints between models of the same and different kinds. The approach bases on a traceability link reference model that defines traceability link types, which are instantiated and maintained by traceability maintenance rules in terms of creation and deletion rules. However, their approach is limited to the domain of traceability link maintenance.

Niere et al. [63] present a incremental processing approach for software design pattern recovery that focuses on detecting the first (and subsequent) software design patterns as fast as possible. Their approach returns matching results in incremental parts comparable with an iterator, but does not support the incremental re-evaluation of detected software design pattern in case the abstract syntax graph of Java source code changes.

7.6. Summary

In this section, we summarize the presented related work and discuss differences to the view maintenance approach presented in this technical report. Discrimination networks have been used for production systems and active relational database systems for rule condition testing and constraint checking. Several kinds of discrimination networks have been investigated and it turned out that neither RETE networks [21] nor TREAT networks [61] are best in general. Instead, Gator networks [37] with an arbitrary network topology are optimal in general, when the network topology is optimized for often stated queries. However, a) discrimination networks found application in relational databases only [21, 61, 37], b) non-relational problems are mapped to the relational case [6], and c) solely RETE networks have been employed for graph grammars by Bunke et al. [9]. During our intensive literature research, we found no approach that employed the concept of Gator networks [37] to graph grammars or view maintenance for graph databases.

Moreover, the discrimination networks RETE and TREAT found application in relational databases [21, 61], but we know no approach that employs these kinds of discrimination networks for view maintenance of graph databases. Furthermore, view maintenance for graph databases seems to be rarely discussed in literature at all although there are several challenges as discussed in this technical report and as also stated by Zhuge et al. [82].

Instead of view maintenance for graph databases, graph indexes have been studied extensively such as path-based indexes (e.g., DataGuides [26], T-Index [58], Fabric [15], A(k)-Index [48]) and indexes that base on frequent substructures (e.g., gIndex [79], Grafil [80], Closure-Tree [41]). However, these approaches mainly focus on index construction and query retrieval, but do not aim at an efficient maintenance of the index. The maintenance task is often mapped to the deletion and re-insertion of *complete* graphs into the graph database, when graphs are modified. But first and foremost, graph indexes aim at accelerating query evaluation and do not maintain answers of pre-computed queries as it is the case for view graphs. Thus, in general graph indexes are scanned (sequentially) to answer queries, while view graphs return query answers in constant time, when the view graph is up-to-date, i.e. maintained immediately when changes to base graphs occur.

Graph query approaches such as model search approaches either employ a general-purpose search index (e.g., MoScript [52], Moogle [56]) or a purpose-specific search index (e.g., Gomes et al. [27], Bozzon et al. [27]). Furthermore, especially model search approaches with purpose-specific search indexes are limited to certain modeling languages such as Unified Modeling Language (UML). Note that both kinds of model search approaches do not pre-compute answers in terms of materialized views as it is also the case for graph indexing approaches.

The presented approaches have been adapted to enable incremental graph pattern matching (e.g., Viatra 2 [6], EMF IncQuery [5]), incremental model constraint checking (e.g., Egyed et al. [17], Groher et al. [17], Cabot et al. [17]), and incremental model transformations (e.g., Hearnden et al. [42], Ráth et al. [67], Jouault et al. [46]) in Model-Driven Engineering. However, only Rete networks [21] have been adapted for graph grammars [9, 6] by mapping graph patterns to tuples. To the best of our knowledge, no approach exists that adapted Gator networks for the use with graph grammars or view graph maintenance for graph databases as presented in this technical report.

8. Conclusion and Future Work

In this technical report, we presented a view maintenance approach for graph databases. The contribution of our approach is twofold: a) we described a view definition language for graph databases (Sec. 4) and b) we described batch and incremental algorithm for maintaining view graphs when base graphs change (Sec. 5). The view definition language bases on a view reference graph that describes the content of view graphs on type level (Sec. 4.1), view modules (Sec. 4.2) that embody view definitions in terms of graph patterns (Sec. 4.4), and view models (Sec. 4.3) that describe dependencies between view modules. The view graph maintenance procedures either process the complete base graph in batch mode (Sec. 5.2 and 5.3) or exploit modification information of base graphs (Sec. 5.4 and 5.5).

Our evaluation (Sec. 6) shows that the incremental maintenance procedures outperform the batch maintenance procedure. While the execution time required by the batch maintenance procedure depends on the base graph sizes, the execution time required by the incremental maintenance procedure depends only on the number of modifications made to base graphs. Thus, in contrast to the batch maintenance procedure, the incremental maintenance procedure scales concerning the size of base graphs, because the effort required to perform an incremental maintenance is independent of the sizes of base graphs.

In our related work discussion, we emphasized that view maintenance for graph databases has been neglected by the research community and is not supported by current graph database implementations. Furthermore, we emphasized that the most generalized form of discrimination networks, i.e. Gator networks, have not been adapted to graph databases for view maintenance although Gator networks can outperform RETE and TREAT networks as has been shown by Hanson et al. [37] for the relational case. To the best of our knowledge, we presented the first approach that mapped Gator networks from relational databases to graph databases. Moreover, discrimination networks do not consider recursion as we do.

In our future work, we plan to proceed with the parallel execution of view modules that can be executed independently from each other. Furthermore, we are going to investigate how to organize the structure of view models to reduce the view maintenance costs. For that purpose, we plan to take runtime information into account to adaptively change the structure of view modules, in contrast to existing approaches that create discrimination networks at compile time.

References

[1] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. MOVIE: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47(2):131–166, 2003.

[2] Renzo Angles. A Comparison of Current Graph Database Models. In *Proceedings of the 28th International Conference on Data Engineering*, pages 171–177. IEEE, April 2012.

[3] Hermann Balsters. Modelling Database Views with Derived Classes in the UML/OCL-framework. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2003.

[4] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying Graph Patterns. In *Proceedings of the 30th Symposium on Principles of Database Systems*, PODS '11, pages 199–210. ACM, 2011.

[5] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental Evaluation of Model Queries over EMF Models. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2010.

[6] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental Pattern Matching in the Viatra Model Transformation System. In *Proceedings of the 3rd International Workshop on Graph and Model Transformations*, GRaMoT '08, pages 25–32. ACM, 2008.

[7] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In *Proceedings of the International Conference on Management of Data*, SIGMOD '86, pages 61–71. ACM, 1986.

[8] Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. Searching Repositories of Web Application Models. In *Web Engineering*. Springer, 2010.

[9] H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 1991.

[10] Jordi Cabot and Ernest Teniente. Incremental Evaluation of OCL Constraints. In *Advanced Information Systems Engineering*, pages 81–95. Springer, 2006.

[11] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589. Morgan Kaufmann Publishers Inc., 1991.

[12] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 121–132. ACM, 2002.

[13] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of the International Conference on Management of Data*, pages 469–480. ACM, 1996.

[14] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting Multiple View Maintenance Policies. In *Proceedings of the 1997 International Conference on Management of Data*, SIGMOD '97, pages 405–416. ACM, 1997.

[15] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 341–350. Morgan Kaufmann Publishers Inc., 2001.

[16] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven Incremental Synchronization of View Models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '14, pages 31–38. ACM, 2014.

[17] Alexander Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering*, pages 381–390. ACM, 2006.

[18] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental Graph Pattern Matching. In *International Conference on Management of Data 2011*, pages 925–936. ACM, 2011.

[19] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph Pattern Matching: From Intractable to Polynomial Time. *Proc. VLDB Endow.*, 3(1):264–275, Sep 2010.

[20] Wenfei Fan, Xin Wang, and Yinghui Wu. Answering Graph Pattern Queries Using Views. In *Proceedings of 30th International Conference on Data Engineering*, pages 184–195. IEEE, March 2014.

[21] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[24] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer, Oct 2006.

[25] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental Updates for Materialized OQL Views. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Deductive and Object-Oriented Databases*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1997.

[26] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 436–445. Morgan Kaufmann Publishers Inc., 1997.

[27] Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, and Carlos Bento. Using WordNet for Case-based Retrieval of UML Models. *AI Communications*, 17(1):13–23, Jan 2004.

[28] Nathan Goodman and Oded Shmueli. Transforming Cyclic Schemas into Trees. In *Proceedings of the 1st Symposium on Principles of Database Systems*, PODS '82, pages 49–54. ACM, 1982.

[29] Timothy Griffin, Leonid Libkin, and Howard Trickey. An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. *Transactions on Knowledge and Data Engineering*, 9(3):508–511, May 1997.

[30] Iris Groher, Alexander Reder, and Alexander Egyed. Incremental Consistency Checking of Dynamic Constraints. In *Fundamental Approaches to Software Engineering*, pages 203–217. Springer, 2010.

[31] Ashish Gupta and José A. Blakeley. Using Partial Information to Update Materialized Views. *Information Systems*, 20(8):641–662, 1995.

[32] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering*, 18(2):3–18, 1995.

[33] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the International Conference on Management of Data*, SIGMOD '93, pages 157–166. ACM, 1993.

[34] Eric N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proceedings of the International Conference on Management of Data*, SIGMOD '92, pages 49–58. ACM, 1992.

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

[36] Eric N. Hanson, Sreenath Bodagala, and Ullas Chadaga. Optimized Trigger Condition Testing in Ariel using Gator Networks. Technical Report TR-97-021, University of Florida, November 1997.

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

[38] Eric N. Hanson, Sreenath Bodagala, Mohammed Hasan, Goutam Kulkarni, and Jayashree Rangarajan. Optimized Rule Condition Testing in Ariel using Gator Networks. Technical Report TR-95-027, University of Florida, October 1995.

[39] Eric N. Hanson and Mohammed Hasan. Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing. Technical Report TR-93-036, University of Florida, December 1993.

[40] John V. Harrison and Suzanne W. Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, pages 56–65. Unknown Psublisher, 1992.

[41] Huahai He and Ambuj K. Singh. Closure-Tree: An Index Structure for Graph Queries. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 38–50. IEEE, 2006.

[42] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, pages 321–335. Springer, 2006.

[43] Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. In *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling*, 2011.

[44] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri and Heike Wehrheim, editors, *Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *FMOODS'06*, pages 171–185. Springer, 2006.

[45] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[46] Frédéric Jouault and Massimo Tisi. Towards Incremental Execution of ATL Transformations. In *Proceedings of the Third International Conference on Theory and Practice of Model Transformations*, ICMT'10, pages 123–137. Springer, 2010.

[47] Salim Jouili and Valentin Vansteenberghe. An Empirical Comparison of Graph Databases. In *Proceedings of the International Conference on Social Computing 2013*, pages 708–715. IEEE, Sep 2013.

[48] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140. IEEE, Mar 2002.

[49] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *Proceedings of the 29th International Conference on Data Engineering*, pages 997–1008. IEEE, April 2013.

[50] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented database system for (software) engineering applications. In *Proceeding*

of the Sixth International Workshop on Computer-Aided Software Engineering, pages 272–286. IEEE, Jul 1993.

[51] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a Graph Oriented (Software) Engineering Database System. *Information Systems*, 20(1):21–51, Mar 1995.

[52] Wolfgang Kling, Frederic Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Proceedings of the 4th International Conference on Software Language Engineering*, pages 180–200. Springer, 2012.

[53] Harumi A. Kuno and Elke A. Rundensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *Transactions on Knowledge and Data Engineering*, 10(5):768–792, Sep 1998.

[54] Ho Soo Lee and Marshall I. Schor. Match Algorithms for Generalized Rete Networks. *Artificial Intelligence*, 54(2):249–274, 1992.

[55] Jixue Liu, Millist Vincent, and Mukesh Mohania. Maintaining Views in Object-Relational Databases. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 102–109. ACM, 2000.

[56] Daniel Lucrédio, Renata Fortes, and Jon Whittle. MOOGLE: a metamodel-based model search engine. *Software & Systems Modeling*, 11(2):183–208, 2010.

[57] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

[58] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 277–295. Springer, 1999.

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

[60] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, volume 1 of *AAAI'90*, pages 685–692. AAAI Press, 1990.

[61] Daniel P. Miranker and Bernie J. Lofaso. The Organization and Performance of a TREAT-Based Production System Compiler. *Transactions on Knowledge and Data Engineering*, 3(1):3–10, March 1991.

[62] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized View Selection and Maintenance Using Multi-query Optimization. In *Proceedings of the International Conference on Management of Data*, SIGMOD '01, pages 307–318. ACM, 2001.

[63] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards Pattern-based Design Recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348. ACM, 2002.

[64] Object Management Group (OMG). OMG Unified Modeling Language (Version 2.3), May 2010. Specification at `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/` (last access: September 10th 2015).

[65] Object Management Group (OMG). OMG Systems Modeling Language (Version 1.3), Jun 2012. Specification at `http://www.omg.org/spec/SysML/1.3/PDF` (last access: September 10th 2015).

[66] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *Transactions on Knowledge and Data Engineering*, 3(3):337–341, Sep 1991.

[67] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live Model Transformations Driven by Incremental Pattern Matching. In *Proceedings of the 6th International Conference on Theory and Practice of Model Transformations*, pages 107–121. Springer, 2008.

[68] István Ráth, Ábel Hegedüs, and Dániel Varró. Derived Features for EMF by Integrating Advanced Model Queries. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ECMFA'12, pages 102–117. Springer, 2012.

[69] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of the International Conference on Management of Data*, SIGMOD '96, pages 447–458. ACM, 1996.

[70] Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, and Beth R. Tibbitts. Advances in Rete Pattern Matching. In *Proceedings of 5th National Conference on Artificial Intelligence*, pages 226–232. AAAI Press, 1986.

[71] Andy Schürr. Progress: A VHL-language based on graph grammars. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 641–659. Springer, 1991.

[72] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528, 2010.

[73] Oded Shmueli and Alon Itai. Maintenance of Views. In *Proceedings of the International Conference on Management of Data*, SIGMOD '84, pages 240–255. ACM, 1984.

[74] Srinath Srinivasa and Martin Maier. 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, Dec 2005.

[75] The Neo4j Team. The Neo4j Manual v2.2.1. Specification at `http://neo4j.com/docs/` (last access: September 10^{th}2015).

[76] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations*, volume 7909 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2013.

[77] D.W. Williams, Jun Huan, and Wei Wang. Graph Database Indexing Using Structured Graph Decomposition. In *Proceedings of the 23^{rd} International Conference on Data Engineering*, pages 976–985. IEEE, April 2007.

[78] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the International Conference on Management of Data*, SIGMOD '04, pages 335–346. ACM, 2004.

[79] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph Indexing Based on Discriminative Frequent Structure Analysis. *Transactions on Database Systems*, 30(4):960–993, Dec 2005.

[80] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure Similarity Search in Graph Databases. In *Proceedings of the International Conference on Management of Data*, SIGMOD '05, pages 766–777. ACM, 2005.

[81] Shijie Zhang, Meng Hu, and Jiong Yang. TreePi: A Novel Graph Indexing Method. In *Proceedings of the 23^{rd} International Conference on Data Engineering*, pages 966–975. IEEE, April 2007.

[82] Yue Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering*, pages 116–125. IEEE, Feb 1998.

A. Metamodel

Fig. A.1 depicts our comprehensive meta-model that describes the view model, view modules and view definition language. We summarize the purpose of each metamodel element in terms of Table A.1.

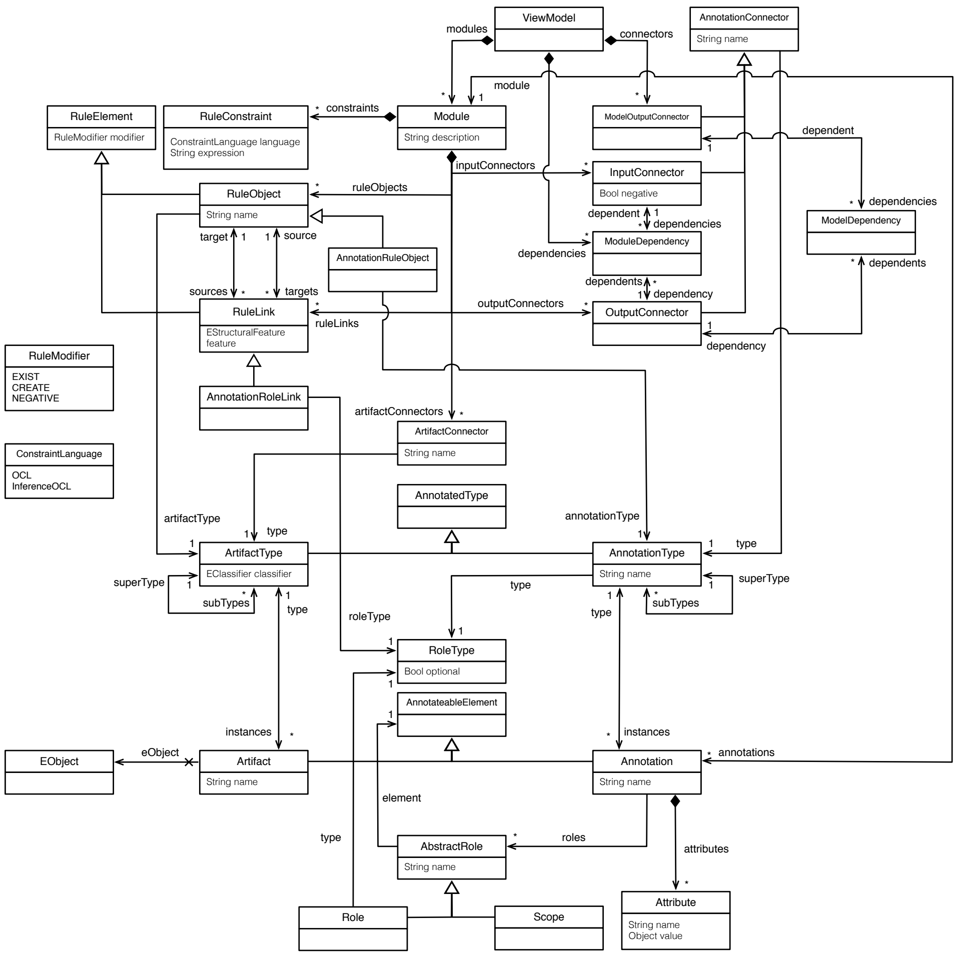


Figure A.1.: Meta model of view definition and annotation language

Table A.1.: Description of metamodel elements

Metamodel Element	Description
`ViewModel`	A `ViewModel` contains `Modules` and describes the interplay of these `Modules` to derive an overall view graph.
`Module`	A `Module` is a container that contains the view definition in terms of a graph pattern. `Modules` describe in terms of connectors which kinds of artifacts and annotations are processed and which kinds of annotations are created when deriving a view graph.
`AnnotationConnector`	The `AnnotationConnector` class is the super class of concrete connectors such as `InputConnectors`, `OutputConnectors`, and `ModelOutputConnectors` and consists of a `name` attribute. Furthermore, each `AnnotationConnector` references an `AnnotationType` to describe which kinds of annotations are processed or created by the `AnnotationConnector`.
`ArtifactConnector`	An `ArtifactConnector` describes which kinds of artifacts are processed by a `Module`. For that purpose, each `ArtifactConnector` references an `ArtifactType`.
`InputConnector`	An `InputConnector` describes which kinds of annotations are processed by a `Module`. For that purpose, each `InputConnector` references an `AnnotationType`.
`OutputConnector`	An `OutputConnector` describes which kinds of annotations are produced by a `Module`. For that purpose, each `OutputConnector` references an `AnnotationType`.
`ModelOutputConnector`	An `ModelOutputConnector` describes which annotations are created by the overall `ViewModel` and, therefore, references an `AnnotationType`.
`ModuleDependency`	A `ModuleDependency` describes the flow of created annotations between `Modules`.
`ModelDependency`	A `ModelDependency` describes which annotations are forwarded to `ModelOutputConnectors` of the `ViewModel`. Therefore, a `ModelDependency` describes which annotations are made visible outside the `ViewModel`.
`RuleElement`	The `RuleElement` class is the super class of graph pattern elements that define the graph query used as view definition. Each `RuleElement` consists of a `RuleModifier`.

RuleModifier	A RuleModifier describes whether a RuleElement belongs to the positive part of the graph pattern (cf. EXIST modifier), belongs to the negative part of the graph pattern (cf. NEGATIVE modifier), or will be created if the graph pattern matches (cf. CREATE modifier).
RuleObject	A RuleObject represents Artifacts (nodes) in base graphs that are part of a graph pattern.
AnnotationRuleObject	An AnnotationRuleObject represents Annotations in view graphs that are part of a graph pattern.
RuleLink	A RuleLink between RuleObjects represents references (cf. EStructuralFeature attribute) between Artifacts (nodes) in base graphs that are part of a graph pattern.
AnnotationRoleLink	An AnnotationRoleLink between AnnotationRuleObjects and RuleObjects describes the Role that must exist as part of the graph pattern or will be created between both an Artifact/Annotation and Annotation if the graph pattern matches.
RuleConstraint	Additionally, a graph pattern can consist of additional rule constraints in terms of OCL or InferenceOCL expressions. For that purpose, a RuleConstraint defines the language used to formulate an expression statement.
ConstraintLanguage	We distinguish two constraint languages: OCL and InferenceOCL. InferenceOCL is an extension of OCL and provides additional procedures that are free from side-effects, e.g. a procedure to match regular expressions.
AnnotatedType	The class AnnotatedType is the super class for ArtifactType and AnnotationType.
ArtifactType	An ArtifactType describes the type of nodes in base graphs. For that purpose, an ArtifactType has an EClassifier attribute that references the classifier. Note that a single inheritance hierarchy of ArtifactTypes exists.
AnnotationType	An AnnotationType describes the type of annotations in view graphs. For that purpose, each AnnotationType has a name attribute and references RoleTypes to describe which kinds of Artifacts and Annotations are annotated by annotations of a certain AnnotationType. Note that a single inheritance hierarchy of AnnotationTypes exists.

RoleType	A RoleType describes the kind of a role in which an Artifact or Annotation participates in a certain Annotation. A RoleType defines which kinds of Artifacts or Annotations are annotated by a Role of a certain RoleType.
AnnotateableElement	The class AnnotateableElement is the super class for all classes that describe nodes in base graphs and views graphs that can be annotated.
Artifact	An Artifact represents a node of a certain ArtifactType in base graphs and can be annotated by annotations.
Annotation	An Annotation of a certain AnnotationType marks a match of a graph pattern. Annotations themself can be annotated by other Annotations. Note that the Annotation class has a reference (cf. module reference) to the Module class to keep track of which Module created a certain Annotation.
AbstractRole	The AbstractRole is the super class for two different kinds of roles and consists of a name to be able to distinguish multiple roles of the same RoleType.
Role	A Role describes in which kind of role (i.e. RoleType) a certain Artifact or Annotation participates in an Annotation.
Scope	A Scope is a role that does not consists of a RoleType. Instead, Scopes are used to keep track of Artifacts or Annotations that lead to graph pattern matches without specifying their Role of a certain RoleType in view definitions.
Attribute	Attributes can be part of Annotations to store additional key-value pairs.
EObject	An EObject is the base class in the Eclipse Modeling Framework. An instance of the EObject class is a node in base graphs represented by Artifacts (cf. eObject reference).

B. View Graph Maintenance Algorithms

In the following sections, we present our maintenance algorithms in pseudocode. We distinguish between pseudocode for naive batch maintenance (Sec. B.1), batch maintenance with preservation (Sec. B.2), and incremental maintenance (Sec. B.3) in black box and white box mode.

B.1. Naive Batch Maintenance

Algorithm 1 shows the overall maintenance procedure for naive batch maintenance. All annotations are deleted (l. 2) and created from scratch again (l. 3).

Algorithm 1.: Naive batch maintenance

```
1: procedure BatchNaiveMaintenance(viewmodel)
2:     remove all annotations from all view graphs
3:     BatchCreate(viewmodel)
```

Algorithm 2 shows the Create step of the batch maintenance procedure in more detail. The view model is iterated considering recursion cycles in view models (l. 5). For each iterated view module relevant artifacts (l. 8) (cf. algorithm 3) and annotations (l. 11) (cf. 4) are collected. Relevant artifacts and annotations are artifacts and annotations that consist of a (sub) type as specified by module input connectors. Afterwards, these artifacts and annotations are passed to the view module, which searches for matches of the graph pattern specified within the module (l. 13). If the module is a fixpoint module and created new annotations, all view modules in the corresponding recursion cycle are executed again (l. 5).

Algorithm 3 describes that the type of each artifact input connector (l. 3) is determined to look up all artifacts with a certain (sub) type (l. 5). Note that the lookup of artifacts with a certain artifact type can be performed efficiently due

Algorithm 2.: Batch maintenance Create step

```
1: procedure BatchCreate(viewmodel)
2:     iterator := getModelIterator(viewmodel)
3:     createdAnnotations := false
4:     created := false
5:     while iterator.hasNext(created) do
6:         module := iterator.next(created)
7:         //Get all artifacts with artifact types (incl. subtypes) as specified by artifact connectors of the module.
8:         artifacts := ArtifactsByType(module)
9:         //Get all annotations that have an annotation type (incl. subtypes) as specified by annotation input
10:        //connectors of the module and were created by dependency modules.
11:        annotations := AnnotationsCreatedByDependencyModules(module)
12:        //Search for graph pattern matches for passed artifacts and annotations.
13:        created := module.create(artifacts,annotations)
14:        createdAnnotations |= created
15:    return createdAnnotations
```

to the `instances` reference between the `ArtifactType` and `Artifact` class in the metamodel (Fig. A.1).

Algorithm 3.: Get all artifacts that are relevant for module

```
1: procedure ArtifactsByType(module)
2:     artifacts := empty
3:     foreach artifact input connector of module do
4:         artifactType := connector.type
5:         artifacts.add(artifactType.instances) //incl. instances of sub types (simplified)
6:     return artifacts
```

Algorithm 4 describes that all annotations created by all dependency modules (l. 3) are collected. Note that the lookup of annotations created by a certain module can be performed efficiently due to the `annotations` reference between the `Module` and `Annotation` class in the metamodel (Fig. A.1).

Algorithm 4.: Get all annotations that are relevant for module

```
1: procedure AnnotationsCreatedByDependencyModules(module)
2:     annotations := empty
3:     foreach annotation input connector of module do
4:         foreach dependencyModule connected to input connector do
5:             annotations.add(dependencyModule.annotations)
6:     return annotations
```

B.2. Batch Maintenance with Preservation

Algorithm 5 shows the overall maintenance procedure for batch maintenance with preservation. The algorithm executes an UDC cycle (l. 4 - 6) until no additional annotations are created during the CREATE step. If the CREATE maintenance step creates new annotations (l. 6), an additional UDC cycle is required (l. 3) to support complex NACs.

Algorithm 5.: Batch maintenance with preservation

```
1: procedure BatchPreserve(viewmodel)
2:     recheckRequired := true
3:     while recheckRequired do
4:         BatchUpdate(viewmodel)
5:         BatchDelete(viewmodel)
6:         recheckRequired := BatchCreate(viewmodel) //cf. l. 15 in algorithm 2
```

Algorithm 6 shows the UPDATE maintenance procedure for batch maintenance mode with preservation in more detail. While iterating the view model (l. 4) annotations previously created by the current view module are collected (l. 7) (cf. algorithm 8) and passed to this view module for checking whether the matches that are marked by these annotations still exist (l. 10). If the match marked by an annotation does not exist anymore, the module sets the annotation obsolete by removing all artifacts and annotations from the roles of the annotation. Otherwise, the annotation is preserved. Recursion cycles are repeated until fixpoint modules do not update annotations anymore (l. 4 and l. 10).

Algorithm 6.: Batch maintenance UPDATE step

```
1: procedure BatchUpdate(viewmodel)
2:     iterator := getModelIterator(viewmodel)
3:     updated := false
4:     while iterator.hasNext(updated) do
5:         module := iterator.next(updated)
6:         //Returns all annotations that were created by passed module.
7:         annotations := AnnotationsCreatedByModule(module)
8:         //Checks whether the graph pattern matches marked by annotations still exist.
9:         //If yes, the annotation is preserved. Otherwise, the annotation is set obsolete.
10:        updated := module.update(annotations)
```

Algorithm 7 shows the Delete maintenance procedure for batch maintenance mode with preservation in more detail. While iterating the view model (l. 4) annotations previously created by the current view module are collected (l. 7) (cf. algorithm 8) and passed to this view module for checking whether the matches that are marked by these annotations do not exist anymore and, thus, the annotations are obsolete and must be deleted. If an annotation is obsolete, the module deletes the annotation from the view graph (l. 10). Recursion cycles are repeated until fixpoint modules do not delete annotations anymore (l. 4 and l. 10).

Algorithm 7.: Batch maintenance Delete step

```
 1: procedure BatchDelete(viewmodel)
 2:     iterator := getModelIterator(viewmodel)
 3:     deleted := false
 4:     while iterator.hasNext(deleted) do
 5:         module := iterator.next(deleted)
 6:         //Returns all annotations that were created by passed module.
 7:         annotations := AnnotationsCreatedByModule(module)
 8:         //Check whether the graph pattern matches marked by annotations are obsolete.
 9:         //If yes, the annotation is deleted. Otherwise, the annotation is preserved.
10:         deleted := module.delete(annotations)
```

Algorithm 8 shows that annotations that were created by a certain view module can be retrieved by traversing the `annotations` reference owned by `Modules` (cf. Fig. A.1).

Algorithm 8.: Determine annotations created by module (cf. metamodel depicted by Fig. A.1)

```
1: procedure AnnotationsCreatedByModule(module)
2:     //cf. annotations reference between Module and Annotation class in metamodel
3:     return module.annotations
```

Algorithm 9 shows that the module that created a certain annotation can be retrieved by traversing the `module` reference owned by `Annotations` (cf. Fig. A.1).

Algorithm 9.: Determine module that is responsible for maintaining a certain annotation (cf. metamodel depicted by Fig. A.1)

```
1: procedure GetResponsibleModule(annotation)
2:     //cf. module reference between Annotation and Module class in metamodel
3:     return annotation.module
```

B.3. Incremental Maintenance

Algorithm 10 shows the overall algorithm for incremental maintenance in more detail. The incremental maintenance requires the view model, information about the modification of base graphs (added, deleted, and modified nodes), and whether the view modules are considered as black boxes or white boxes. When white box semantic is employed, the graph pattern specified within the view modules is taken into account, in contrast to black box semantic.

First, suspicious annotations are derived from the provide modification events of base graphs (l. 3) and passed to the incremental Update procedure (l. 4). Afterwards, obsolete annotations are derived from the provided modification events of base graphs (l. 6) and passed to the incremental Delete procedure (l. 7). Finally, the incremental Create procedure is triggered to check whether new matches exist (l. 8).

Algorithm 10.: Incremental maintenance

```
1: procedure IncrementalMaintenance(viewmodel,modificationEvents,mode)
2:     //Suspicious annotations are annotations that reference modified artifacts.
3:     suspiciousAnnotations := DetermineSuspiciousAnnotations(modificationEvents)
4:     IncrementalUpdate(suspiciousAnnotations)
5:     //Obsolete annotations are annotations with at least one role that does not reference an artifact or annotation anymore.
6:     obsoleteAnnotations := DetermineObsoleteAnnotations(modificationEvents)
7:     IncrementalDelete(obsoleteAnnotations)
8:     IncrementalCreate(viewmodel,modificationEvents,mode)
```

Algorithm 11 describes how suspicious annotations are determined. While iterating captured modification events of base graphs (l. 3), the algorithm checks whether the event reports the modification of an artifact in base graphs (l. 4). If yes, all annotations attached to this artifact are considered as suspicious (l. 6), because the modification of the artifact may lead to the case that matches marked by these annotations do not exist anymore. Thus, a rechecking of all matches described by annotations in which the modified artifact is involved is required to ensure that base graphs and view graphs remain consistent. An artifact is considered as modified when a) attributes of the artifact are modified, or b) another artifact is added to or removed from references owned by the artifact, or c) the artifact is added to or removed from a reference owned by another artifact.

Algorithm 11.: Determine suspicious annotations

```
1: procedure DETERMINESUSPICIOUSANNOTATIONS(modificationEvents)
2:     annotations := empty
3:     foreach event in modificationEvents do
4:         if event type = MODIFIEDARTIFACT then
5:             artifact := artifact considered by event
6:             foreach annotation attached via roles to artifact do
7:                 annotations.add(annotation)
8:     return annotations
```

Algorithm 12 describes how obsolete annotations are determined. While iterating captured modification events of base graphs (l. 3), the algorithm checks whether the event reports the deletion of an artifact in base graphs (l. 4). If yes, all annotations attached to this artifact are considered as obsolete (l. 6), because the deletion of the artifact leads to the case that the matches marked by these annotations do not exist anymore and the annotations must be deleted to ensure that base graphs and view graphs remain consistent. Artifacts are considered as deleted when they are removed from base graphs, i.e. they are not reachable via edges within base graphs anymore.

Algorithm 12.: Determine obsolote annotations

```
1: procedure DETERMINEOBSOLETEANNOTATIONS(modificationEvents)
2:     annotations := empty
3:     foreach event in modificationEvents do
4:         if event type = DELETEDARTIFACT then
5:             artifact := artifact considered by event
6:             foreach annotation attached via roles to artifact before deletion do
7:                 annotations.add(annotation)
8:     return annotations
```

Algorithm 13 describes the UPDATE step of the incremental maintenance procedure. The algorithm determines the module that originally created the suspicious annotation (l. 3) (cf. algorithm 9). Afterwards, the responsible module checks whether the match marked by the suspicious annotation still exists (l. 6). If the annotation becomes obsolete, the incremental Delete step is triggered for this annotation (l. 8). Otherwise, all dependent annotations are checked whether they became obsolete by calling the incremental Update procedure for all dependent annotations (l. 11), because dependent modules are allowed to define additional conditions over artifacts that are part of matches retrieved from dependency modules.

Algorithm 13.: Incremental maintenance Update step

```
 1: procedure IncrementalUpdate(suspiciousAnnotations)
 2:     foreach suspiciousAnnotation in suspiciousAnnotations do
 3:         module := GetResponsibleModule(suspiciousAnnotation)
 4:         //Checks whether the graph pattern match marked by the suspicious annotation still exists.
 5:         //If yes, annotation is preserved. Otherwise, the annotation is set obsolete.
 6:         module.update(suspiciousAnnotation)
 7:         if suspiciousAnnotation became obsolete then
 8:             IncrementalDelete(suspiciousAnnotation)
 9:         else
10:             foreach dependentAnnotation of suspiciousAnnotation do
11:                 IncrementalUpdate(dependentAnnotation)
```

Algorithm 14 shows the Delete step of the incremental maintenance procedure. The algorithm determines the module that originally created the annotation (l. 3). Afterwards, the module deletes the annotation, if the annotation is obsolete (l. 6). If the annotation was deleted (l. 7), all dependent annotations become obsolete as well and are deleted by calling the incremental Delete procedure for all dependent annotations (l. 9).

Algorithm 14.: Incremental maintenance Delete step

```
1: procedure IncrementalDelete(obsoleteAnnotations)
2:     foreach obsoleteAnnotation in obsoleteAnnotations do
3:         module := GetResponsibleModule(obsoleteAnnotation)
4:         //Checks whether the annotation is obsolete.
5:         //If yes, annotation is deleted. Otherwise, the annotation is preserved.
6:         module.delete(obsoleteAnnotation)
7:         if obsoleteAnnotation was deleted then
8:             foreach dependentAnnotation of obsoleteAnnotation do
9:                 IncrementalDelete(dependentAnnotation)
```

Algorithm 15 shows the CREATE step of the incremental maintenance procedure. While iterating the view model considering recursion cycles (l. 4), for each module the initial scope is derived from captured modification events of base graphs (l. 6) that is required to compute all artifacts and annotations that must to be checked for new matches. Depending on whether black box mode (l. 8) or white box mode (l. 10) is employed different scope computation procedures are employed that differ in the issue whether the graph pattern specified within the module is taken into account or not to compute the scope that is passed to the module for incrementally searching for new matches. Afterwards, the artifacts and annotations in the computed scope are passed to the CREATE procedure of the module to search for new matches of graph patterns (l. 13). If the module consists of dependent modules and these dependent modules use annotations created by the current module in negative manner (l. 15), suspicious annotations are determined (l. 16) and passed to the update procedure of dependent modules (l. 17), because created annotations mark matches that may dissatisfy matches of dependent modules due to NACs that are not fulfilled anymore.

Algorithm 15.: Incremental maintenance CREATE step

```
 1: procedure INCREMENTALCREATE(viewmodel,modificationEvents,mode)
 2:     iterator := getModelIterator(viewmodel)
 3:     created := false
 4:     while iterator.hasNext(created) do
 5:         module := iterator.next(created)
 6:         startScope := DETERMINESTARTSCOPE(modificationEvents,module)
 7:         if mode = BLACKBOX then
 8:             scope := DETERMINEBLACKBOXSCOPE(startScope)
 9:         else if mode = WHITEBOX then
10:             scope := DETERMINEWHITEBOXSCOPE(startScope)
11:         artifacts := artifacts contained by scope
12:         annotations := annotations contained by scope
13:         created := module.create(artifacts,annotations)
14:         foreach dependentModule of module do
15:             if dependentModule uses annotations created by module in negative manner then
16:                 suspiciousAnnotations := SUSPICIOUSANNOTATIONSDUETOCREATEDANNOTATIONS(module, annotations currently created by module)
17:                 dependentModule.update(suspiciousAnnotations)
```

Algorithm 16 shows how the start scope is derived from modification events of base graphs. While iterating captured modification events, created and modified artifacts are added to the scope, if the artifact has an artifact (sub) type as specified by input connectors of the module (l. 6), because added and modified artifacts can lead to new matches when PACs are considered. Furthermore, each annotation created by dependency modules during the current maintenance cycle is added to the scope, because they may lead to new matches of dependent modules (l. 9). Finally, all artifacts and annotations that were referenced by annotations that have been deleted by dependency modules are added to the scope (l. 13), because these artifacts and annotations may belong to matches that require the satisfaction of a NAC. This NAC may become true due to annotations deleted by dependency modules.

Algorithm 16.: Determine start scope from modification events and annotations created and deleted by dependency modules

```
1: procedure DetermineStartScope(modificationEvents,module)
2:     scope := empty
3:     foreach event in modificationEvents do
4:         if event.type = CREATEDARTIFACT or event.type = MODIFIEDARTIFACT then
5:             artifact := artifact considered by event
6:             if artifact type is considered by module then
7:                 scope.add(artifact)
8:     //Add graph pattern matched by dependency modules to scope (PAC case).
9:     foreach annotation created by dependency modules during current maintenance cycle do
10:        scope.add(annotation)
11:    //Add artifacts and annotations that were part of a graph pattern matched by dependency modules
12:    //to scope (NAC case).
13:    foreach annotation deleted by dependency modules during current maintenance cycle do
14:        foreach element referenced by annotation before deletion do
15:            scope.add(element) //element is artifact or annotation
16:    return scope
```

Algorithm 17 shows the algorithm to determine the scope consisting of artifacts and annotations that are required to incrementally compute new matches due to changes of base graphs in black box mode. For each element in the start scope (cf. algorithm 16) is checked whether it is an artifact or annotation (l. 4 and l. 21).

If the element is an artifact (l. 4) it is checked whether other artifacts are directly reachable via references owned by the artifact (l. 7). Note that we assume that all references are traversable bidirectional. If such artifacts exist and are not already contained by the scope, it is checked whether the artifact has an artifact (sub) type as specified by the artifact connectors of the module. If yes, the artifact is added to the scope and the scope is considered as changed.

Moreover, if the element is an artifact (l. 4) it is checked which annotations are attached to this artifact (l. 15). If an attached annotation has an annotation type as specified by annotation input connectors of the module and is not already contained by the scope, the annotation is added to scope and the scope is considered as changed.

If the element is an annotation (l. 21), all roles owned by this annotation are checked whether they reference an artifact or annotation that has an artifact (sub) type or annotation (sub) type as specified by artifact or annotation input connectors of the module (l. 24). If yes and these artifacts or annotations are not already contained by the scope, they are added to the scope and the scope is considered as changed.

Algorithm 18 shows the algorithm to determine the scope consisting of artifacts and annotations to incrementally computed new matches due to changes of base graphs in white box mode. The algorithm is similar to the algorithm described for black box mode (cf. algorithm 17). In contrast to black box mode, the white box mode considers the graph pattern specified within modules. By doing so, the algorithm can narrow the scope, because it only adds artifacts and annotations to the scope when they are reachable via references or roles that are part of the graph pattern. The differences in contrast to black box mode are highlighted grey (l. 9, 18, 28, and 34).

Algorithm 19 describes how suspicious annotations are determined when annotations were created by dependency modules. The algorithm is very similar to the algorithm for computing the scope in black box mode (cf. algorithm 17). Artifacts and annotations referenced by annotations that have been created by dependency modules are considered as start scope. The major difference is that annotations, which have an equal (sub) type as specified by the annotation *output* connector of the module (l. 24 and l. 41), are considered as suspicious annotations, because these annotations are reachable from the artifacts and annotations contained by the start scope and the matches marked by these annotations may not exist anymore since annotations created by dependency modules may dissatisfy a NAC. Note that a similar algorithm can be employed when white box semantic of modules is used. For that purpose, the algorithm has to check whether artifacts and annotations are reachable via references or roles that are part of the graph pattern specified within the module.

Algorithm 17.: Determine scope in black box mode

```
procedure DeterminеBlackBoxScope(scope)
    scopeChanged := false
    foreach element in scope do
        if element is artifact then
            artifact := element
            //Check for neighbour artifacts with relevant type.
            foreach reference owned by artifact do
                targets := target objects of reference
                foreach target in targets do
                    //Relevant (super) types are specified by artifact input connectors.
                    if target is not contained by scope and artifact type of target is relevant for module then
                        scope.add(target)
                        scopeChanged := true
            //Check which annotations with relevant annotation type are attached to artifacts.
            foreach role in which artifact acts do
                annotation := annotation that owns role
                //Relevant (super) types are specified by annotation input connectors.
                if annotation is not in scope and type of annotation is relevant for module then
                    scope.add(annotation)
                    scopeChanged := true
        else if element is annotation then
            annotation := element
            //Check if annotation references elements with relevant artifact type or annotation type.
            foreach role owned by annotation do
                target := element referenced by role
                //Relevant (super) types are specified by artifact and annotation input connectors.
                if target is not in scope and type of target is relevant for module then
                    scope.add(target)
                    scopeChanged := true
            foreach role in which annotation acts do
                dependentAnnotation := annotation that owns role
                //Relevant (super) types are specified annotation input connectors.
                if dependentAnnotation is not in scope and annotation type is relevant for module then
                    scope.add(dependentAnnotation)
                    scopeChanged := true
    //Perform recursive call, if additional elements were added to scope.
    if scopeChanged then
        DetermineBlackBoxScope(scope)
    return scope
```

Algorithm 18.: Determine scope in white box mode

```
procedure DETERMINEWHITEBOXSCOPE(module,scope)
    scopeChanged := false
    foreach element in scope do
        if element is artifact then
            artifact := element
            //Check for neighbour artifacts with relevant type.
            //Relevant (super) types are specified by artifact input connectors.
            foreach reference of element do
                if reference is part of the graph pattern specified by module then
                    targets := target objects of reference
                    foreach target in targets do
                        if target is not contained by scope and type of target is relevant for module then
                            scope.add(target)
                            scopeChanged := true
            //Check which annotations with relevant type are attached to artifacts.
            //Relevant (super) types are specified by annotation input connectors.
            foreach role in which element acts do
                if role is part of the graph pattern specified by module then
                    annotation := annotation that owns role
                    if annotation is not in scope and type of annotation is relevant for module then
                        scope.add(annotation)
                        scopeChanged := true
        else if element is annotation then
            annotation := element
            //Check if annotation references elements with relevant artifact type or annotation type.
            //Relevant (super) types are specified by artifact and annotation input connectors.
            foreach role owned by annotation do
                if role is part of the graph pattern specified by module then
                    target := element referenced by role
                    if target is not in scope and type of target is relevant for module then
                        scope.add(target)
                        scopeChanged := true
            foreach role in which annotation acts do
                if role is part of the graph pattern specified by module then
                    dependentAnnotation := annotation that owns role
                    //Relevant (super) types are specified annotation input connectors.
                    if dependentAnnotation is not in scope and type of annotation is relevant for module then
                        scope.add(dependentAnnotation)
                        scopeChanged := true
    //Perform recursive call, if additional elements were added to scope.
    if scopeChanged then
        DETERMINEWHITEBOXSCOPE(scope)
    return scope
```

Algorithm 19.: Determine scope in black box mode when annotations were created

```
 1: procedure SuspiciousAnnotationsDueToCreatedAnnotations(module,createdAnnotations)
 2:     suspiciousAnnotations := empty
 3:     foreach annnotaton in createdAnnotations do
 4:         scope := artifacts and annotations referenced by annotation
 5:         scopeChanged := true
 6:         while scopeChanged do
 7:             scopeChanged := false
 8:             foreach element in scope do
 9:                 if element is artifact then
10:                     artifact := element
11:                     //Check for neighbour artifacts with relevant type.
12:                     foreach neighbour of artifact do
13:                         //Relevant (super) types are specified by artifact input connectors.
14:                         if scope does not contain neigbour and neighbour has relevant type for module then
15:                             scope.add(neighbour)
16:                             scopeChanged := true
17:                     //Check which annotations with relevant type are attached to artifacts.
18:                     foreach role in which artifact acts do
19:                         annotation := annotation that owns role
20:                         //Relevant (super) types are specified by annotation input connectors.
21:                         if scope does not contain annotation and type of annotation is relevant for module then
22:                             scope.add(annotation)
23:                             scopeChanged := true
24:                         if annotation has an equal (sub) type as specified by module *output* connector then
25:                             suspiciousAnnotations.add(annotation)
26:                 else if element is annotation then
27:                     annotation := element
28:                     //Check if annotation references elements with relevant artifact type or annotation type.
29:                     foreach role owned by annotation do
30:                         target := element referenced by role
31:                         //Relevant (super) types are specified by artifact and annotation input connectors.
32:                         if scope does not contain target and type of target is relevant for module then
33:                             scope.add(target)
34:                             scopeChanged := true
35:                     foreach role in which annotation acts do
36:                         dependentAnnotation := annotation that owns role
37:                         //Relevant (super) types are specified annotation input connectors.
38:                         if dependentAnnotation is not in scope and type of annotation is relevant for module
    then
39:                             scope.add(dependentAnnotation)
40:                             scopeChanged := true
41:                     if annotation has an equal (sub) type as specified by module *output* connector then
42:                         suspiciousAnnotations.add(annotation)
43:     return suspiciousAnnotations
```

C. View Modules for Design Pattern Recovery

The following figures show the employed view modules. Note that some view modules approximate design patterns, especially when non-structural design patterns are recovered. We do not aim at improving the precision and recall of recovered software design patterns.

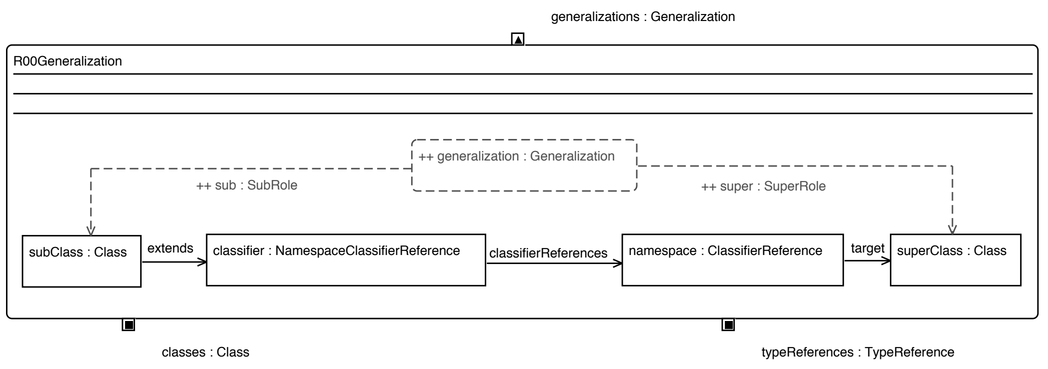

generalizations : Generalization
R00Generalization
++ generalization : Generalization
++ sub : SubRole
++ super : SuperRole
subClass : Class
extends
classifier : NamespaceClassifierReference
classifierReferences
namespace : ClassifierReference
target
superClass : Class
classes : Class
typeReferences : TypeReference

Figure C.1.: `Generalization` module – A generalization between two classes exists, if the sub class points via a namespace classifier and classifier reference to its super class.

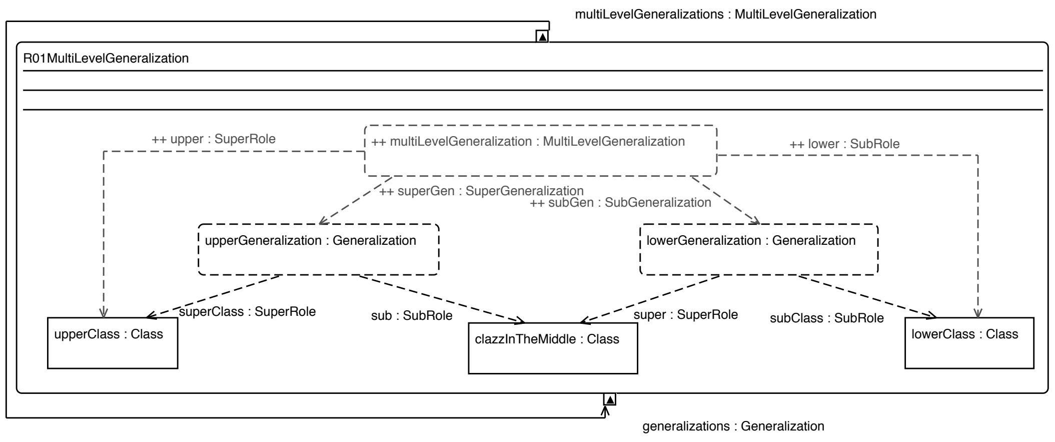
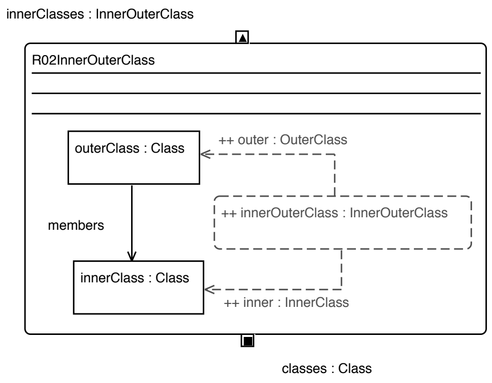
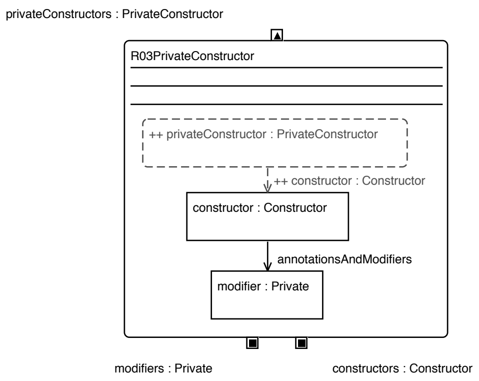

multiLevelGeneralizations : MultiLevelGeneralization
R01MultiLevelGeneralization
++ upper : SuperRole
++ multiLevelGeneralization : MultiLevelGeneralization
++ lower : SubRole
++ superGen : SuperGeneralization
++ subGen : SubGeneralization
upperGeneralization : Generalization
lowerGeneralization : Generalization
superClass : SuperRole
sub : SubRole
super : SuperRole
subClass : SubRole
upperClass : Class
clazzInTheMiddle : Class
lowerClass : Class
generalizations : Generalization

Figure C.2.: `MultiLevelGeneralization` module – A multi-level generalization between two classes exists, if the super class of a generalization is the sub class of another generalization.

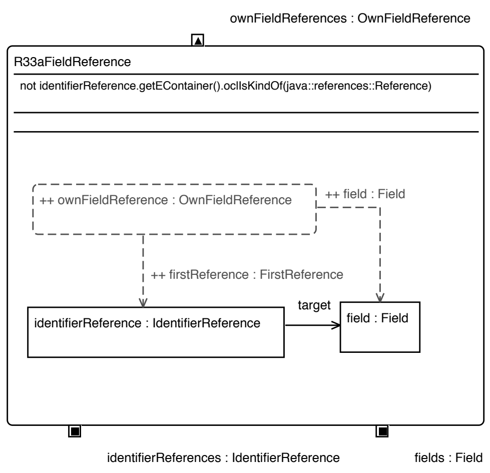



privateConstructors : PrivateConstructor
R03PrivateConstructor
++ privateConstructor : PrivateConstructor
++ constructor : Constructor
constructor : Constructor
annotationsAndModifiers
modifier : Private
modifiers : Private
constructors : Constructor

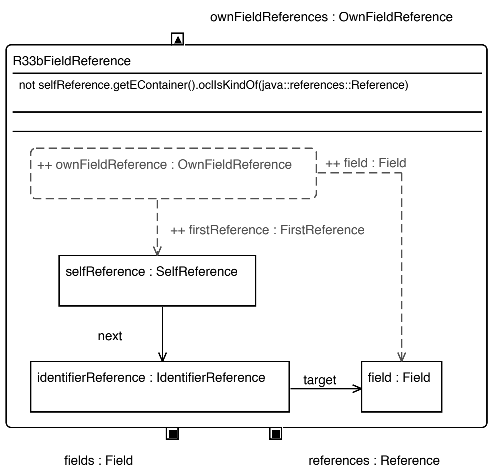
(a) `InterOuterClass` module – A class is an inner class, if it is a member of another class.

(b) `PrivateConstructor` module – A constructor is a private constructor, if the constructor consists of a private modifier.

Figure C.3.: `InterOuterClass` module and `PrivateConstructor` module


ownFieldReferences : OwnFieldReference
R33aFieldReference
not identifierReference.getEContainer().oclIsKindOf(java::references::Reference)
++ ownFieldReference : OwnFieldReference
++ field : Field
++ firstReference : FirstReference
identifierReference : IdentifierReference
target
field : Field
identifierReferences : IdentifierReference
fields : Field

(a) `FieldReference` module (without self reference) – An own field reference is a reference to a field in the same class, e.g. without using the keyword "this".


ownFieldReferences : OwnFieldReference
R33bFieldReference
not selfReference.getEContainer().oclIsKindOf(java::references::Reference)
++ ownFieldReference : OwnFieldReference
++ field : Field
++ firstReference : FirstReference
selfReference : SelfReference
next
identifierReference : IdentifierReference
target
field : Field
fields : Field
references : Reference

(b) `FieldReference` module (with self reference) – An field reference is a reference to a field in the same class, e.g. with using the keyword "this".

Figure C.4.: `FieldReference` modules with and without self reference

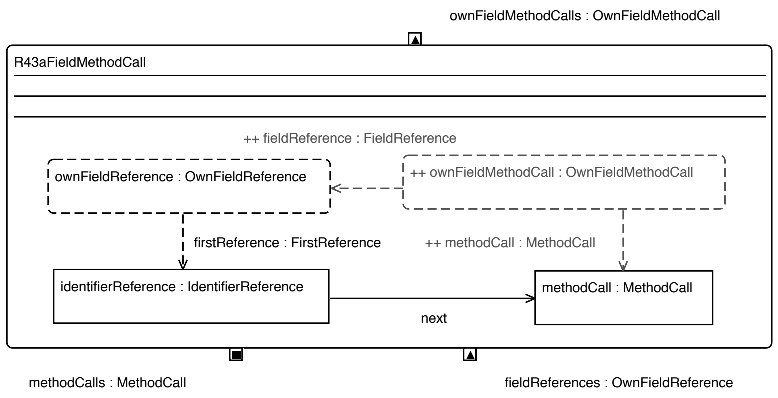

ownFieldMethodCalls : OwnFieldMethodCall
R43aFieldMethodCall
++ fieldReference : FieldReference
ownFieldReference : OwnFieldReference
++ ownFieldMethodCall : OwnFieldMethodCall
firstReference : FirstReference
++ methodCall : MethodCall
identifierReference : IdentifierReference
methodCall : MethodCall
next
methodCalls : MethodCall
fieldReferences : OwnFieldReference

Figure C.5.: `FieldMethodCall` module (without self reference) – A field method call exists, if a method is called on a field of the same class, e.g. without "this" keyword.

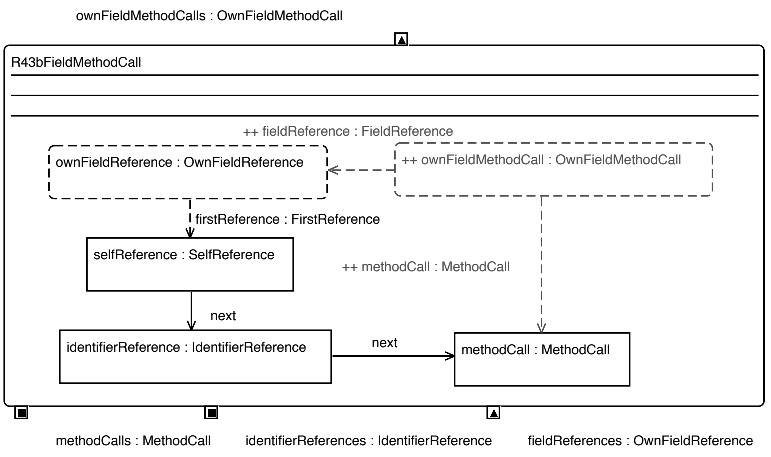

ownFieldMethodCalls : OwnFieldMethodCall
R43bFieldMethodCall
++ fieldReference : FieldReference
ownFieldReference : OwnFieldReference
++ ownFieldMethodCall : OwnFieldMethodCall
firstReference : FirstReference
selfReference : SelfReference
++ methodCall : MethodCall
next
identifierReference : IdentifierReference
next
methodCall : MethodCall
methodCalls : MethodCall
identifierReferences : IdentifierReference
fieldReferences : OwnFieldReference

Figure C.6.: `FieldMethodCall` module (with self reference) – A field method call exists, if a method is called on a field of the same class, e.g. with "this" keyword.

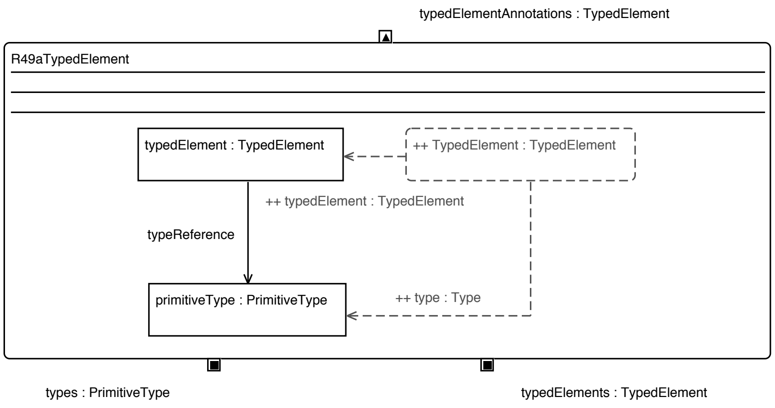

typedElementAnnotations : TypedElement
R49aTypedElement
typedElement : TypedElement
++ TypedElement : TypedElement
++ typedElement : TypedElement
typeReference
primitiveType : PrimitiveType
++ type : Type
types : PrimitiveType
typedElements : TypedElement

Figure C.7.: `TypedElement` module (with primitive type) – A typed element can consists of a primitive type.

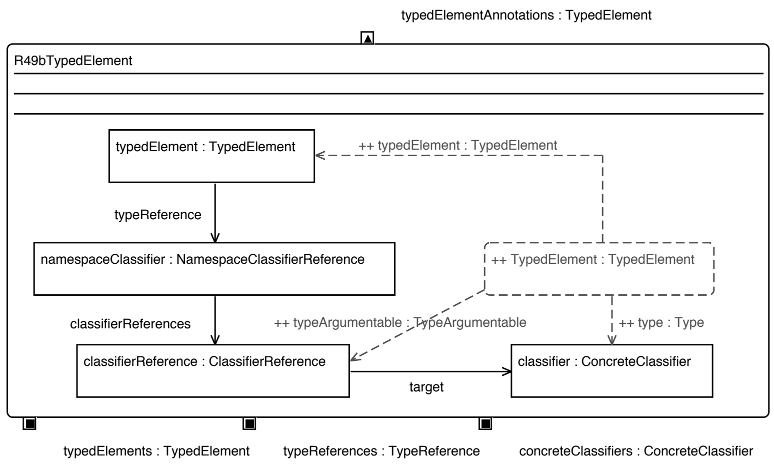

typedElementAnnotations : TypedElement
R49bTypedElement
typedElement : TypedElement
++ typedElement : TypedElement
typeReference
namespaceClassifier : NamespaceClassifierReference
++ TypedElement : TypedElement
classifierReferences
++ typeArgumentable : TypeArgumentable
++ type : Type
classifierReference : ClassifierReference
classifier : ConcreteClassifier
target
typedElements : TypedElement
typeReferences : TypeReference
concreteClassifiers : ConcreteClassifier

Figure C.8.: `TypedElement` module (with complex type) – A typed element can consists of a concrete classifiers.

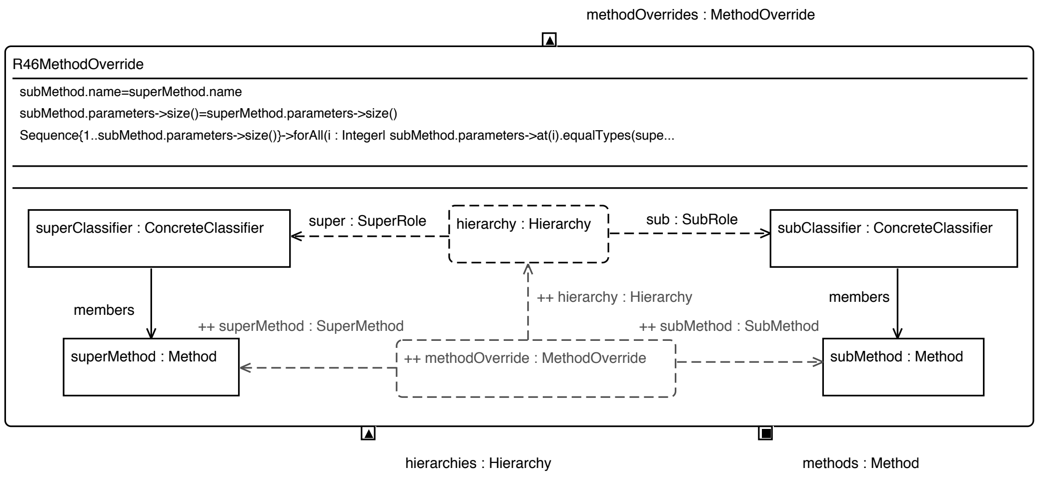
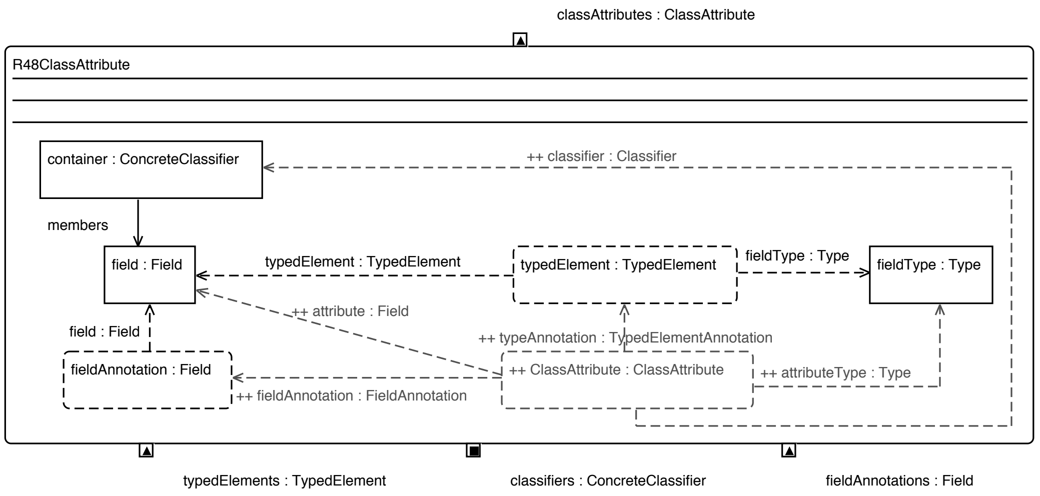

methodOverrides : MethodOverride
R46MethodOverride
subMethod.name=superMethod.name
subMethod.parameters->size()=superMethod.parameters->size()
Sequence{1..subMethod.parameters->size()}->forAll(i : Integerl subMethod.parameters->at(i).equalTypes(supe...
superClassifier : ConcreteClassifier
super : SuperRole
hierarchy : Hierarchy
sub : SubRole
subClassifier : ConcreteClassifier
members
++ hierarchy : Hierarchy
members
++ superMethod : SuperMethod
++ subMethod : SubMethod
superMethod : Method
++ methodOverride : MethodOverride
subMethod : Method
hierarchies : Hierarchy
methods : Method

Figure C.9.: `MethodOverride` module – A method in a sub class overrides the method of its super class, if a generalization between both classes exists, the methods have the same name, the same number of parameters and the same parameter types.



classAttributes : ClassAttribute
R48ClassAttribute
container : ConcreteClassifier
++ classifier : Classifier
members
field : Field
typedElement : TypedElement
typedElement : TypedElement
fieldType : Type
fieldType : Type
++ attribute : Field
field : Field
++ typeAnnotation : TypedElementAnnotation
fieldAnnotation : Field
++ ClassAttribute : ClassAttribute
++ attributeType : Type
++ fieldAnnotation : FieldAnnotation
typedElements : TypedElement
classifiers : ConcreteClassifier
fieldAnnotations : Field

Figure C.10.: `ClassAttribute` module – A field is a class attribute, if the field is contained by a concrete classifier and has a primitive type or complex type.

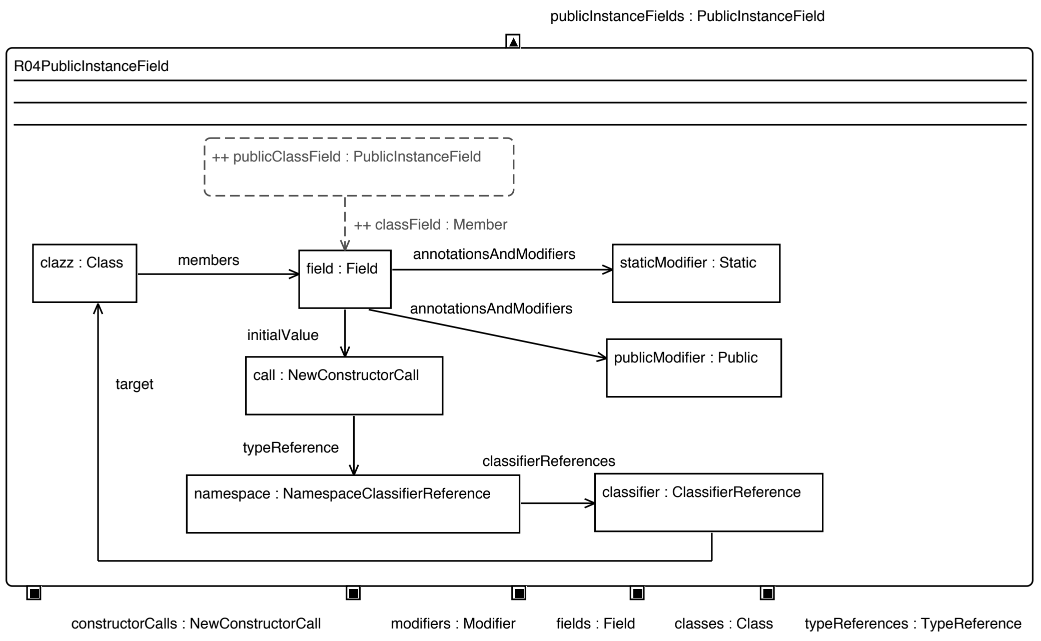


Figure C.11.: `PublicInstanceField` module – A field is a public instance field, if the field has a static and public modifier and the initial value of the field is an instance of the class that contains the field as a member. The initial value of the field is obtained from the call of the constructor of the class that owns the field, e.g. public instance field or public instance method.

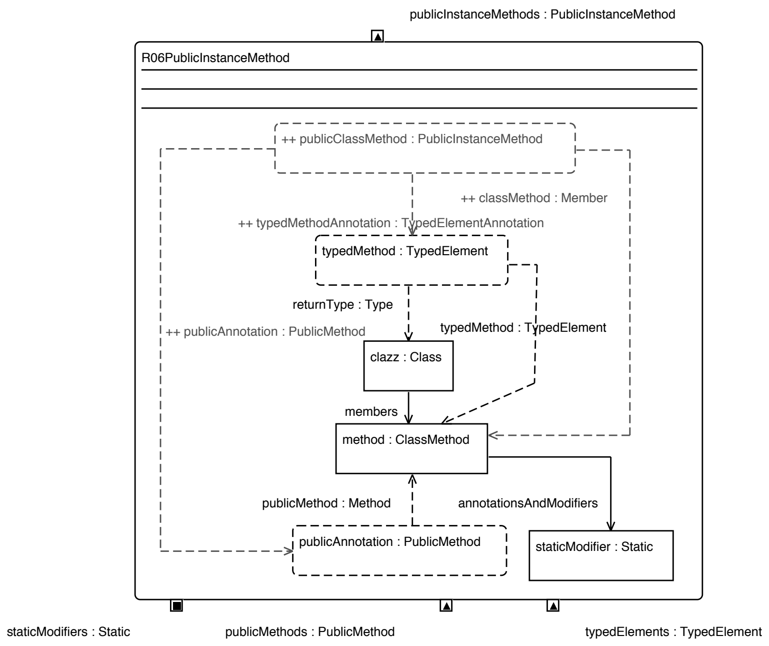

publicInstanceMethods : PublicInstanceMethod
R06PublicInstanceMethod
++ publicClassMethod : PublicInstanceMethod
++ classMethod : Member
++ typedMethodAnnotation : TypedElementAnnotation
typedMethod : TypedElement
returnType : Type
++ publicAnnotation : PublicMethod
typedMethod : TypedElement
clazz : Class
members
method : ClassMethod
publicMethod : Method
annotationsAndModifiers
publicAnnotation : PublicMethod
staticModifier : Static
staticModifiers : Static
publicMethods : PublicMethod
typedElements : TypedElement

Figure C.12.: `PublicInstanceMethod` module – A method is a public instance method, if the method consists of a static and public modifier and the method returns an instance of the class that owns the method.

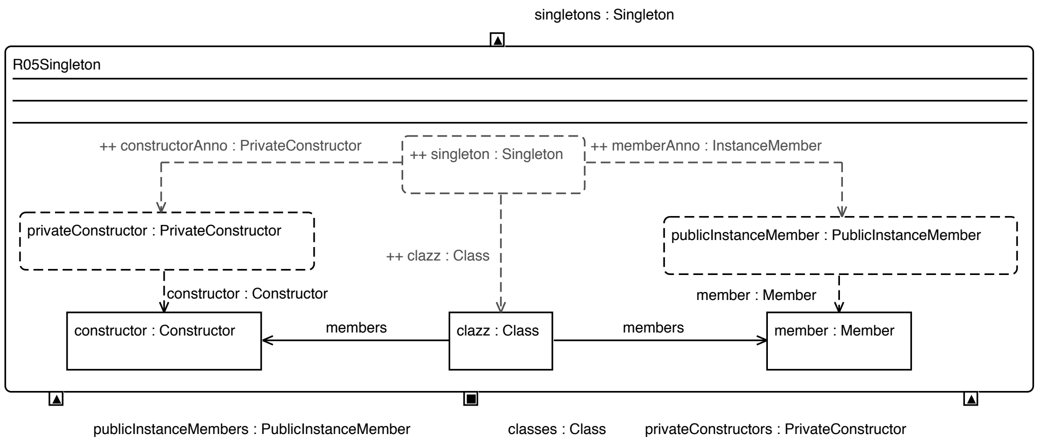

singletons : Singleton
R05Singleton
++ constructorAnno : PrivateConstructor
++ singleton : Singleton
++ memberAnno : InstanceMember
privateConstructor : PrivateConstructor
++ clazz : Class
publicInstanceMember : PublicInstanceMember
constructor : Constructor
member : Member
constructor : Constructor
members
clazz : Class
members
member : Member
publicInstanceMembers : PublicInstanceMember
classes : Class
privateConstructors : PrivateConstructor

Figure C.13.: `Singleton` module – A class is a Singleton design pattern, if the class consists of a private constructor and a member that stores or returns an instance of the class.

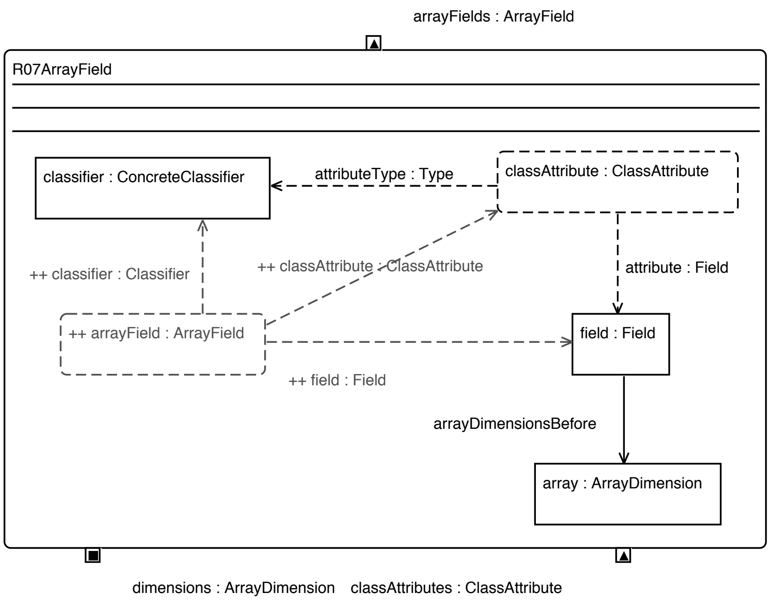

arrayFields : ArrayField
R07ArrayField
classifier : ConcreteClassifier
attributeType : Type
classAttribute : ClassAttribute
++ classifier : Classifier
++ classAttribute : ClassAttribute
attribute : Field
++ arrayField : ArrayField
field : Field
++ field : Field
arrayDimensionsBefore
array : ArrayDimension
dimensions : ArrayDimension
classAttributes : ClassAttribute

Figure C.14.: `ArrayField` module – A field is an array field, if the field is an attribute of a class and consists of an array dimension.

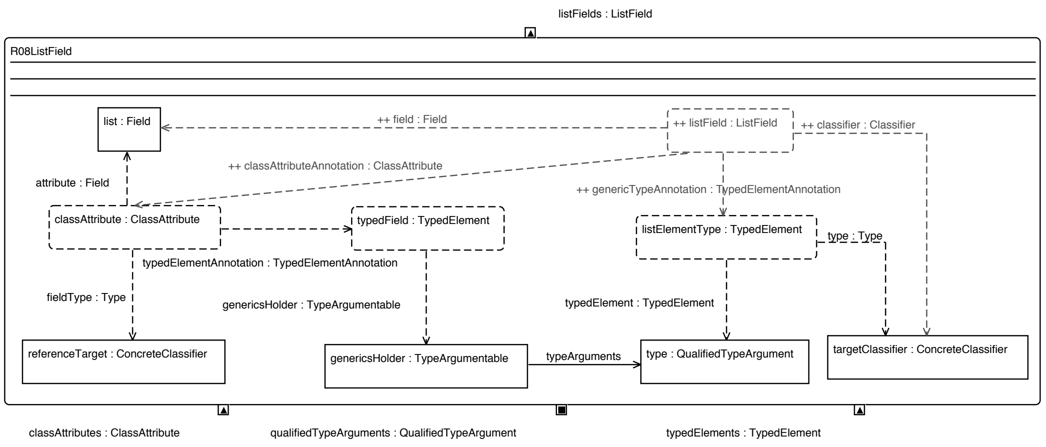

listFields : ListField
R08ListField
list : Field
++ field : Field
++ listField : ListField
++ classifier : Classifier
++ classAttributeAnnotation : ClassAttribute
attribute : Field
++ genericTypeAnnotation : TypedElementAnnotation
classAttribute : ClassAttribute
typedField : TypedElement
listElementType : TypedElement
type : Type
typedElementAnnotation : TypedElementAnnotation
fieldType : Type
genericsHolder : TypeArgumentable
typedElement : TypedElement
referenceTarget : ConcreteClassifier
genericsHolder : TypeArgumentable
typeArguments
type : QualifiedTypeArgument
targetClassifier : ConcreteClassifier
classAttributes : ClassAttribute
qualifiedTypeArguments : QualifiedTypeArgument
typedElements : TypedElement

Figure C.15.: `ListField` module – A field is a list field, if the field is a class attribute and defines the type of the elements contained by the list.

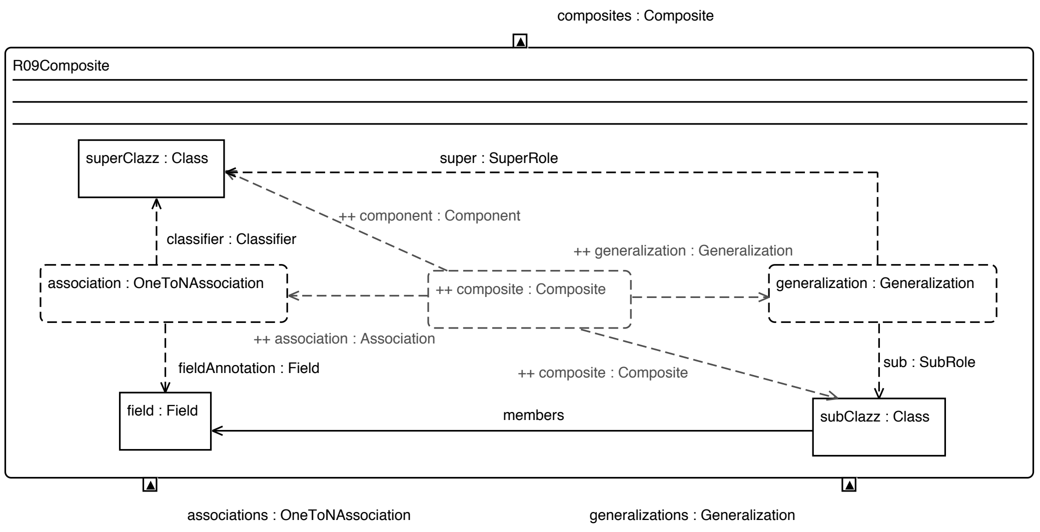


Figure C.16.: `Composite` module – Two classes constitute a Composite design pattern, if a generalization (incl. multi-level generalization) between both classes exists and the sub class owns a to-many reference that has the super class as target classifier.

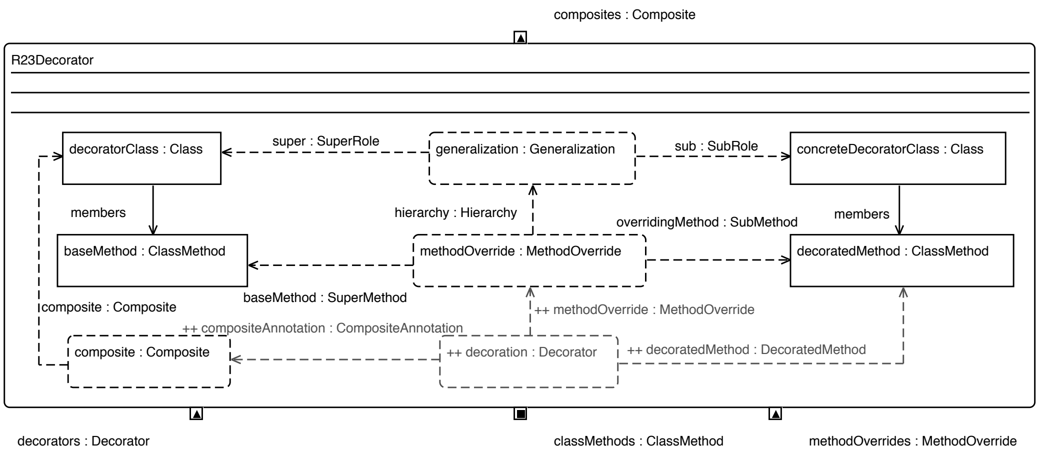

composites : Composite
R23Decorator
decoratorClass : Class
super : SuperRole
generalization : Generalization
sub : SubRole
concreteDecoratorClass : Class
members
hierarchy : Hierarchy
overridingMethod : SubMethod
members
baseMethod : ClassMethod
methodOverride : MethodOverride
decoratedMethod : ClassMethod
baseMethod : SuperMethod
composite : Composite
++ methodOverride : MethodOverride
++ compositeAnnotation : CompositeAnnotation
composite : Composite
++ decoration : Decorator
++ decoratedMethod : DecoratedMethod
decorators : Decorator
classMethods : ClassMethod
methodOverrides : MethodOverride

Figure C.17.: `Decorator` module – A Decorator design pattern exists, if a Composite design pattern exists and the sub class in the Composite design pattern overrides a method of the super class with additional functionality.

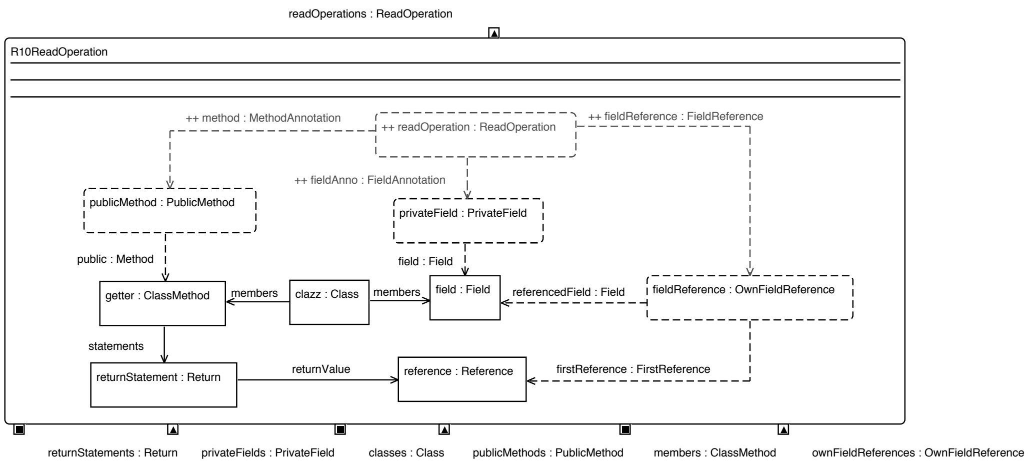

readOperations : ReadOperation
R10ReadOperation
++ method : MethodAnnotation
++ readOperation : ReadOperation
++ fieldReference : FieldReference
++ fieldAnno : FieldAnnotation
publicMethod : PublicMethod
privateField : PrivateField
public : Method
field : Field
getter : ClassMethod
members
clazz : Class
members
field : Field
referencedField : Field
fieldReference : OwnFieldReference
statements
returnStatement : Return
returnValue
reference : Reference
firstReference : FirstReference
returnStatements : Return
privateFields : PrivateField
classes : Class
publicMethods : PublicMethod
members : ClassMethod
ownFieldReferences : OwnFieldReference

Figure C.18.: `ReadOperation` module – A read operation (getter method) of a class is a public method that returns the value of a private field owned by this class.

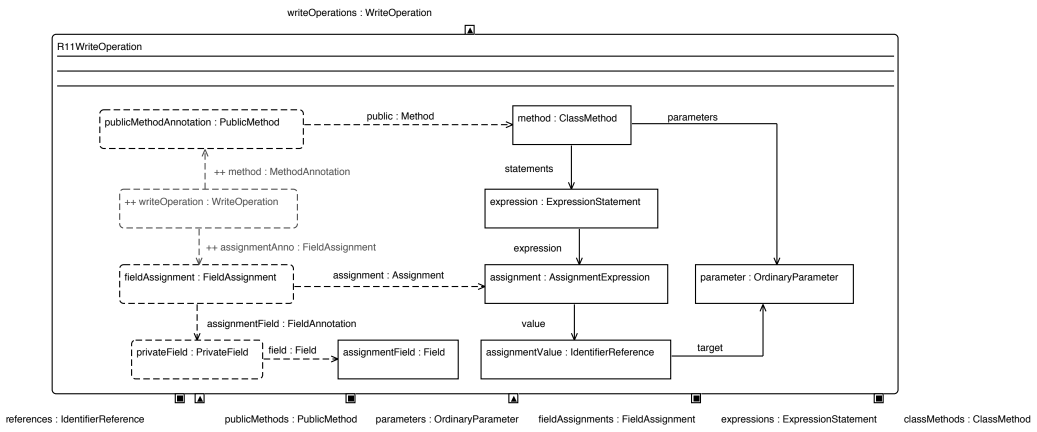

writeOperations : WriteOperation
R11WriteOperation
publicMethodAnnotation : PublicMethod
public : Method
method : ClassMethod
parameters
++ method : MethodAnnotation
statements
++ writeOperation : WriteOperation
expression : ExpressionStatement
++ assignmentAnno : FieldAssignment
expression
fieldAssignment : FieldAssignment
assignment : Assignment
assignment : AssignmentExpression
parameter : OrdinaryParameter
assignmentField : FieldAnnotation
value
privateField : PrivateField
field : Field
assignmentField : Field
assignmentValue : IdentifierReference
target
references : IdentifierReference
publicMethods : PublicMethod
parameters : OrdinaryParameter
fieldAssignments : FieldAssignment
expressions : ExpressionStatement
classMethods : ClassMethod

Figure C.19.: `WriteOperation` module – A write operation (setter method) of a class is a public method with at least one parameter. The value of the parameter must be assigned to a private field owned by this class.

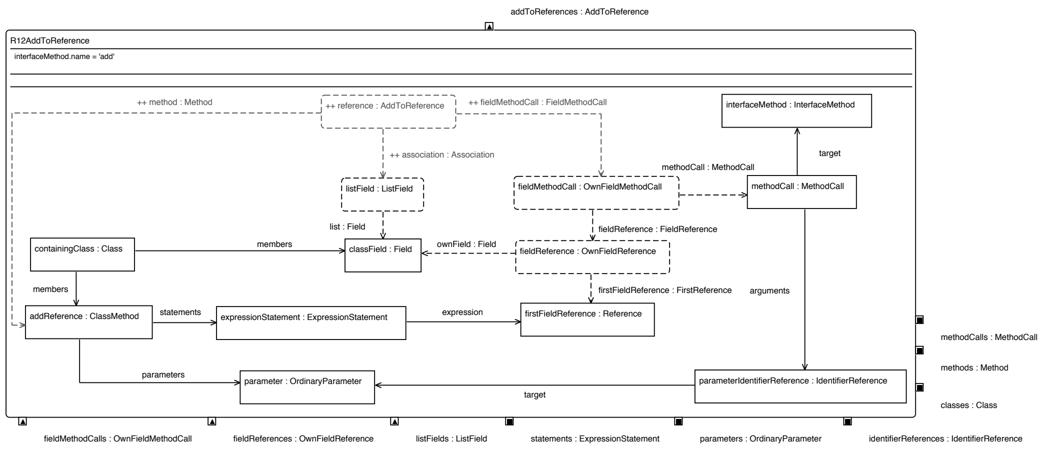

addToReferences : AddToReference
R12AddToReference
interfaceMethod.name = 'add'
++ method : Method
++ reference : AddToReference
++ fieldMethodCall : FieldMethodCall
interfaceMethod : InterfaceMethod
++ association : Association
target
methodCall : MethodCall
listField : ListField
fieldMethodCall : OwnFieldMethodCall
methodCall : MethodCall
list : Field
fieldReference : FieldReference
containingClass : Class
members
classField : Field
ownField : Field
fieldReference : OwnFieldReference
members
firstFieldReference : FirstReference
arguments
addReference : ClassMethod
statements
expressionStatement : ExpressionStatement
expression
firstFieldReference : Reference
methodCalls : MethodCall
methods : Method
parameters
parameter : OrdinaryParameter
target
parameterIdentifierReference : IdentifierReference
classes : Class
fieldMethodCalls : OwnFieldMethodCall
fieldReferences : OwnFieldReference
listFields : ListField
statements : ExpressionStatement
parameters : OrdinaryParameter
identifierReferences : IdentifierReference

Figure C.20.: `AddToReference` module – A public method adds the value of a method parameter to a list attribute of a class, if the attribute is a list, a method called "add" is called on this list, and the argument passed to this method is the argument passed via the containing method.

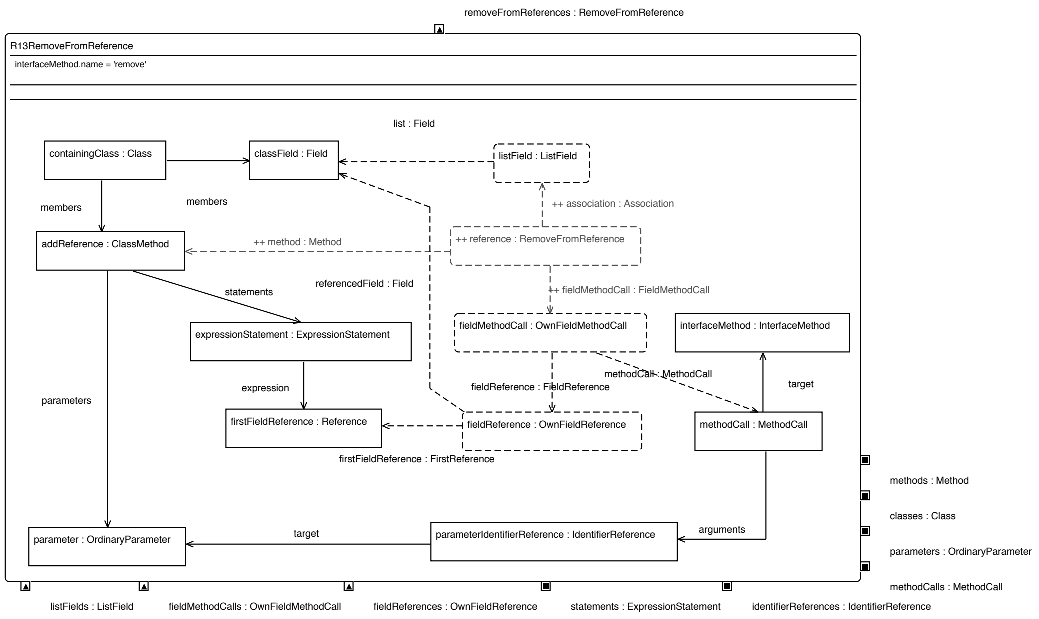

removeFromReferences : RemoveFromReference
R13RemoveFromReference
interfaceMethod.name = 'remove'
list : Field
containingClass : Class
classField : Field
listField : ListField
members
members
++ association : Association
addReference : ClassMethod
++ method : Method
++ reference : RemoveFromReference
referencedField : Field
statements
++ fieldMethodCall : FieldMethodCall
fieldMethodCall : OwnFieldMethodCall
interfaceMethod : InterfaceMethod
expressionStatement : ExpressionStatement
methodCall : MethodCall
parameters
expression
fieldReference : FieldReference
target
firstFieldReference : Reference
fieldReference : OwnFieldReference
methodCall : MethodCall
firstFieldReference : FirstReference
methods : Method
classes : Class
arguments
parameter : OrdinaryParameter
target
parameterIdentifierReference : IdentifierReference
parameters : OrdinaryParameter
methodCalls : MethodCall
listFields : ListField
fieldMethodCalls : OwnFieldMethodCall
fieldReferences : OwnFieldReference
statements : ExpressionStatement
identifierReferences : IdentifierReference

Figure C.21.: `RemoveFromReference` module – A public method removes the value of a method parameter from a list attribute of a class, if the attribute is a list, a method called "remove" is called on this list, and the argument passed to this method is the argument passed via the containing method.

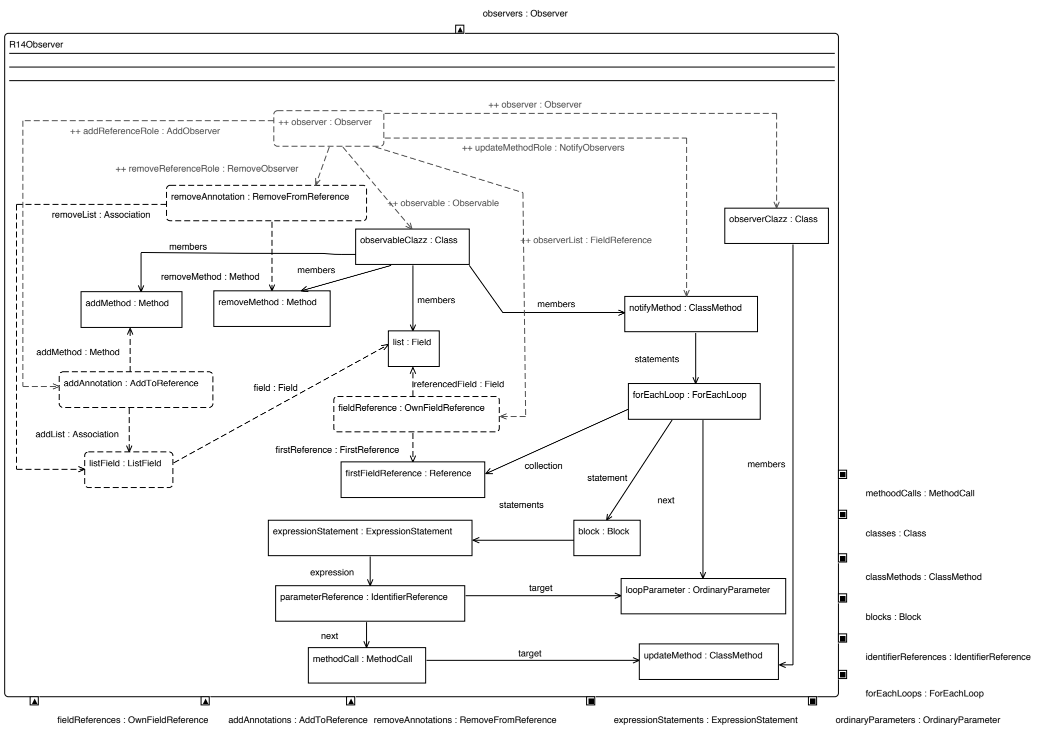

observers : Observer
R14Observer
++ observer : Observer
++ addReferenceRole : AddObserver
++ observer : Observer
++ updateMethodRole : NotifyObservers
++ removeReferenceRole : RemoveObserver
removeAnnotation : RemoveFromReference
++ observable : Observable
removeList : Association
observerClazz : Class
observableClazz : Class
members
++ observerList : FieldReference
removeMethod : Method
members
addMethod : Method
removeMethod : Method
members
members
notifyMethod : ClassMethod
list : Field
statements
addMethod : Method
referencedField : Field
addAnnotation : AddToReference
field : Field
forEachLoop : ForEachLoop
fieldReference : OwnFieldReference
addList : Association
firstReference : FirstReference
listField : ListField
firstFieldReference : Reference
collection
statement
members
next
statements
expressionStatement : ExpressionStatement
block : Block
expression
target
parameterReference : IdentifierReference
loopParameter : OrdinaryParameter
next
target
methodCall : MethodCall
updateMethod : ClassMethod
methoodCalls : MethodCall
classes : Class
classMethods : ClassMethod
blocks : Block
identifierReferences : IdentifierReference
forEachLoops : ForEachLoop
fieldReferences : OwnFieldReference
addAnnotations : AddToReference
removeAnnotations : RemoveFromReference
expressionStatements : ExpressionStatement
ordinaryParameters : OrdinaryParameter

Figure C.22.: `Observer` module – An Observer design pattern exists, if two classes exists that represent the observers and observables. The observable class must implement a method that iterates over a list field that contains all observers of the observable and calls a method of each observer to notify the observer about changes of observables. Furthermore, the observeable class must provide two methods that enable to add and remove observers from the list of observers.

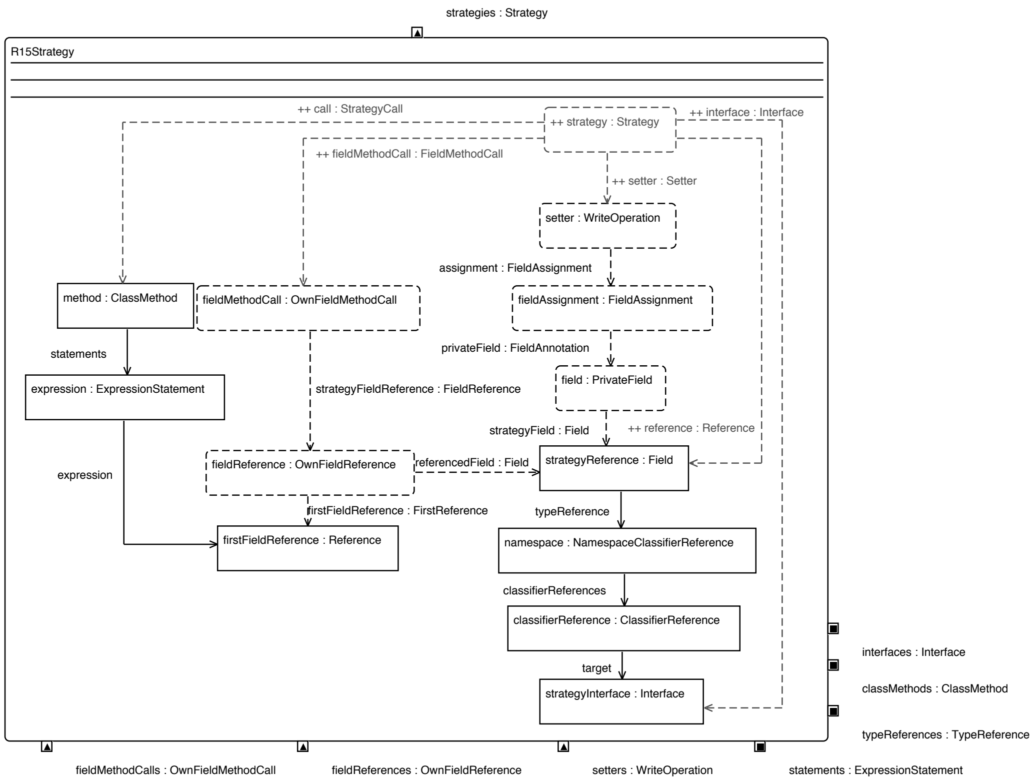
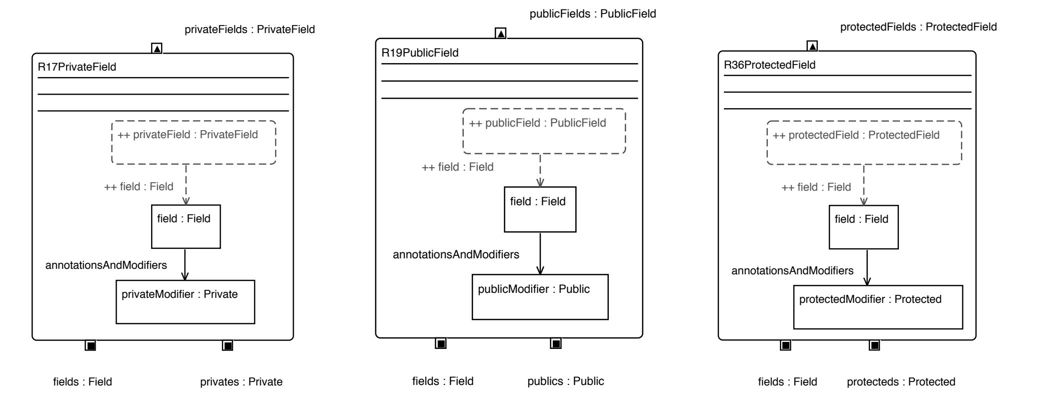

strategies : Strategy
R15Strategy
++ call : StrategyCall
++ strategy : Strategy
++ interface : Interface
++ fieldMethodCall : FieldMethodCall
++ setter : Setter
setter : WriteOperation
assignment : FieldAssignment
method : ClassMethod
fieldMethodCall : OwnFieldMethodCall
fieldAssignment : FieldAssignment
statements
privateField : FieldAnnotation
expression : ExpressionStatement
strategyFieldReference : FieldReference
field : PrivateField
strategyField : Field
++ reference : Reference
expression
fieldReference : OwnFieldReference
referencedField : Field
strategyReference : Field
firstFieldReference : FirstReference
typeReference
firstFieldReference : Reference
namespace : NamespaceClassifierReference
classifierReferences
classifierReference : ClassifierReference
target
strategyInterface : Interface
interfaces : Interface
classMethods : ClassMethod
typeReferences : TypeReference
fieldMethodCalls : OwnFieldMethodCall
fieldReferences : OwnFieldReference
setters : WriteOperation
statements : ExpressionStatement

Figure C.23.: `Strategy` module – A Strategy design pattern exists, if a write operation (setter method) exists that enables to set the strategy that has to be used and a method exists that executes the set strategy.



(a) `PrivateField` module – A field is a privat field, if it consists of a private modifier.

(b) `PublicField` module – A field is a public field, if it consists of a public modifier.

(c) `ProtectedField` module – A field is a protected field, if it consists of a protected modifier.

Figure C.24.: `PrivateField` module, `PublicField` module, and `ProtectedField` module

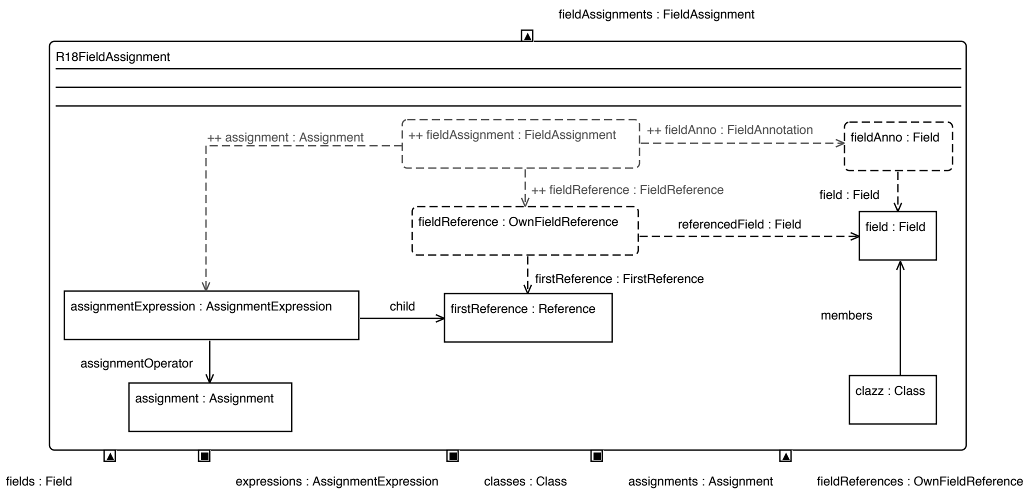

fieldAssignments : FieldAssignment
R18FieldAssignment
++ assignment : Assignment
++ fieldAssignment : FieldAssignment
++ fieldAnno : FieldAnnotation
fieldAnno : Field
++ fieldReference : FieldReference
field : Field
fieldReference : OwnFieldReference
referencedField : Field
field : Field
firstReference : FirstReference
assignmentExpression : AssignmentExpression
child
firstReference : Reference
members
assignmentOperator
assignment : Assignment
clazz : Class
fields : Field
expressions : AssignmentExpression
classes : Class
assignments : Assignment
fieldReferences : OwnFieldReference

Figure C.25.: `FieldAssignment` module – A field assignment is present, if a field of a class is referenced in an expression that consists of an assignment.

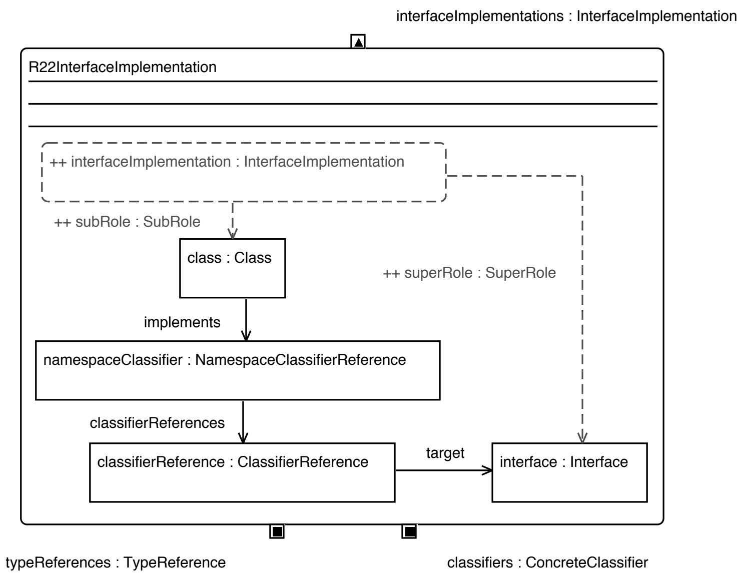

interfaceImplementations : InterfaceImplementation
R22InterfaceImplementation
++ interfaceImplementation : InterfaceImplementation
++ subRole : SubRole
class : Class
++ superRole : SuperRole
implements
namespaceClassifier : NamespaceClassifierReference
classifierReferences
classifierReference : ClassifierReference
target
interface : Interface
typeReferences : TypeReference
classifiers : ConcreteClassifier

Figure C.26.: `InterfaceImplementation` module – An interface implementation between two classifiers exists, if the class that implements the interface points via a namespace classifier and classifier reference to its interface.

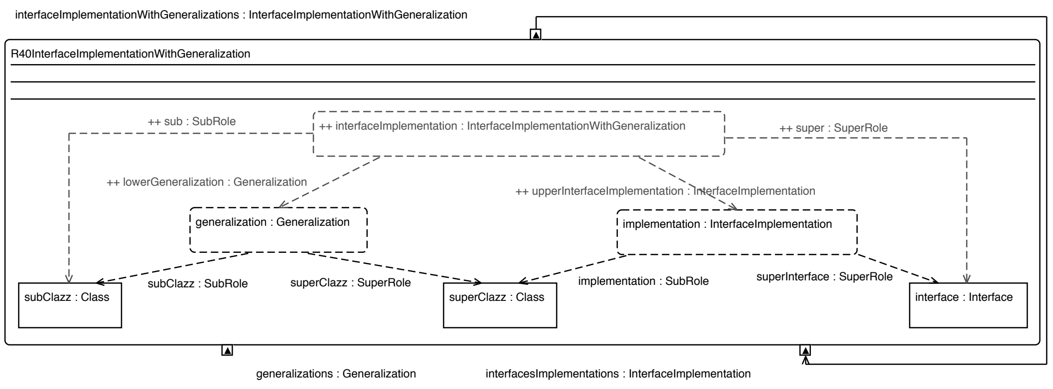

interfaceImplementationWithGeneralizations : InterfaceImplementationWithGeneralization
R40InterfaceImplementationWithGeneralization
++ sub : SubRole
++ interfaceImplementation : InterfaceImplementationWithGeneralization
++ super : SuperRole
++ lowerGeneralization : Generalization
++ upperInterfaceImplementation : InterfaceImplementation
generalization : Generalization
implementation : InterfaceImplementation
subClazz : SubRole
superClazz : SuperRole
implementation : SubRole
superInterface : SuperRole
subClazz : Class
superClazz : Class
interface : Interface
generalizations : Generalization
interfacesImplementations : InterfaceImplementation

Figure C.27.: `InterfaceImplementationWithGeneralization` module – A class implements an interface, if one super class of this class implements the interface.

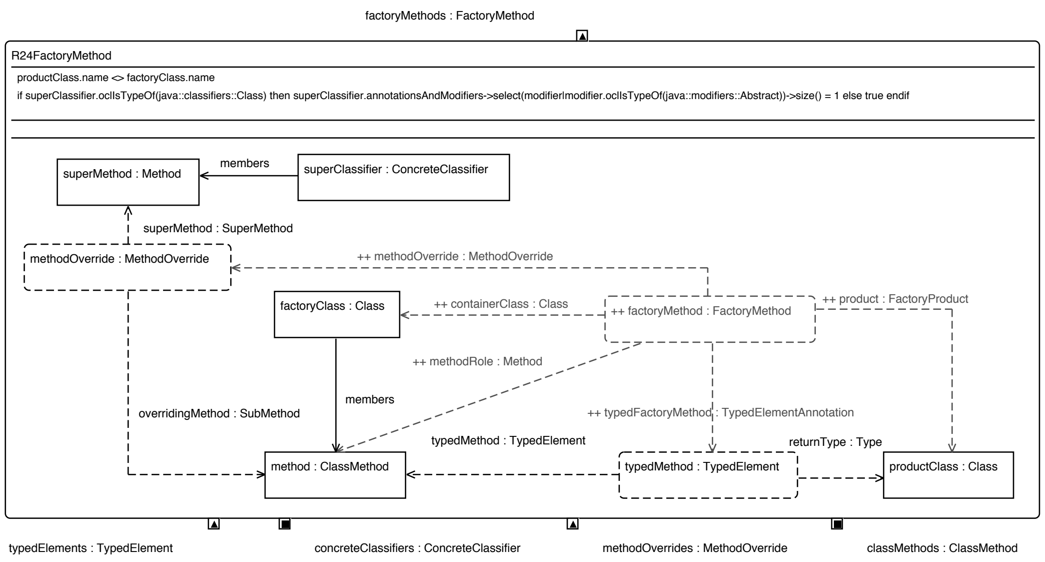

factoryMethods : FactoryMethod
R24FactoryMethod
productClass.name <> factoryClass.name
if superClassifier.oclIsTypeOf(java::classifiers::Class) then superClassifier.annotationsAndModifiers->select(modifier|modifier.oclIsTypeOf(java::modifiers::Abstract))->size() = 1 else true endif
superMethod : Method
members
superClassifier : ConcreteClassifier
superMethod : SuperMethod
methodOverride : MethodOverride
++ methodOverride : MethodOverride
factoryClass : Class
++ containerClass : Class
++ factoryMethod : FactoryMethod
++ product : FactoryProduct
++ methodRole : Method
members
overridingMethod : SubMethod
++ typedFactoryMethod : TypedElementAnnotation
typedMethod : TypedElement
returnType : Type
method : ClassMethod
typedMethod : TypedElement
productClass : Class
typedElements : TypedElement
concreteClassifiers : ConcreteClassifier
methodOverrides : MethodOverride
classMethods : ClassMethod

Figure C.28.: `FactoryMethod` module – A method is a factory method, if the class that contains the method is a sub class or implements an interface and returns as product an instance of a classifier that is different from the super class/interface. Furthermore, if the sub class extends a super class the super class must be abstract.

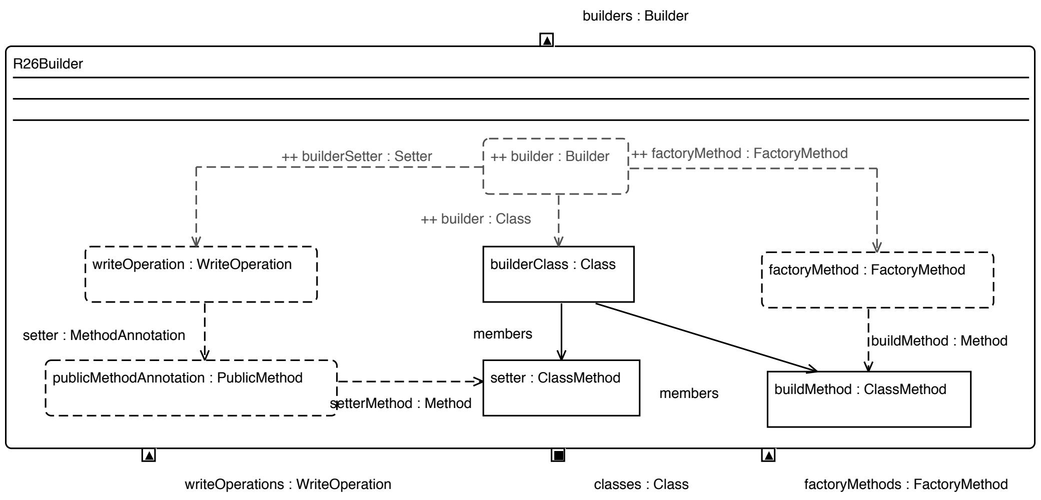

builders : Builder
R26Builder
++ builderSetter : Setter
++ builder : Builder
++ factoryMethod : FactoryMethod
++ builder : Class
writeOperation : WriteOperation
builderClass : Class
factoryMethod : FactoryMethod
setter : MethodAnnotation
members
buildMethod : Method
publicMethodAnnotation : PublicMethod
setter : ClassMethod
members
buildMethod : ClassMethod
setterMethod : Method
writeOperations : WriteOperation
classes : Class
factoryMethods : FactoryMethod

Figure C.29.: `Builder` module – A class represents a Builder design pattern, if the class consists of a Factory Method design pattern and consists of a write operation to set properties used during the creation of objects.

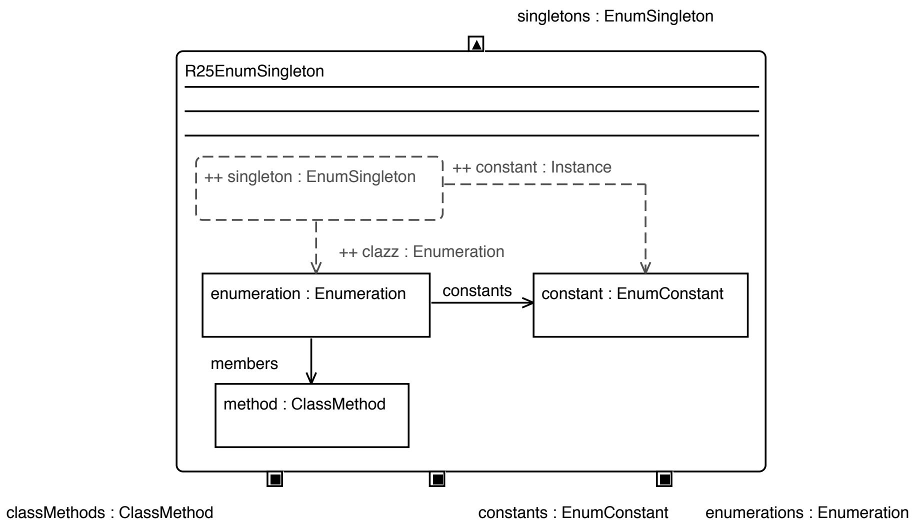


Figure C.30.: `EnumSingleton` module – An enumeration with a method and a constant can be considered as Singleton design pattern.

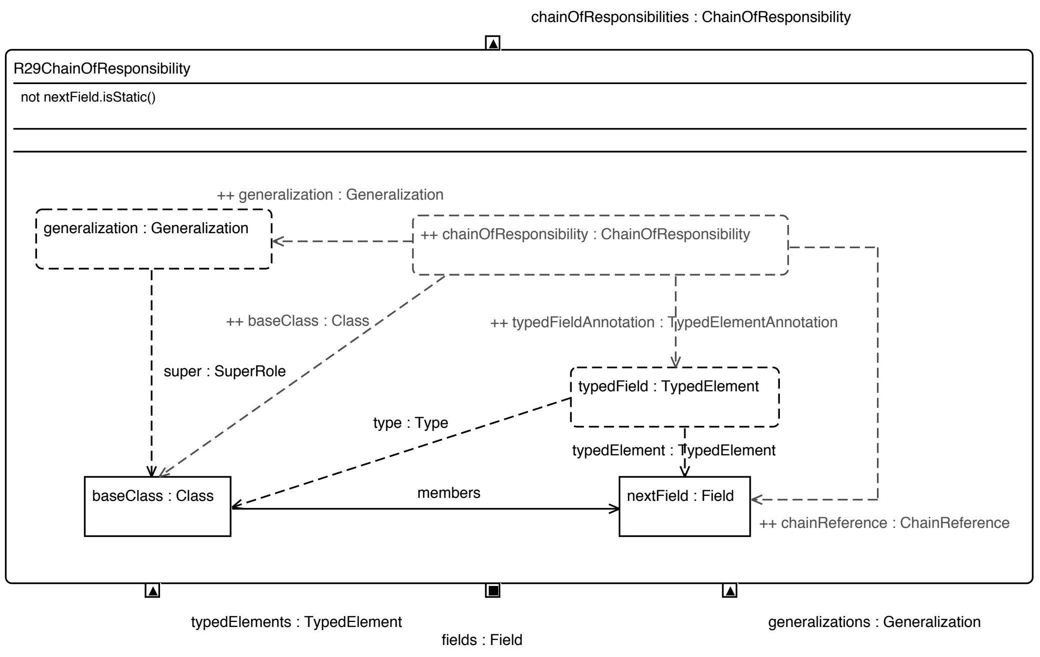

chainOfResponsibilities : ChainOfResponsibility
R29ChainOfResponsibility
not nextField.isStatic()
++ generalization : Generalization
generalization : Generalization
++ chainOfResponsibility : ChainOfResponsibility
++ baseClass : Class
++ typedFieldAnnotation : TypedElementAnnotation
super : SuperRole
typedField : TypedElement
type : Type
typedElement : TypedElement
baseClass : Class
members
nextField : Field
++ chainReference : ChainReference
typedElements : TypedElement
fields : Field
generalizations : Generalization

Figure C.31.: `ChainOfResponsibility` module – A Chain of Responsibility design pattern exists, if a non-static class field has the same type as the class. Furthermore, the class has to be a super class.

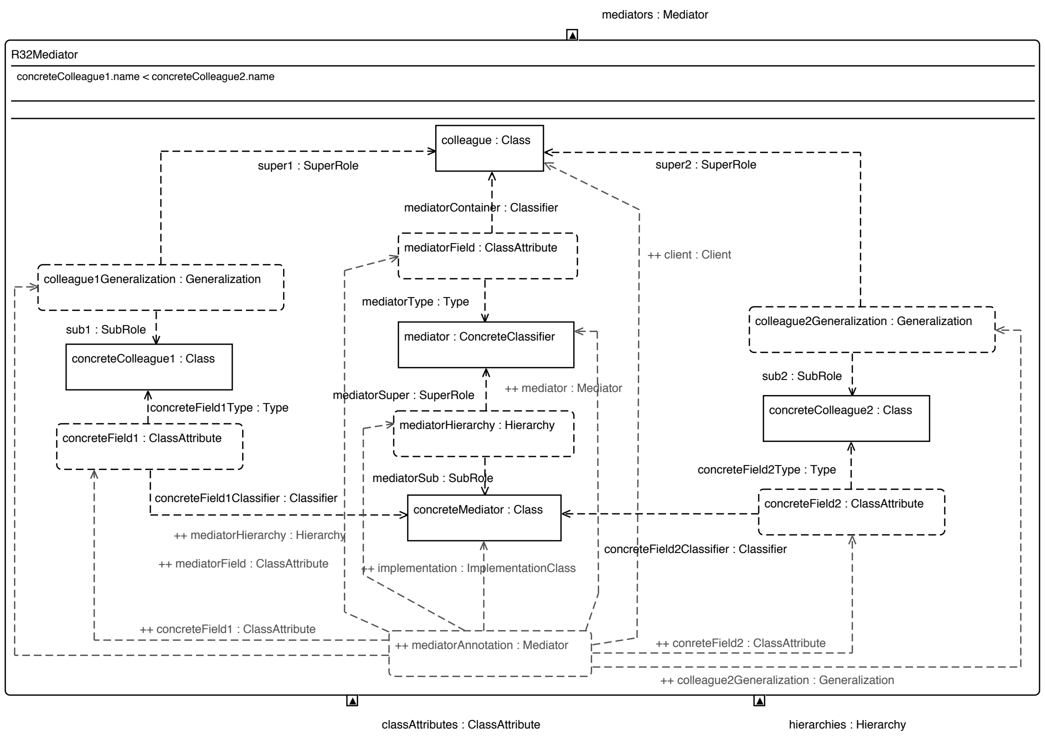

mediators : Mediator
R32Mediator
concreteColleague1.name < concreteColleague2.name
colleague : Class
super1 : SuperRole
super2 : SuperRole
mediatorContainer : Classifier
mediatorField : ClassAttribute
++ client : Client
colleague1Generalization : Generalization
mediatorType : Type
colleague2Generalization : Generalization
sub1 : SubRole
mediator : ConcreteClassifier
concreteColleague1 : Class
sub2 : SubRole
++ mediator : Mediator
concreteColleague2 : Class
concreteField1Type : Type
mediatorSuper : SuperRole
concreteField1 : ClassAttribute
mediatorHierarchy : Hierarchy
concreteField2Type : Type
mediatorSub : SubRole
concreteField1Classifier : Classifier
concreteMediator : Class
concreteField2 : ClassAttribute
++ mediatorHierarchy : Hierarchy
concreteField2Classifier : Classifier
++ mediatorField : ClassAttribute
++ implementation : ImplementationClass
++ concreteField1 : ClassAttribute
++ mediatorAnnotation : Mediator
++ coneteField2 : ClassAttribute
++ colleague2Generalization : Generalization
classAttributes : ClassAttribute
hierarchies : Hierarchy

Figure C.32.: `Mediator` module – A class represents a Mediator design pattern, if the class consists of sub classes that implement concrete mediators that consist of references to colleague classes and each colleague class knows its mediator class.

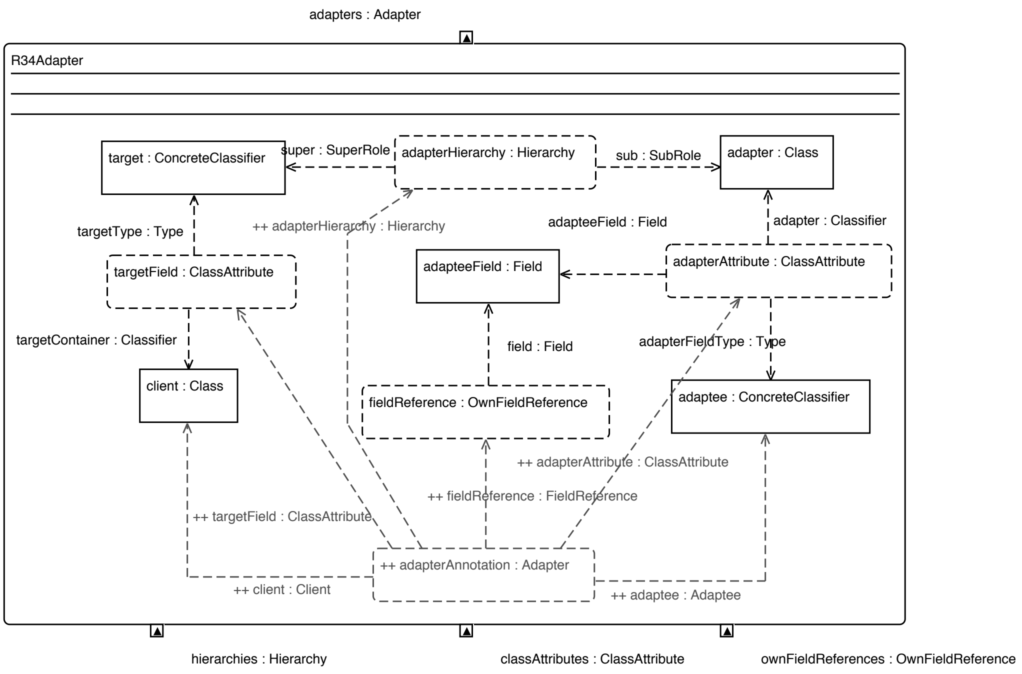

adapters : Adapter
R34Adapter
target : ConcreteClassifier
super : SuperRole
adapterHierarchy : Hierarchy
sub : SubRole
adapter : Class
targetType : Type
++ adapterHierarchy : Hierarchy
adapteeField : Field
adapter : Classifier
targetField : ClassAttribute
adapteeField : Field
adapterAttribute : ClassAttribute
targetContainer : Classifier
field : Field
adapterFieldType : Type
client : Class
fieldReference : OwnFieldReference
adaptee : ConcreteClassifier
++ adapterAttribute : ClassAttribute
++ fieldReference : FieldReference
++ targetField : ClassAttribute
++ adapterAnnotation : Adapter
++ client : Client
++ adaptee : Adaptee
hierarchies : Hierarchy
classAttributes : ClassAttribute
ownFieldReferences : OwnFieldReference

Figure C.33.: `Adapter` module – An Adapter design pattern exists, if a hierarchy (e.g., interface implementation, generalization) between two classes exists and the sub class consists a field that has the type of the class that is adapted. Furthermore, the field has to be used within the adapter class.

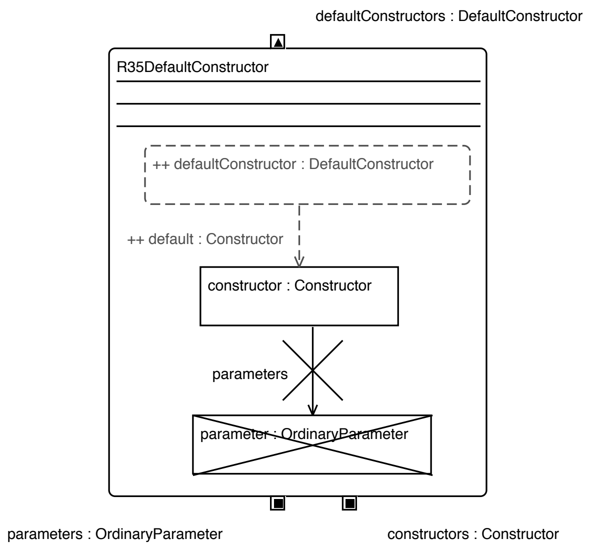
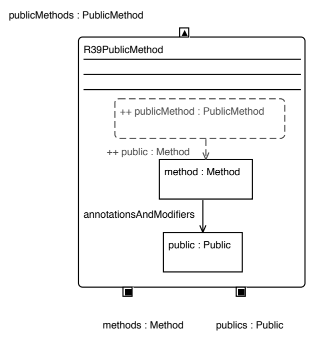
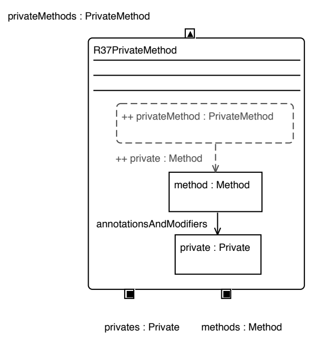

defaultConstructors : DefaultConstructor
R35DefaultConstructor
++ defaultConstructor : DefaultConstructor
++ default : Constructor
constructor : Constructor
parameters
parameter : OrdinaryParameter
parameters : OrdinaryParameter
constructors : Constructor

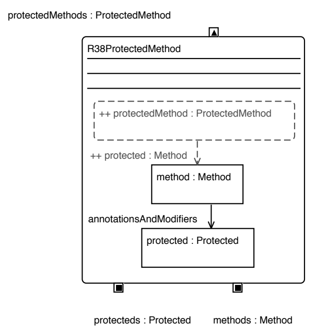
Figure C.34.: `DefaultConstructor` module – A default constructor is a constructor that does not consist of parameters.



(a) `PublicMethod` module – A method is public, if it consists of a public modifier.



(b) `PrivateMethod` module – A method is private, if it consists of a private modifier.



(c) `ProtectedMethod` module – A method is protected, if it consists of a protected modifier.

Figure C.35.: `PublicMethod` module, `PrivateMethod` module, and `Protected-Method` module

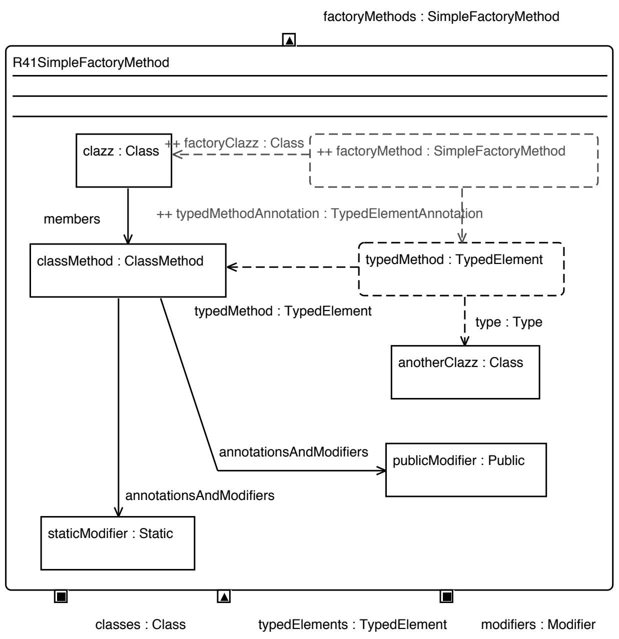

factoryMethods : SimpleFactoryMethod
R41SimpleFactoryMethod
clazz : Class
++ factoryClazz : Class
++ factoryMethod : SimpleFactoryMethod
members
++ typedMethodAnnotation : TypedElementAnnotation
classMethod : ClassMethod
typedMethod : TypedElement
typedMethod : TypedElement
type : Type
anotherClazz : Class
annotationsAndModifiers
publicModifier : Public
annotationsAndModifiers
staticModifier : Static
classes : Class
typedElements : TypedElement
modifiers : Modifier

Figure C.36.: `SimpleFactoryMethod` module – A class implements a simple factory method, if it implements a method that consists of static and public modifiers and returns an instance of another class.

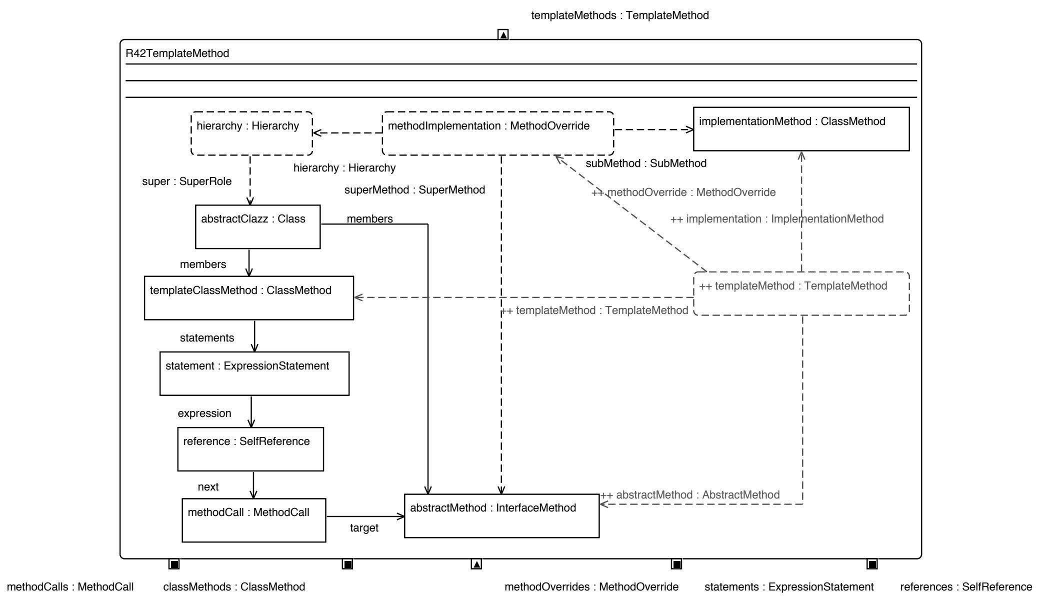

templateMethods : TemplateMethod
R42TemplateMethod
hierarchy : Hierarchy
methodImplementation : MethodOverride
implementationMethod : ClassMethod
hierarchy : Hierarchy
subMethod : SubMethod
super : SuperRole
superMethod : SuperMethod
++ methodOverride : MethodOverride
abstractClazz : Class
members
++ implementation : ImplementationMethod
members
templateClassMethod : ClassMethod
++ templateMethod : TemplateMethod
++ templateMethod : TemplateMethod
statements
statement : ExpressionStatement
expression
reference : SelfReference
next
methodCall : MethodCall
target
abstractMethod : InterfaceMethod
++ abstractMethod : AbstractMethod
methodCalls : MethodCall
classMethods : ClassMethod
methodOverrides : MethodOverride
statements : ExpressionStatement
references : SelfReference

Figure C.37.: `TemplateMethod` module – A class implements a Template Method design pattern, if the class consists of a method that calls other methods of the same class and these other methods are overriden in sub classes to refine the algorithm implemented by the template method.

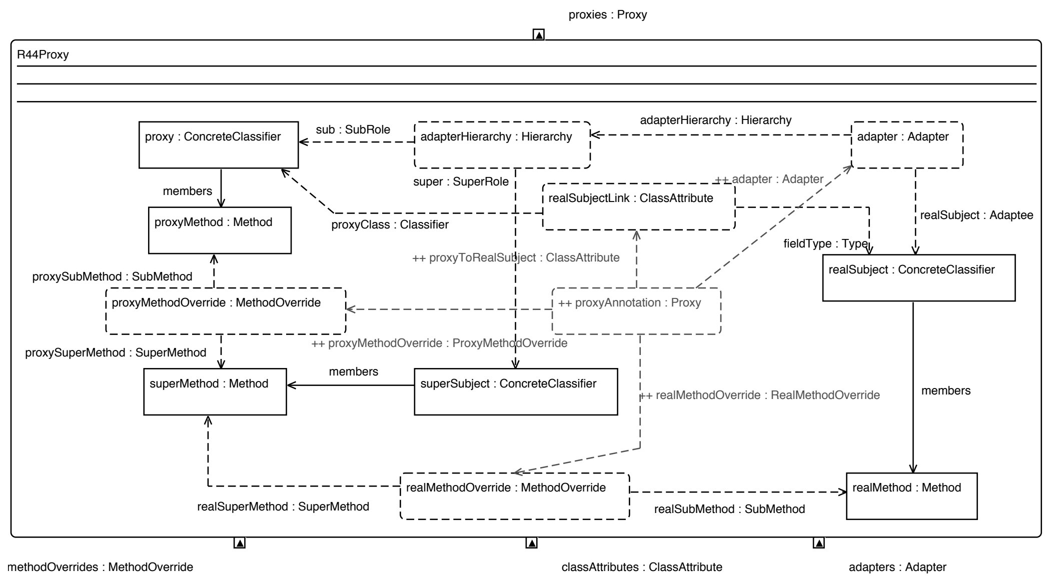

proxies : Proxy
R44Proxy
proxy : ConcreteClassifier
sub : SubRole
adapterHierarchy : Hierarchy
adapterHierarchy : Hierarchy
adapter : Adapter
members
super : SuperRole
++ adapter : Adapter
realSubjectLink : ClassAttribute
proxyMethod : Method
proxyClass : Classifier
realSubject : Adaptee
fieldType : Type
++ proxyToRealSubject : ClassAttribute
realSubject : ConcreteClassifier
proxySubMethod : SubMethod
proxyMethodOverride : MethodOverride
++ proxyAnnotation : Proxy
++ proxyMethodOverride : ProxyMethodOverride
proxySuperMethod : SuperMethod
members
superMethod : Method
superSubject : ConcreteClassifier
++ realMethodOverride : RealMethodOverride
members
realMethodOverride : MethodOverride
realMethod : Method
realSuperMethod : SuperMethod
realSubMethod : SubMethod
methodOverrides : MethodOverride
classAttributes : ClassAttribute
adapters : Adapter

Figure C.38.: `Proxy` module – A Proxy design pattern exists, if the super class of the adaptee has another sub class that overrides the same method of the super class as the adaptee.

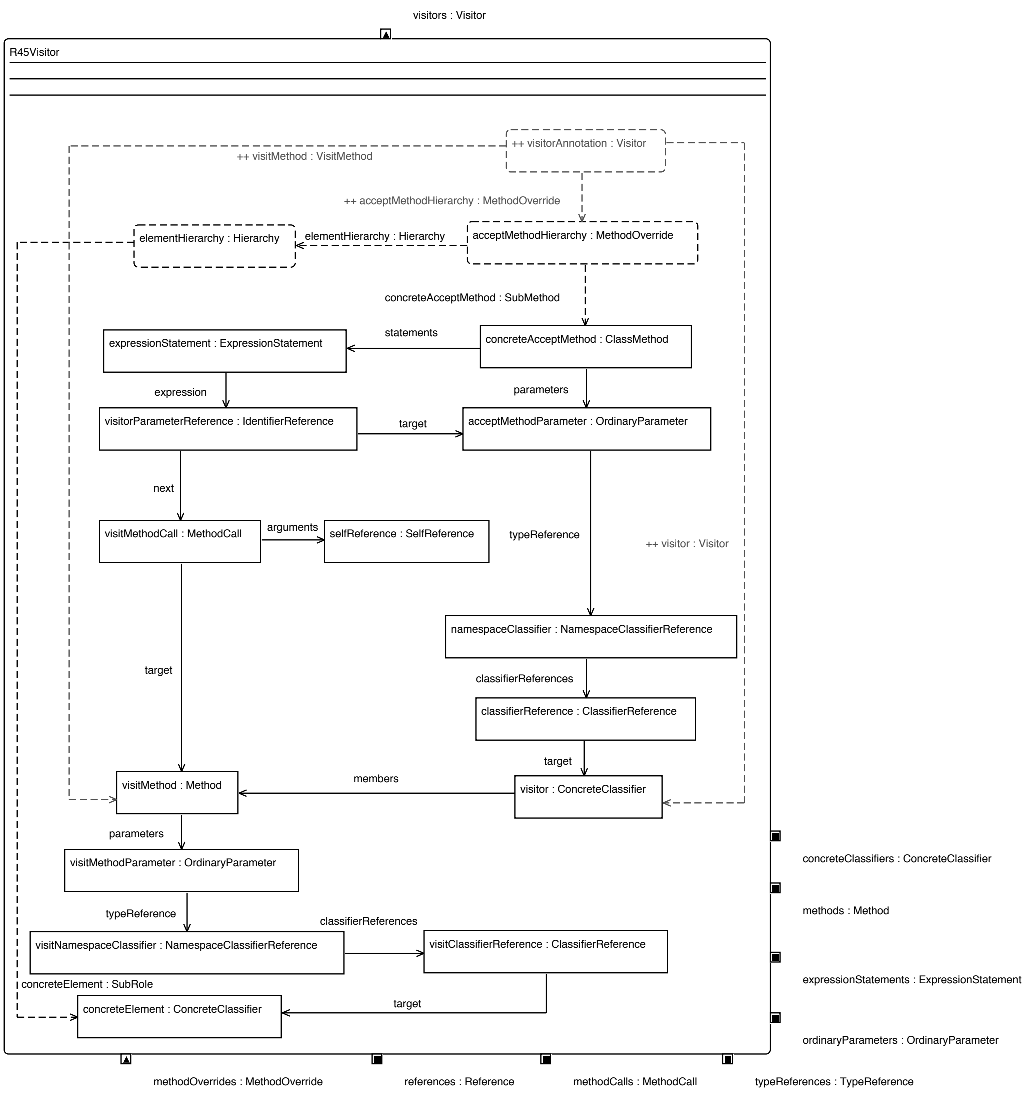

visitors : Visitor
R45Visitor
++ visitorAnnotation : Visitor
++ visitMethod : VisitMethod
++ acceptMethodHierarchy : MethodOverride
elementHierarchy : Hierarchy
elementHierarchy : Hierarchy
acceptMethodHierarchy : MethodOverride
concreteAcceptMethod : SubMethod
statements
expressionStatement : ExpressionStatement
concreteAcceptMethod : ClassMethod
expression
parameters
visitorParameterReference : IdentifierReference
target
acceptMethodParameter : OrdinaryParameter
next
visitMethodCall : MethodCall
arguments
selfReference : SelfReference
typeReference
++ visitor : Visitor
namespaceClassifier : NamespaceClassifierReference
classifierReferences
classifierReference : ClassifierReference
target
target
members
visitMethod : Method
visitor : ConcreteClassifier
parameters
visitMethodParameter : OrdinaryParameter
typeReference
classifierReferences
visitNamespaceClassifier : NamespaceClassifierReference
visitClassifierReference : ClassifierReference
concreteElement : SubRole
concreteElement : ConcreteClassifier
target
concreteClassifiers : ConcreteClassifier
methods : Method
expressionStatements : ExpressionStatement
ordinaryParameters : OrdinaryParameter
methodOverrides : MethodOverride
references : Reference
methodCalls : MethodCall
typeReferences : TypeReference

Figure C.39.: `Visitor` module – A Visitor design pattern exists, if the sub classes of a class override the accept method and call the visit method of the visitor that is passed as parameter to the overridden accept methods. Furthermore, the parameter type of the visit method(s) is a class that overrides the accept method.

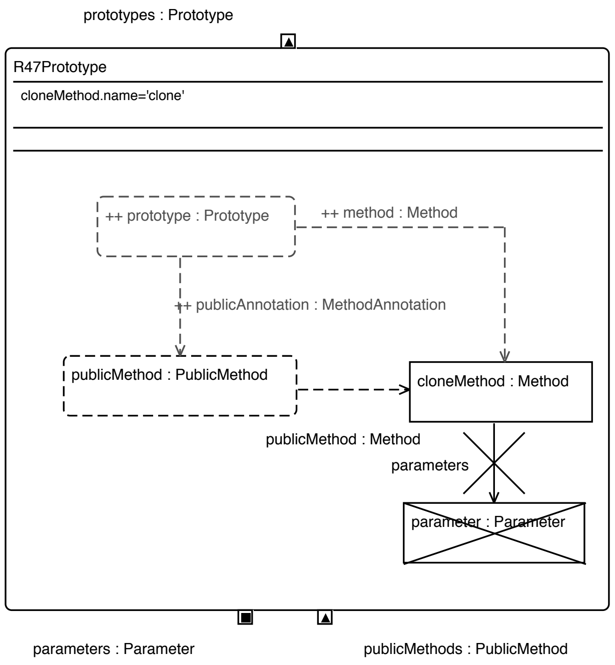


Figure C.40.: `Prototype` module – A Prototype design pattern exists, if a public method with name 'clone' exists and does not consist of parameters.

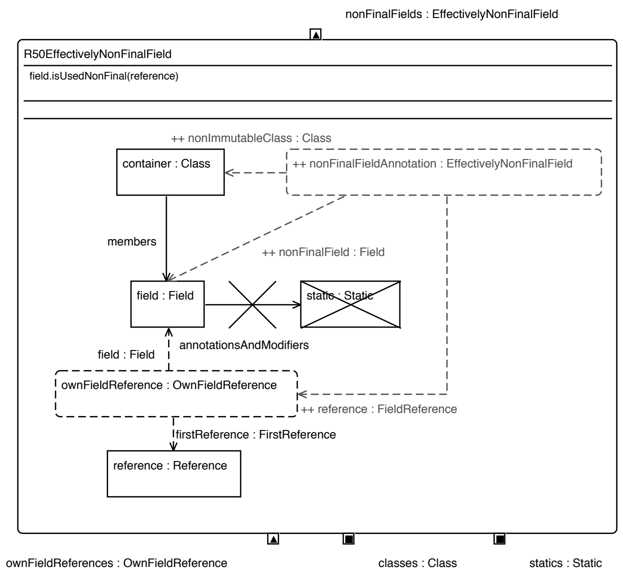

nonFinalFields : EffectivelyNonFinalField
R50EffectivelyNonFinalField
field.isUsedNonFinal(reference)
++ nonImmutableClass : Class
container : Class
++ nonFinalFieldAnnotation : EffectivelyNonFinalField
members
++ nonFinalField : Field
field : Field
static : Static
annotationsAndModifiers
field : Field
ownFieldReference : OwnFieldReference
++ reference : FieldReference
firstReference : FirstReference
reference : Reference
ownFieldReferences : OwnFieldReference
classes : Class
statics : Static

Figure C.41.: `EffectivelyNonFinalField` module – A field of a class is effectively non final, if it does not consists of a static modifier and is only modified within static or constructor code.

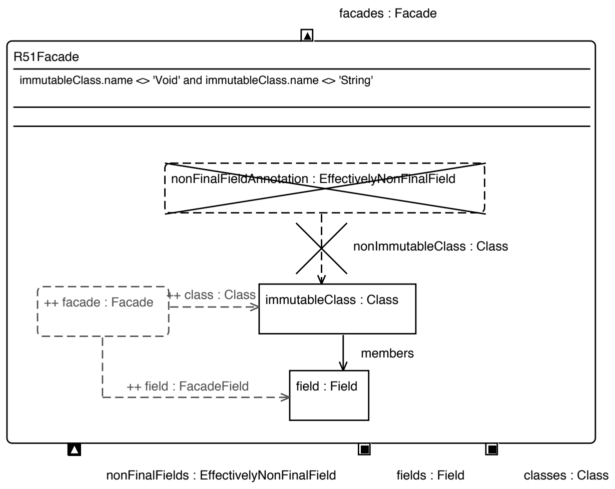

facades : Facade
R51Facade
immutableClass.name <> 'Void' and immutableClass.name <> 'String'
nonFinalFieldAnnotation : EffectivelyNonFinalField
nonImmutableClass : Class
++ facade : Facade
++ class : Class
immutableClass : Class
members
++ field : FacadeField
field : Field
nonFinalFields : EffectivelyNonFinalField
fields : Field
classes : Class

Figure C.42.: `Facade` module – A class implements the Facade design pattern, if it does not consist of effectively non final fields.

ISBN 978-3-86956-339-8
ISSN 1613-5652