Hasso Plattner Institute for Software Systems Engineering
System Analysis and Modeling Group

Thesis

A Framework for Incremental View Graph Maintenance

Dissertation
zur Erlangung des akademischen Grades
"Doktor der Ingenieurwissenschaften"
(Dr.-Ing.)
in der Wissenschaftsdisziplin
"IT-Systems Engineering"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Thomas Beyhl

Potsdam, Dezember 30th, 2017

Abstract

Nowadays, graph data models are employed, when relationships between entities have to be stored and are in the scope of queries. For each entity, this graph data model locally stores relationships to adjacent entities. Users employ graph queries to query and modify these entities and relationships. These graph queries employ graph patterns to lookup all subgraphs in the graph data that satisfy certain graph structures. These subgraphs are called graph pattern matches. However, this graph pattern matching is NP-complete for subgraph isomorphism. Thus, graph queries can suffer a long response time, when the number of entities and relationships in the graph data or the graph patterns increases.

One possibility to improve the graph query performance is to employ graph views that keep ready graph pattern matches for complex graph queries for later retrieval. However, these graph views must be maintained by means of an incremental graph pattern matching to keep them consistent with the graph data from which they are derived, when the graph data changes. This maintenance adds subgraphs that satisfy a graph pattern to the graph views and removes subgraphs that do not satisfy a graph pattern anymore from the graph views.

Current approaches for incremental graph pattern matching employ Rete networks. Rete networks are discrimination networks that enumerate and maintain all graph pattern matches of certain graph queries by employing a network of condition tests, which implement partial graph patterns that together constitute the overall graph query. Each condition test stores all subgraphs that satisfy the partial graph pattern. Thus, Rete networks suffer high memory consumptions, because they store a large number of partial graph pattern matches. But, especially these partial graph pattern matches enable Rete networks to update the stored graph pattern matches efficiently, because the network maintenance exploits the already stored partial graph pattern matches to find new graph pattern matches. However, other kinds of discrimination networks exist that can perform better in time and space than Rete networks. Currently, these other kinds of networks are not used for incremental graph pattern matching.

This thesis employs generalized discrimination networks for incremental graph pattern matching. These discrimination networks permit a generalized network structure of condition tests to enable users to steer the trade-off between memory consumption and execution time for the incremental graph pattern matching. For that purpose, this thesis contributes a modeling language for the effective definition of generalized discrimination networks. Furthermore, this thesis contributes an efficient and scalable incremental maintenance algorithm, which updates the (partial) graph pattern matches that are stored by each condition test. Moreover, this thesis provides a modeling evaluation, which shows that the proposed modeling language enables the effective modeling of generalized discrimination networks. Furthermore, this thesis provides a performance evaluation, which shows that a) the incremental maintenance algorithm scales, when the graph data becomes large, and b) the generalized discrimination network structures can outperform Rete network structures in time and space at the same time for incremental graph pattern matching.

Zusammenfassung

Heutzutage werden Graphdatenmodelle verwendet um Beziehungen zwischen Entitäten zu speichern und diese Beziehungen später abzufragen. Jede Entität im Graphdatenmodell speichert lokal die Beziehungen zu anderen verknüpften Entitäten. Benutzer stellen Suchanfragen um diese Entitäten und Beziehungen abzufragen und zu modifizieren. Dafür machen Suchanfragen Gebrauch von Graphmustern um alle Teilgraphen in den Graphdaten zu finden, die über bestimmte Graphstrukturen verfügen. Diese Teilgraphen werden Graphmusterübereinstimmung (Match) genannt. Allerdings ist diese Suche nach Matches NP-vollständig für Teilgraphisomorphie. Daher können Suchanfragen einer langen Antwortzeit unterliegen, wenn die Anzahl von Entitäten und Beziehungen in den Graphdaten oder -mustern ansteigt.

Eine Möglichkeit die Antwortzeiten von Suchanfragen zu verbessern ist Matches für komplexe Suchanfragen in sogenannten Sichten über die Graphdaten für spätere Suchanfragen bereitzuhalten. Allerdings müssen diese Sichten mittels einer inkrementellen Suche nach Matches gewartete werden um sie konsistent zu den sich ändernden Graphdaten zu halten. Diese Wartung ergänzt Teilgraphen, die Graphmuster erfüllen, in den Sichten und entfernt Teilgraphen, die Graphmuster nicht mehr erfüllen, aus den Sichten.

Aktuelle Ansätze für die inkrementelle Suche nach Matches verwenden Rete Netzwerke. Rete Netzwerke sind sogenannte Discrimination Networks, die alle Matches für bestimmte Suchanfragen aufzählen und warten, indem sie ein Netzwerk aus einzelnen Teilgraphmustern anwenden, die zusammen eine vollständige Suchanfrage ergeben. Das Netzwerk speichert für jedes Teilgraphmuster welche Teilgraphen das Teilgraphmuster erfüllen. Daher haben Rete Netzwerke einen hohen Speicherverbrauch, da sie alle Zwischenergebnisse speichern müssen. Jedoch sind es gerade diese gespeicherten Zwischenergebnisse, die es dem Rete Netzwerk ermöglichen die gespeicherten Zwischenergebnisse effizient zu warten, da diese Zwischenergebnisse für das Auffinden neuer Matches ausgenutzt werden. Allerdings existieren andere Arten von Discrimination Networks, die hinsichtlich Speicher- und Zeitverbrauch effizienter sind, aber derzeit nicht für die inkrementelle Suche nach Matches verwendet werden.

Diese Doktorarbeit wendet eine verallgemeinerte Art von Discrimination Networks an. Diese verallgemeinerte Art ermöglicht es die Balance zwischen Speicher- und Zeitverbrauch für die inkrementelle Suche nach Matches zu steuern. Dafür stellt diese Arbeit eine Modellierungssprache vor, die es ermöglicht verallgemeinerte Arten von Discrimination Networks effektiv zu spezifizieren. Darauf aufbauend stellt diese Arbeit einen inkrementellen Algorithmus vor, der die gespeicherten Matches effizient und skalierbar wartet. Abschließend stellt diese Arbeit eine Evaluierung vor, die aufzeigt dass die Modellierungssprache eine effektive Spezifikation von verallgemeinerten Discrimination Networks erlaubt. Außerdem zeigt die Evaluierung, dass a) der inkrementelle Wartungsalgorithmus für große Graphdaten skaliert und b) die Netzwerkstrukturen von verallgemeinerten Discrimination Networks im Vergleich zu den Netzwerkstrukturen von Rete Netzwerken im Speicher- und Zeitverbrauch für die inkrementelle Suche nach Matches effizienter sein können.

Contents

1. Introduction

Graphs are general-purpose data structures [83] that store entities and relationships between these entities. Graph nodes represent entities and graph edges between graph nodes represent relationships between these entities.

Use cases for graphs are manifold. For example, social networks are graphs that enable to leverage information about people and their connections to gain competitive and operational advantages [82, p. 106]. In social networks, graph nodes represent people and graph edges represent relationships between people. People either volunteer these relationships, e. g., when they add another person as friend, or relationships are derived from the gathered data in the social network, e. g., from similar interests of two friends. Based on such social networks, recommendation systems emerge as additional use case for graphs [82, p. 107]. Recommendation systems analyze relationships between people and things based on the behaviors, interests, and purchases of people to recommend other interesting things to them such as products that were bought by other people with similar interests. Today's IT infrastructures base on telecommunication networks. Telecommunication networks are also use cases for graphs. In telecommunication networks, graph nodes represent components such as server and client computers, routers, and switches. Graph edges represent wired and wireless connections between such components. The representation as graph structure enables capacity planning and impact analyses [54, p. 98]. When dealing with large IT infrastructures, the management of complex access rights for users and groups of users is a big issue. Users and groups with access permissions and prohibitions for certain files and folders in a file system constitute a graph [54, p. 97]. Graph nodes represent users, groups, files, and folders. Graph edges represent relationships between them, e. g., access permissions [82, p. 110].

Figure 1.1 shows how such graphs are processed. The graph storage (abbr. storage) stores graphs and the graph pattern matching engine (abbr. matching engine) evaluates graph queries (abbr. queries) that are stated by users. These queries either search for graph nodes and edges that satisfy certain graph patterns (abbr. patterns) or apply graph transformations (abbr. transformations) to modify graphs. A graph pattern describes a graph structure that has to be found in graphs. For example, users search for cliques of friends in social networks. A graph transformation adds, deletes, or modifies graph nodes and edges. For example, users build or break friendships in social networks. The matching engine employs graph pattern matching (abbr. pattern matching) to search for graph nodes and edges that satisfy the patterns and to apply transformations. When the matching engine evaluates patterns, the pattern matching looks up subgraphs in the graph storage that satisfy the patterns and returns these subgraphs as graph query result (abbr. query result). A subgraph satisfies a pattern, if a structure-preserving mapping between the graph nodes and edges of the pattern and the subgraph exists. The retrieved subgraphs are called graph pattern matches (abbr. matches).

However, graph pattern matching is NP-complete for subgraph isomorphism [36]. Thus, queries can suffer a long response time, when the number of graph nodes and graph edges in graphs or patterns increases. In general, three possibilities exist how to increase the throughput of the matching engine, when evaluating queries. Figure 1.1 shows graph indexes, graph slicing, and graph views as possibilities to increase the throughput.

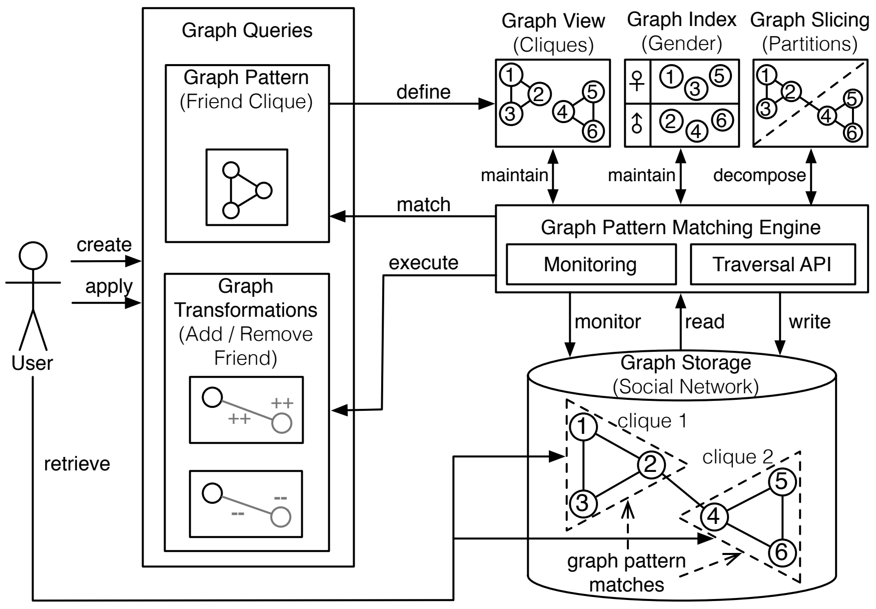


Figure 1.1.: Overview of graph pattern matching

The first possibility deals with graph indexes (abbr. indexes) that index graph nodes and edges concerning their properties, e. g., types, labels, and attribute values. For example, an index can enumerate all female and male persons in a social network. Then, the matching engine uses the indexes to efficiently look up graph nodes and edges with certain properties, e. g., certain types, labels, and attribute values. For example, Goldman et al. [42] group graph nodes in the index that are reachable via the same path in the graphs.

The second possibility deals with graph slicing (abbr. slicing) to enable the parallel evaluation of patterns during the pattern matching. The parallel evaluation of patterns is called *parallel* graph pattern matching. For that purpose, the matching engine slices graphs into partitions that can be processed independently by multiple and also distributed matching engines. Finding such a slicing of graphs is a challenging task, because matches may be distributed over multiple partitions that are processed by independent matching engines. When naively slicing graphs, the matching engines do not find these matches. For example, Krause et al. [64] employ the Bulk Synchronous Parallel model.

The third possibility deals with graph views (abbr. views), which store and maintain matches as result of queries. The overall process is called *incremental* graph pattern matching. Users provide these queries in terms of graph patterns, beforehand. Then, the matching engine executes the queries to retrieve and, afterwards, store all matches in the view, for later retrieval. For example, the view can store all matches for the friend clique query. When users state queries, the matching engine looks up the matches in the view in constant time[1]. However, the view must be kept up-to-date to ensure that the stored matches are equal to the matches in the graphs. In general, views maintain these matches by re-evaluating the queries only for changed portions of the graphs. For that purpose, the matching engine monitors the graphs in the graph storage and uses the captured changes to prune the search spaces of queries. For example, Bergmann et al. [7] employ incremental graph pattern matching for querying graphs in Model-Driven Engineering (MDE).

[1]Excluding the linear cost induced by the number of retrieved graph pattern matches.

This thesis focuses on graph views to a) increase the throughput of graph queries and b) hide, enrich, as well as restructure query results for later retrieval. When *hiding parts of query results* in views, certain graph nodes and edges are filtered out to neglect details that are not relevant for the purposes of the queries. When *enriching query results* with additional information, queries infer additional knowledge and assign additional roles to graph nodes and edges that are part of the query results. When *restructuring query results*, multiple views are reused to derive additional views that set the content of these views into new relationships.

This thesis employs graph views, because it is more efficient and economical to re-evaluate patterns only for small portions of graphs a second time as *incremental* pattern matching approaches do than re-evaluate patterns for the complete graphs from scratch as *parallel* pattern matching approaches do, when graphs change. Thus, parallel pattern matching is not in the scope of this thesis. Furthermore, graph indexes are not in the scope of this thesis, because they do not store query results in terms of graph pattern matches.

The contributions of this thesis are a) a modeling language for the definition of graph views, b) maintenance algorithms that keep these views up-to-date, when graphs change, and c) an evaluation that shows the effectiveness and efficiency of the proposed approach in comparison to the state of the art of incremental graph pattern matching.

The modeling language enables to model definitions of views that a) enable to specify the content of views, b) enable to maintain matches that are stored by views, and c) enable to steer the trade-off between the memory consumption for storing matches and the time required for maintaining these matches by means of modular graph patterns.

The maintenance algorithms process these definitions of views to initially create and, afterwards, maintain the matches that are stored by graph views. When re-evaluating queries to update the views, the maintenance algorithms process graphs incrementally by considering only changed portions of these graphs. In contrast to existing approaches for incremental graph pattern matching, the approach in this thesis is a native solution that employs graph transformations to create and maintain views. The existing approaches are no native solutions, because they map incremental graph pattern matching back to the relational domain.

The evaluation shows that the modeling language and maintenance algorithms together *can* outperform current techniques for incremental graph pattern matching in time and space at the same time, when they are employed for the maintenance of views.

The remainder of this chapter is organized as follows. Section 1.1 describes the research topic that is addressed in this thesis by discussing modeling, algorithmic, and realization challenges regarding the effective specification as well as efficient and scalable maintenance of graph views. Afterwards, Section 1.2 describes the overall goals of this thesis. Section 1.3 summarizes the state of the art in incremental graph pattern matching. Section 1.4 discusses the contributions of this thesis in comparison to the state of the art. Finally, Section 1.5 outlines the overall structure of this thesis.

1.1. Research Topic

The research topic of this thesis deals with retrieving domain knowledge that is embedded in graph-structured data. This thesis refers to domain knowledge as knowledge of a certain domain or discipline. Graphs store this domain knowledge as adjacent graph nodes that are connected by graph edges. This thesis refers to these graphs as *base graphs*.

Users employ graph queries to a) retrieve domain knowledge from base graphs and b) modify the domain knowledge stored by base graphs. At any time, the graph views must store all graph pattern matches retrieved by queries, also when base graphs change. For that purpose, the views must be kept up-to-date. Therefore, the research topic deals with the efficient and scalable maintenance of the views to ensure that the matches stored by the views are consistent with the base graphs. The research topic of this thesis focuses on the efficiency and scalability of this view maintenance. In this thesis, the term efficient maintenance refers to the capability of the system to avoid wasting computational resources such as computation time and memory footprint, when maintaining views. The term scalable maintenance refers to the ability of the system to not decrease its maintenance performance for views, when the number of graph nodes and edges of base graphs increases.

Furthermore, the research topic aims for a generic approach that is applicable to different kinds of graph data models and query languages.

The remainder of this section is organized as follows. Section 1.1.1 discusses challenges concerning the specification of graph views. This thesis refers to these kinds of challenges as modeling challenges. Section 1.1.2 discusses challenges concerning the efficiency and scalability of the view maintenance. This thesis refers to these kinds of challenges as algorithmic challenges. Furthermore, this thesis aims for the realization of the proposed concepts. Section 1.1.3 discusses arising realization challenges.

1.1.1. Modeling Challenges

The modeling challenges in this thesis deal with the effective specification of graph views, which describes the kinds of graph pattern matches that are stored by views.

Notion of Graph Views (C1a)

When talking about graph views several questions arise. For example, what exactly are views? Are views just collections of graph nodes [101]? Or, are views collections of graph nodes and edges? If graph edges are included in views, are all graph edges between adjacent graph nodes included or only selected graph edges? Besides the content of views, are views collections of tuples as in relational databases? Or, are views graphs, too?

Data Model of Graph Views (C1b)

Different kinds of graph data models exist [2]. The fundamental question that arises is which kind of data model should be employed for graph views. Is a relational data model appropriate or is a graph data model the better choice?

Definition of Graph Views (C1c)

When a clear notion of views exists and an appropriate data model is chosen for views, the question arises how to define views? The definition of views specifies the content of views. Are declarative, imperative, or both kinds of query languages appropriate for the definition of graph views? How expressive must be the language for such definitions? Furthermore, the chosen definition language must permit the efficient and scalable maintenance of graph views.

Reuse of Graph Views (C1d)

When graph queries are composed of other queries, definitions of graph views should be able to refer to graph pattern matches that are stored by views that consist of matches for the other queries. Then, the question arises how to refer to certain graph nodes and edges that are part of the other views and have special roles in these views? How to refer to these graph nodes and edges effectively and efficiently, i. e., without the need to check the role of these graph nodes and edges again. Then, the question arises how graph nodes and edges of different views can be combined to define derived additional views. How to combine definitions of views in terms of conjunctions and disjunctions? How to express that definitions of views are reused in negative sense, i. e., that certain graph pattern matches must not exist? How to permit definitions of views that store graph pattern matches for graph queries that employ recursive definitions of graph patterns?

1.1.2. Algorithmic Challenges

The algorithmic challenges of this thesis deal with the efficient and scalable maintenance of graph views to keep these views consistent with the base graphs from which they are derived.

Materialization of Graph Views (C2a)

When graph views are derived from base graphs, several questions arise on how to materialize these views. For example, are graph nodes and edges in graph views copies of graph nodes and edges of base graphs [101]? Or, are graph nodes and edges of views delegates that refer back to graph nodes and edges of base graphs? Then, views are only meaningful together with the base graphs from which the views are derived. Or, does a view abstract from graph nodes and edges and employs a different concept to store graph nodes and edges that are part of graph pattern matches?

Consistency of Graph Database Views (C2b)

When base graphs change, the derived graph views must be kept consistent with base graphs by means of a view maintenance. For example, when matches do not exist in base graphs anymore, these matches must be also removed from the derived views. When new matches occur in base graphs, these matches must be also added to the derived views. Then, the question arises how to perform this view maintenance efficiently and scalable. Deriving views always from scratch although only few graph nodes and edges of base graphs changed does not scale with the size of the base graphs, because the time that is required to maintain the derived views depends on the number of graph nodes and edges that are stored by these base graphs. Instead, only added, deleted, and modified graph nodes and edges of base graphs should be input to the view maintenance. Then, the effort that is required to perform the maintenance only depends on the number of these graph nodes and edges.

Impact of Graph Changes on Graph Views (C2c)

When base graphs change, the derived graph views must be maintained. It depends on the actual changes of base graphs and on the actual definitions of the views whether derived views must be maintained or not. Then, the question arises how to efficiently look up affected graph pattern matches that are stored by views without the need to explore all stored matches?

1.1.3. Realization Challenges

For an evaluation of the concepts, which are developed in this thesis, the following realization challenges have to be solved appropriately.

Cope with Different Graph Data Models (C3a)

Angles [2] states that different graph data models are employed to store domain knowledge by means of graphs. For example, labeled graphs, typed graphs with type inheritance, hypergraphs with n-ary graph edges, and nested graphs with nested graph nodes exist. Furthermore, graph nodes and edges can own attributes. Moreover, graph edges can be undirected or directed. These different graph data models introduce two realization challenges.

First, it depends on the actual domain knowledge, which kind of graph data model fits best to store graphs. Therefore, this thesis has to provide generic concepts that work for as many different graph data models as possible.

Second, the graph data models have certain characteristics such as directed graph edges that cannot be traversed in backward direction easily. However, when maintaining views these edges must be traversable in forward and backward direction. Therefore, this thesis has to cope with the limitations of graph data models for view maintenance.

Cope with Different Graph Query Languages (C3b)

In this thesis, users employ query languages a) to query and modify base graphs and b) to define the content of views. As stated by Angles [2], users express queries by means of declarative and imperative query languages. For example, Cypher [82] is a declarative query language that enables to describe how graph pattern matches that belong to the graph query result look like. For example, Neo4j [82] provides a traversal application programming interface (API) for several programming languages [82] that enables to implement queries with imperative statements, which describe the algorithm to look up graph pattern matches. Therefore, this thesis has to cope with queries that are expressed in different query languages. In the worst case, queries are black boxes that hide the actual graph queries. That means, the proposed approach must not rely on the characteristics of certain query languages.

1.2. Goals of this Thesis

The overall goal of this thesis is to provide a framework for the incremental maintenance of graph pattern matches by means of graph views, when graph data models are employed to store graphs and graph pattern matches. Johnson et al. [56] state that a framework *"is an abstract design for solutions to a family of related problems"*. Johnson et al. [56] state that *"methods defined by users to tailor the framework will often be called from within the framework itself, rather than from the user's application code"* and that *"the framework often plays the role of the main program in coordinating and sequencing application activity"*. Johnson et al. [56] explain that a *"way to customize a framework is to supply it with a set of components that provide the application specific behavior"*. Finally, Johnson et al. [56] describe that these components must implement a certain interface, which has to be understood by the user, who implements these components.

Accordingly, this thesis aims for a framework that provides an architecture design for creating and maintaining graph views using the technique of incremental graph pattern matching. This thesis aims for a framework that is easily applicable, when views of arbitrary graph domains have to be maintained. That means, the users of the framework must be only responsible to describe which kind of domain knowledge has to be retrieved from and maintained by views. The users of the framework do not have to care about the maintenance of these views on their own.

Figure 1.2 gives an overview of the goals of this thesis. These goals are a modeling language (G1) that enables to define graph views, the embedding of graph queries as means to define the content of graph views (G2), maintenance algorithms that keep graph views consistent with their base graphs (G3), and a realization of the proposed concepts (G4) to enable an evaluation of these concepts (G5). The following sections describe each major goal.

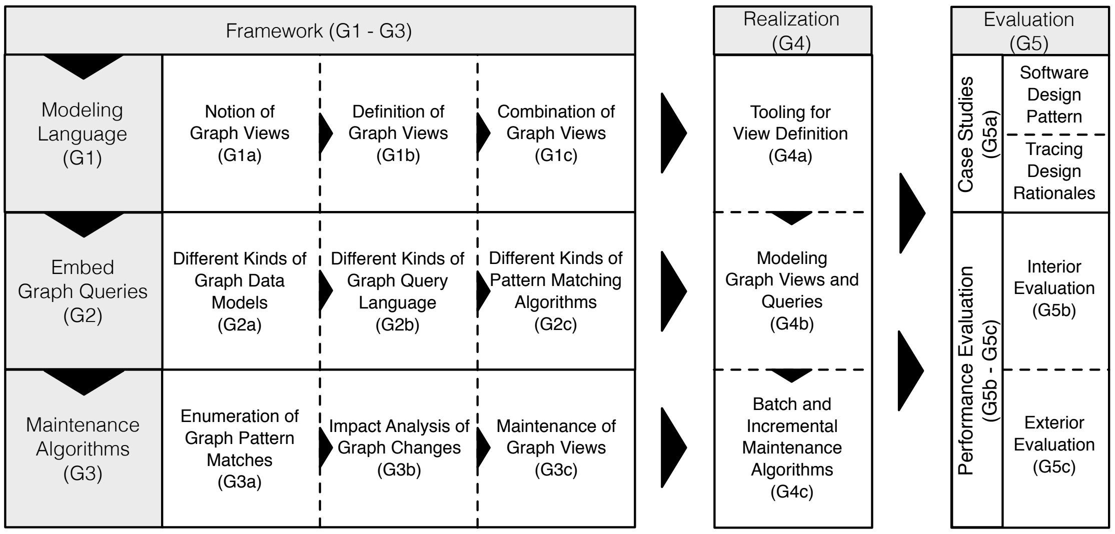

Framework (G1 - G3)
Realization (G4)
Evaluation (G5)
Modeling Language (G1)
Notion of Graph Views (G1a)
Definition of Graph Views (G1b)
Combination of Graph Views (G1c)
Tooling for View Definition (G4a)
Case Studies (G5a)
Software Design Pattern
Tracing Design Rationales
Embed Graph Queries (G2)
Different Kinds of Graph Data Models (G2a)
Different Kinds of Graph Query Language (G2b)
Different Kinds of Pattern Matching Algorithms (G2c)
Modeling Graph Views and Queries (G4b)
Performance Evaluation (G5b - G5c)
Interior Evaluation (G5b)
Maintenance Algorithms (G3)
Enumeration of Graph Pattern Matches (G3a)
Impact Analysis of Graph Changes (G3b)
Maintenance of Graph Views (G3c)
Batch and Incremental Maintenance Algorithms (G4c)
Exterior Evaluation (G5c)

Figure 1.2.: Overview of goals

1.2.1. Modeling Language for Graph Views (G1)

This thesis aims for a platform- and technology-independent approach that enables to embed graph queries into an overall architecture to derive and maintain graph views. Based upon this architecture, generic maintenance algorithms for graph views can be developed that only base on the concepts of this architecture and abstract from the implementation details of the embedded queries. Then, the users can stay focused on the definition of graph views by means of graph queries and do not have to care about the maintenance of these views.

The expected outcome of this goal setting is a modeling language that enables to define graph views that can dependent on each other. The modeling language must be in line with the maintenance algorithms for these views. For that purpose, a notion of graph views has to be developed (G1a). Furthermore, the modeling language has to enable a) the definition of single views (G1b) and b) the combination of multiple views (G1c) to an overall view model.

Notion of Graph Views (G1a)

This thesis must develop a clear notion of graph views, which enable to store graph pattern matches. Then, an appropriate graph data model for views has to be provided, which enables to enumerate all graph pattern matches effectively that are retrieved by graph queries.

Definition of Single Graph Views (G1b)

This thesis must develop concepts that enable to derive single graph views by means of user-defined graph queries. Thus, the modeling language must enable to embed user-defined graph queries into the overall view model that defines the views. The proposed modeling language must not restrict users to a certain graph query language, because this thesis aims for a generic framework that enables to define and maintain views independent from certain graph pattern matching technologies and query languages.

Combination of Multiple Graph Views (G1c)

The proposed modeling language must enable to combine definitions of graph views, because definitions of views should be able to reuse graph pattern matches that are stored by other views to avoid matching equal (sub-) patterns multiple times. Therefore, the approach must enable the definitions of views that refer to the definitions of other views to reuse their graph pattern matches. Then, an arbitrary granularity level should be supported to enable users to choose a granularity level that fits best for the domain knowledge that they want to retrieve from and maintain in views. Therefore, the modeling approach neither should enforce a certain granularity level for definitions of views nor should enforce a certain dependency structure of views. Instead, general-purpose operations must be supported that enable users to combine definitions of views easily. For example, users would like to express conjunctions, disjunctions, negations and also recursive definitions, when they combine views.

1.2.2. Embed Different Kinds of Graph Queries (G2)

This thesis aims for a framework that enables users to employ imperative and declarative graph queries for the definition and maintenance of graph views, because both kinds of queries are employed to query graphs, in practice. Then, the view maintenance must be designed in a way that the maintenance algorithms do not exploit implementation details of graph queries to create and maintain graph views.

The expected outcome of this goal setting are concepts that enable to apply the framework to different kinds of graph data models (G2a), different kinds of graph query languages (G2b), and different kinds of graph pattern matching algorithms (G2c).

Support Different Kinds of Graph Data Models (G2a)

In practice, different graph data models can be employed to store graphs [2]. Thus, the framework must support different kinds of graph data models and the developed concepts must be easily applicable to these graph data models.

Support Different Kinds of Graph Query Languages (G2b)

One goal of this thesis is to support arbitrary graph query languages for the definition and maintenance of graph views. The framework has to enable that these kinds of queries languages can be employed at the same time, because these kinds of queries are employed simultaneously to query graphs, in practice. The developed concepts must be independent from the employed query languages.

Support Different Kinds of Graph Pattern Matching Algorithms (G2c)

When different kinds of graph queries have to be supported by the framework, the framework also has to support different kinds of pattern matching algorithms. For example, different graph pattern matching algorithms implement injective and non-injective pattern matching, i. e., different nodes of a graph pattern can or cannot match the same node of a graph.

1.2.3. Incremental Maintenance of Deductive Graph Database Views (G3)

This thesis employs graph views to keep ready answers for graph queries. These views must be kept up-to-date in a manner that they store exactly the same graph pattern matches, which also exist in the base graphs from which the views are derived. Thus, the views must be maintained to ensure that they are consistent with their base graphs. This thesis aims for an efficient and scalable maintenance of graph views that only depends on the number of changed graph nodes and edges of base graphs concerning the execution time of the maintenance.

The expected outcome of this goal setting are concepts how to materialize views (G3a), how to efficiently and scalable relate changes of base graphs to graph pattern matches that are stored by views (G3b), and algorithms for the incremental maintenance of graph views (G3c).

Enumeration of Graph Pattern Matches (G3a)

This thesis must provide concepts to enumerate graph pattern matches by means of graph views. These concepts must not copy graph nodes and edges from base graphs to view graphs, because these copies require additional memory and must be kept synchronized with the base graphs. Instead, this thesis must provide concepts for the enumeration of graph pattern matches that are memory-efficient.

Impact Analysis of Graph Changes (G3b)

This thesis must relate changes of base graphs efficiently to graph pattern matches that are stored by graph views and to matches that are currently missing in these views. The framework must preserve matches in the views that still exist in base graphs. The framework must remove matches from views that disappeared from base graphs. The framework must add matches to views that additionally appeared in base graphs. Furthermore, the changes of views must be propagated to dependent views as well.

Maintenance of Graph Database Views (G3c)

Deriving graph views always from scratch although only few graph nodes and edges of base graphs changed is inefficient and does not scale, especially when the number of graph nodes and edges of base graphs increases. Instead, an approach is required that prunes the search space to a minimal subset of graph nodes and edges, which must be at least considered to keep derived views consistent with their base graphs, when maintaining views. This thesis aims for an approach that only depends on the number of changed graph nodes and edges of base graphs concerning the execution time of the maintenance to enable an efficient and scalable maintenance of views.

1.2.4. Realization of Concepts (G4)

This thesis presents concepts for the definition of graph views by embedding graph queries into these definitions and maintaining the views, when base graphs change. This thesis aims for the realization of these concepts for evaluation purposes.

The expected outcome of this goal setting are software tools that enable to model definitions of graph views (G4a). Furthermore, graph views have to be modeled and graph queries have to be created (G4b) with the help of these software tools for the evaluation of the proposed modeling concepts. Moreover, the proposed maintenance algorithms have to be implemented for a performance evaluation (G4c).

Tooling for View Definition (G4a)

For the modeling language, a graphical editor has to be realized that enables to model definitions of graph views in concrete syntax and embed graph queries into these definitions.

Modeling Views and Graph Queries (G4b)

This thesis aims for the modeling of graph views and graph queries for conducting case studies. For that purpose, editors have to be provided that enable to model views and embed queries.

Batch and Maintenance Algorithms (G4c)

This thesis aims for a realization of the proposed maintenance algorithms. For a performance evaluation, the batch and incremental view maintenance algorithms have to be implemented.

1.2.5. Evaluation of Concepts (G5)

This thesis provides a modeling language and maintenance algorithms for graph views. This thesis aims for an evaluation of the modeling language and maintenance algorithms.

The expected outcome of this goal setting are case studies, which demonstrate that a) the proposed modeling language is appropriate (G5a), b) the developed algorithms outperform alternative algorithms presented in this thesis (G5b), and c) the proposed maintenance algorithms perform better than existing algorithms for incremental pattern matching (G5c).

Case Studies (G5a)
This thesis aims for an evaluation of the proposed modeling language in terms of two case studies. The first case study deals with the recovery of employed software design patterns from abstract syntax graphs. The second case study deals with the tracing of design rationales.

Interior Evaluation (G5b)
This thesis provides batch and incremental algorithms for the maintenance of graph views. One goal of this thesis is to compare the performance of the incremental maintenance algorithms with a comparable batch algorithm that performs a real maintenance of graph views.

Exterior Evaluation (G5c)
This thesis provides concepts that are extensions of existing concepts. Therefore, this thesis aims for a comparison of the proposed algorithms with existing approaches to prove that the concepts presented in this thesis pay off in comparison to existing approaches.

1.3. State of the Art

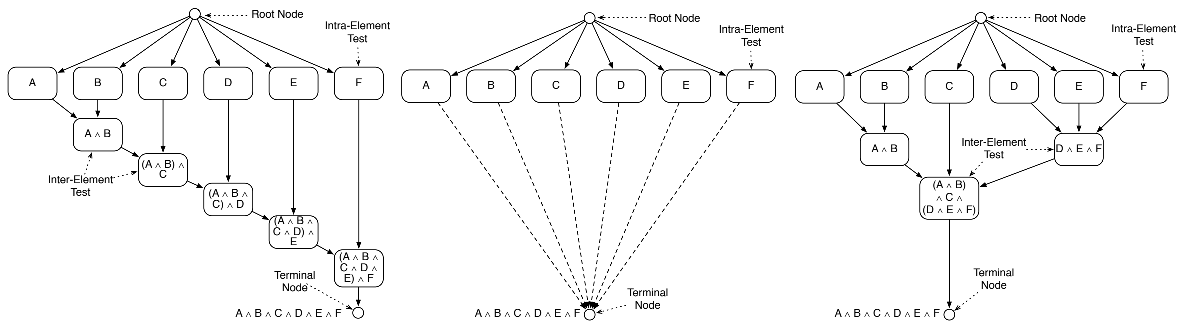
This section introduces the state of the art of incremental view maintenance and incremental graph pattern matching. This section focuses on related work that needs to be introduced for the discussion of the contribution of this thesis. Section 1.3.1 introduces discrimination networks for the view maintenance of relational data. Section 1.3.2 describes incremental graph pattern matching as technique for the view maintenance of graph-structured data. Finally, Section 1.3.3 derives observations from the presented state of the art.

1.3.1. Discrimination Networks for View Maintenance

Discrimination networks enumerate all objects that satisfy certain condition tests and enable to update these enumerations efficiently, when objects change. Originally, discrimination networks are employed for rule triggering in rule-based systems and the view maintenance of relational databases. Discrimination networks consist of networks nodes and edges that constitute a directed acyclic graph (DAG). The network nodes implement condition tests and store all objects that satisfy the condition tests by means of an internal memory. The network edges forward objects that satisfy these conditions to successor network nodes. Network nodes with one input implement *intra-element tests* such as attribute or type constraints. Network nodes with two or more inputs implement *inter-element tests*, because they combine the condition test results of predecessor network nodes, e. g., in terms of join conditions.

When objects in the data set change, these changes are propagated through the network to update the objects that are stored by network nodes. When objects do not satisfy a condition test anymore, they are removed from the internal memory of the network node. When objects satisfy a condition test, they are added to the internal memory of the network node.

The network maintenance handles modifications of objects as the deletion and subsequent re-creation of these objects.



(a) Rete network (b) Treat network (c) Gator network

Figure 1.3.: Kinds of discrimination networks

The literature distinguishes Rete [32], Treat [73], and Gator networks [48]. Figure 1.3 depicts the differences of these networks for the same condition. The letters A to F denote intra-element tests. The conjunctions of the condition tests A to F denote inter-element tests.

The time and space complexity of these networks depends on their structure of network nodes and edges. The more network nodes exist the more intermediate condition test results must be stored and maintained, but also can be exploited to decrease the time that is required to maintain the state of the network. Thus, the space complexity is dominated by the number of network nodes. The more intermediate condition test results are processed by a network node, the higher is the time complexity of the network node. Thus, the time complexity of the network is dominated by the network node with the highest time complexity.

Rete Networks

Figure 1.3(a) depicts a Rete network [32]. Ordinary Rete networks are characterized by a left- or right-associative network structure [67] that consists of network nodes with at most two inputs. Thus, complex condition tests must be decomposed into multiple partial condition tests that are implemented by multiple subsequent network nodes.

The advantage of Rete networks is that they maintain a large amount of partial condition test results as collections of objects and instantly provide objects that satisfy these condition tests for further processing. Furthermore, only those partial test results are added to or deleted from the internal memory of network nodes that result from object changes by exploiting already stored test results to minimize the effort to update the network.

However, the large amount of the stored partial test results is also the biggest disadvantage of Rete networks, because they result in a high memory footprint [73]. Furthermore, the operations to update Rete networks are similar, when objects are added and removed. Thus, deletions of objects are as expensive as additions of objects [73].

Treat Networks

Figure 1.3(b) depicts a Treat network [73]. Treat networks aim for overcoming the disadvantages of Rete networks. In contrast to Rete networks, Treat networks only consist of network nodes that perform intra-element tests. Treat networks do not employ network nodes with two or more inputs. Thus, Treat networks do not maintain results of inter-element tests.

Treat networks can outperform Rete networks, due to the fact that storing and maintaining the test results of inter-element tests in Rete networks is sometimes not beneficial [73]. On the one hand, Treat network require more effort to compute inter-element tests in comparison to Rete networks, because Treat networks do not maintain and store results of inter-element tests. On the other hand, Treat networks require less effort, when results of inter-element tests have to be removed in comparison to Rete networks, because the object changes do not have to be propagated to dependent network nodes that implement inter-element tests. The evaluation in [73] shows that the maintenance effort required by Treat networks can be lower than the maintenance effort required by Rete networks.

Gator Networks

Figure 1.3(c) depicts a Gator network [48]. Gator networks enable to create and maintain intermediate forms of discrimination networks by taking the advantages and disadvantages of Rete and Treat networks into account. Hanson et al. state that *"with Gator, it is possible to get additional advantages from optimization"* [48], because network nodes that compute and maintain results of inter-element tests *"are only materialized when they are beneficial"* [48]. As a result, Gator networks allow to control the trade-off between time and space complexity.

In contrast to Rete networks, Gator networks can consist of network nodes with more than two inputs to combine multiple inter-element tests in one network node. An internal evaluation plan describes the evaluation order of the combined inter-element tests. The evaluation plan is created, when the Gator network is constructed.

Furthermore, Gator networks can employ additional memory network nodes that store partial test results to steer the memory consumption of the network [48]. For the sake of simplicity, this thesis omits the description of these memory network nodes. Hanson et al. [48] provide a description of these memory network nodes.

In summary, Gator networks allow to specify intermediate forms of discrimination networks by supporting network nodes with more than two inputs in comparison to Rete and Treat networks. Thus, Gator networks are more general than Rete and Treat networks. The structure of Gator networks enables to describe Rete and Treat network structures as well. Rete and Treat network structures are the most extreme variants of the Gator network structure, because Rete networks decompose inter-element tests into multiple network nodes with at most two inputs, while Treat networks do not consider inter-element tests at all. It is the task of an optimization algorithm to compute an optimal network structure. Hanson et al. [48] propose different optimization algorithms based on cost functions that take statistics about update cardinalities and update frequencies of objects into account.

1.3.2. Discrimination Networks for Incremental Graph Pattern Matching

Incremental graph pattern matching bases on the concepts of discrimination networks. The following paragraphs describe which kinds of discrimination networks are adapted for incremental pattern matching.

Rete Networks for Graph Grammars

Bunke et al. [19] transfer the concepts of Rete networks to the efficient implementation of graph grammars for directed *labeled* graphs. According to Habel et al. [47], in a *"graph grammar approach it is defined how and under which conditions graph productions can be applied to a given graph in order to obtain a derived graph"*. The approach of Bunke et al. [19] derives the Rete network from the left-hand sides of graph productions, which describe the conditions under which a graph is derived from a given graph. Bunke et al. [19] adapt network nodes of Rete networks. In general, they distinguish node, edge, and subgraph checkers. Node and edge checkers employ intra-element tests. Subgraph checkers employ inter-element tests.

Node checkers receive graph nodes from the root of the network and check whether these nodes have a certain label. Edge checkers receive graph edges from the root of the network and check whether these edges have a certain label, the source nodes of these edges have a certain label, and the target nodes of these edges have a certain label. Subgraph checkers have two inputs and receive subgraphs from different edge checkers. Subgraph checkers combine two subgraphs to larger subgraphs, when both subgraphs have graph nodes in common. Subgraph checkers define in terms of a list which graph nodes must be shared by subgraphs.

Rete Networks for Model Transformations

Bergmann et al. [9] extend the approach of Bunke et al. [19] by transferring the concepts of Rete networks from graph grammars to model transformations of the MDE domain. Model transformations specify the applicability of model manipulations by means of graph patterns that must be satisfied by subgraphs of the model in order to manipulate models. Bergmann et al. [9] describe an incremental graph pattern matching engine that stores graph pattern matches and maintains these matches, when models change. This incremental graph pattern matching engine adapts Rete networks [32] to enable the transformation language of the VIATRA2 framework to make use of matches that are maintained within a Rete network. The approach derives Rete networks automatically from the graph patterns of model transformations based upon a heuristic. Then, matches can be retrieved *"in constant time excluding the linear cost induced by the size of the result set itself"* [9]. Bergmann et al. [9] state that *"the main ideas behind the incremental pattern matcher are conceptually similar to relational algebra"*. In the approach of Bergmann et al. [9], the internally employed Rete network represents information in terms of tuples that consist of model elements. Each network node in the Rete network is related to a (partial) graph pattern and stores a set of tuples that satisfy this (partial) pattern. Bergmann et al. [9] state that *"this set of tuples is in analogy with the relation concept of relational algebra"* and, thus, the authors map incremental graph pattern matching as challenge of the graph domain back to the relational domain.

The approach of Bergmann et al. [9] adds additional kinds of network nodes to Rete networks to make Rete networks applicable to graph pattern matching. Bergmann et al. [9] state that *"miscellaneous input nodes represent containment, generic type information, and other relationship between model elements"* and, therefore, multiple kinds of network nodes for intra-element tests seem to exist without explicitly discussing these additional network nodes. Furthermore, Rete networks in VIATRA2 distinguish network nodes for inter-element tests such as join nodes, negative nodes, and term evaluator nodes that consist of two inputs. Join nodes implement natural joins for tuples received by the first and second input. These joins are performed by means of an effective index structure that enables to check whether tuples of the first input can be joined with tuples of the second input. Negative nodes implement anti-joins for tuples received by the first and second input. Negative nodes store tuples received by the first input, that cannot be joined with any tuple received by the second input. Term evaluator nodes implement attribute conditions such as arithmetical and logical functions.

Production nodes store complete graph pattern matches and employ projections to filter out elements from tuples that are not required. These production nodes are used to implement disjunctions of two patterns. Then, each graph pattern is matched by a separate Rete network that have a production node in common, which employs a true union operation to store received graph pattern matches. Furthermore, production nodes are used to implement recursive definitions of graph patterns.

The approach of Bergmann et al. [9] is adapted for incremental model queries [7], live model transformations [80], derivation of model features [81] and model synchronization [27].

1.3.3. Discussion

In general, graph pattern matching is NP-complete for subgraph isomorphism [36]. When the graph pattern size k is fixed, the worst-case complexity is $O(n^k)$ for a graph with size n [58, 102]. Therefore, finding all subgraphs in base graphs that match a certain graph pattern can be very time-consuming, when graph queries and / or base graphs become large. One possibility to increase the performance of graph queries is to enumerate graph pattern matches as answers for these queries by means of graph views. Thus, algorithms are required that ensure the consistency of graph views with their base graphs.

The domain of relational view maintenance employs discrimination networks to maintain views. The disadvantages of Rete [32] and Treat networks [73] are addressed by Gator networks [48]. In contrast to Rete and Treat networks, Gator networks enable to employ network nodes with more than two inputs. Therefore, Gator networks allow to optimize the network structure concerning time and space complexity [48].

Bunke et al. [19] transfer Rete networks to the efficient implementation of graph grammars for directed labeled graphs. Bergmann et al. [9] transfer Rete networks to incremental graph pattern matching for models as special kind of graph based upon the original work of Bunke et al. [19]. Bunke et al. [19] and Bergmann et al. [9] extend Rete networks by additional network nodes to make Rete networks applicable for incremental graph pattern matching.

The following paragraphs derive observations from the presented state of the art that must be considered, when aiming for a generic incremental maintenance of graph views.

No Gator Networks for Incremental Graph Pattern Matching

Currently, Gator networks as most general kind of discrimination network are not transferred from the relational domain to the graph domain, although Gator networks can outperform Rete networks and Treat networks in time and space at the same time.

Missing Native Concept for Graph Views

Current adaptations of Rete networks for incremental pattern matchings suffer similar problems as known from the relational domain such as expensive join operations to reconstruct graph edges between graph nodes, because current approaches do not employ graph data models for storing graph pattern matches natively.

Missing Native Maintenance

Current approaches transfer Rete networks to the graph domain by mapping graph operations back to relational operations such as join operations. This issue becomes apparent due to special network nodes that are added to Rete networks to support special operations such as negations and term evaluations. Furthermore, tuples of the relational domain are used to store graph pattern matches. Thus, current approaches for incremental graph pattern matching do not implement Rete network natively by means of graph operations.

No Real View Maintenance

Rete networks map modifications of objects to sequences of deletion and creation operations, when maintaining the state of the network. Thus, Rete networks do not employ a real view maintenance for modified objects, because the derived tuples change their identity during the maintenance procedure, when they are deleted first and re-created afterwards.

Inefficient Maintenance for Object Deletions

In discrimination networks the deletion of objects is maintained in the same way as the creation of objects. Thus, the deletion of objects is an expensive operation [96] and, thus, compromises the maintenance performance.

Restriction to Models
Currently, incremental graph pattern matching by means of discrimination networks is limited to models as special kind of graph. Instead, discrimination networks should be employed for general kinds of graphs in a manner that the concepts for storing graph pattern matches and maintaining these graph pattern matches work for different kinds of graphs.

Heuristic-Based Derivation of Rete networks
Current approaches for incremental graph pattern matching employ heuristics to derive Rete networks from the left-hand side of model transformations. Thus, these approaches do not enable to model the discrimination network structure to control the trade-off between memory consumption and execution time, when maintaining the state of the network.

1.4. Contribution

This thesis employs Gator networks as technique for incremental graph pattern matching, when maintaining graph views. It is worth to employ Gator networks for the definition and incremental maintenance of graph views, because current approaches for incremental graph pattern matching are limited to Rete networks. The contributions of this thesis are as follows.

Gator Network Structures for View Maintenance
This thesis contributes a modeling language for the definition and incremental maintenance of graph views. The modeling language enables users to model generalized discrimination networks with network nodes that enable to embed queries. The modeling language support Gator network structures [48] as most generalized kind of discrimination network. Thus, the modeling language also supports Treat and Rete network structures. The modeling of generalized discrimination network structures enable developers to control the trade-off between time and space complexity. The modeling language enables to combine definitions for views by means of conjunctions, disjunctions, negations, and recursions. Especially, the recursions are an extension of discrimination networks, because the original discrimination networks are acyclic. The recursions enable developers to formulate path expressions and enable views to enumerate matches for these path expressions.

Notion of Native Graph Views
This thesis contributes the notion of graph views by means of a graph data model that enables to mark graph nodes that belong to graph pattern matches. Therefore, graph views are themselves graphs that can be processed by means of graph operations. These graph views do not store copies of graph nodes and edges that must be kept synchronized with base graphs.

Native Maintenance of Graph Views
This thesis contributes a maintenance algorithm for graph views that is designed for the use with graph data and does *not* map incremental graph pattern matching back to the relational domain as existing approaches do. In contrast to existing approaches for incremental graph pattern matching, the approach employs general-purpose network nodes that are not dedicated to a specific purpose such as negation or term evaluation. Furthermore, the maintenance algorithm performs an impact analysis based upon modifications of base graphs to prune search spaces for graph queries, when graph views have to be maintained. This impact analysis enables to efficiently lookup graph pattern matches that are obsolete, suspicious, or missing

in graph views. Furthermore, the maintenance algorithm employs native graph operations instead of relational operations and performs a real view maintenance, because the algorithm extends the maintenance algorithms of discrimination networks by an explicit maintenance phase for modified objects to avoid the deletion and re-creation of condition test results.

Realization for Evaluation
This thesis contributes an implementation of the developed concepts for an evaluation of these concepts. The realization demonstrates the concepts by means of case studies.

Evaluation of the Modeling Language and Maintenance Algorithms
This thesis contributes a modeling and performance evaluation of the developed concepts. The modeling evaluation demonstrates the effectiveness of the proposed modeling language and shows that a graph data model for storing graph pattern matches reduces the modeling effort for graph queries, which define the kinds of graph pattern matches that are stored by graph views. The performance evaluation shows that the employed Gator network structures can outperform equivalent Rete network structures for incremental view maintenance in time and space at the same time. Furthermore, the performance evaluation shows that the proposed approach for incremental view maintenance can outperform existing approaches for incremental graph pattern matchings that base on generated Rete network structures.

Beyond this Thesis
This thesis aims for a modeling approach that enables users to model discrimination network structures manually to control the trade-off between time and space complexity. An automatic optimization of discrimination network structures is not in the scope of this thesis.

This thesis aims for an efficient and scalable incremental maintenance of graph views. A parallel execution of discrimination networks is left for future work.

This thesis does not consider the equivalence of stated queries with the queries for which views enumerate matches. Thus, this thesis does not relate stated queries to graph views that store matches for these queries.

1.5. Outline

This thesis is structured as follows. Chapter 2 describes the foundations of this thesis. Chapter 3 introduces the running example that is used throughout this thesis. Chapter 4 describes the requirements that are addressed in this thesis and gives an overview of the concepts that are presented in this thesis. Chapter 5 describes employed graph data models for base graphs and graph views. Chapter 6 presents the modeling language for the definition of graph views by means of discrimination networks. Chapter 7 describes the transformation of graph views to create and maintain views. Chapter 8 describes batch and maintenance algorithms for the maintenance of graph views. Chapter 9 optimizes these maintenance algorithms to handle duplicates in graph views efficiently. Chapter 10 compares the proposed approach with related approaches for the definition of graph views. Chapter 11 compares the proposed approach with related approaches concerning the memory consumption of graph views and the execution time of the maintenance algorithms. Chapter 12 describes related work and compares it with the proposed approach. Chapter 13 concludes this thesis and outlines the horizon of possible future work.

2. Foundations

This chapter introduces the conceptual foundations on which the remainder of this thesis builds. Section 2.1 describes the notion of graphs and graph transformations. Section 2.2 describes the notion of models and metamodels as special kinds of graphs.

2.1. Graphs and Graph Transformations

This section explains the concepts of graphs and graph transformations. This thesis describes the developed concepts by using the example of directed typed attributed graphs. Note that the developed concepts can be adapted for other kinds of graph as well, e. g. hypergraphs or nested graphs [2]. Section 2.1.1 describes graphs as means to store entities and relationships between these entities. Section 2.1.2 describes graph transformations as means to manipulate entities and their relationships. The following definitions are adapted from [16, 38, 41, 66].

2.1.1. Graphs

Graphs are general-purpose data structures that store entities and relationships between these entities. According to Definition 1, graphs consist of graph nodes and graph edges. Graph nodes represent entities. Graph edges represent relationships between entities. Definition 1 describes graph edges as directed relationships between graph nodes. Two graph nodes are adjacent, when a graph edge connects both graph nodes.

Definition 1 (Graph)
A graph $G = (N_G, E_G, s_G, t_G)$ consists of

- *a set N_G of graph nodes,*
- *a set E_G of graph edges,*
- *a source function $s_G : E_G \to N_G$, which returns the source node of an edge, and*
- *a target function $t_G : E_G \to N_G$, which returns the target node of an edge.*

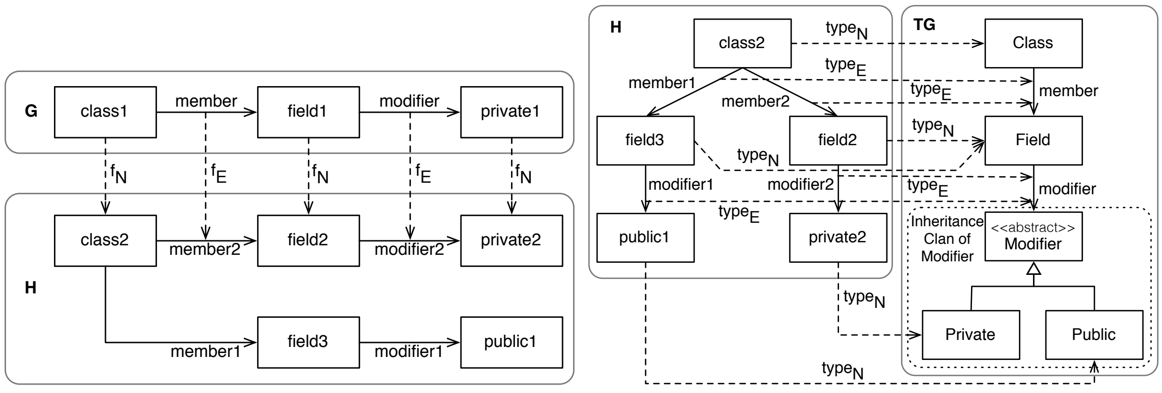
This chapter employs graphs that describe concepts of object-oriented languages as running example. For example, the top of Figure 2.1(a) depicts a graph G that consists of the graph nodes class1, field1, and private1 as well as the graph edges member and modifier. These graph nodes and edges describe that a class owns a private field.

Definition 2 (Graph Morphism)
Given a graph $G = (N_G, E_G, s_G, t_G)$ and $H = (N_H, E_H, s_H, t_H)$, a graph morphism $f : G \to H$ from graph G to H is a pair of functions (f_N, f_E) with $f_N : N_G \to N_H$ and $f_E : E_G \to E_H$, which map nodes and edges of G to nodes and edges of H. The following properties must hold:

- $\forall e \in E_G : f_N(s_G(e)) = s_H(f_E(e))$ *and*
- $\forall e \in E_G : f_N(t_G(e)) = t_H(f_E(e))$

According to Definition 2, a graph G exists in a graph H, if a structure-preserving mapping from the graph nodes and edges of graph G to the graph nodes and edges of graph H exists. That means, all adjacent nodes of graph G must be mapped to adjacent nodes of graph H. Note, graph H may contain graph nodes and edges that are not covered by this mapping and multiple graph nodes of graph G may be mapped to the same graph node of graph H.

For example, Figure 2.1(a) depicts that graph G exists in graph H as denoted by the dashed lines that are labeled with the function names f_N and f_E. For example, the function f_N maps the class1 graph node of graph G to the class2 graph node of graph H and the function f_E maps the member graph edge of graph G to the member2 graph edge of graph H.



(a) Simple graphs and graph morphism

(b) Typed graph

Figure 2.1.: Graph and graph morphism

Moreover, graph nodes and edges can consist of attributes that store additional properties. For the sake of simplicity, this chapter omits attributes of graph nodes and edges, because the formal definitions for these kinds of graphs are complex. Lara et al. [66] formalize attributes of graph nodes and edges as E-graphs. E-graphs are graphs that additionally consist of data nodes that represent attributes of graph nodes and edges. These data nodes are connected to graph nodes and edges by means of node attribute edges and edge attribute edges. Appendix A provides the formal definitions for E-graphs and beyond.

This thesis employs (attributed) typed graphs with type inheritance. Definition 3 introduces the notion of type graphs that describe types of graph nodes and edges that can exist in graphs. According to Definition 4, these types of graph nodes constitute a type hierarchy.

Definition 3 (Type Graph with Inheritance)
A type graph with inheritance $TG = (T, I, A)$ consists of

- *a graph $T = (N_T, E_T, s_T, t_T)$,*
- *an inheritance relation $I \subseteq N_T \times N_T$, and*
- *a set $A \subseteq N_T$ of abstract nodes.*

The inheritance relation is

- *reflexive: $(n, n) \in I$*
- *anti-symmetric: $(m, n) \in I \wedge (n, m) \in I \Rightarrow m = n$, and*
- *transitive: $(m, n) \in I \wedge (n, o) \in I \Rightarrow (m, o) \in I$*

Definition 4 (Inheritance Clan)
Given a type graph with inheritance $TG = (T, I, A)$ and $T = (N_T, E_T, s_T, t_T)$, an inheritance clan is defined by $clan_I(n) = \{m | m, n \in N_T \wedge (m, n) \in I\}$. If $m \in clan_I(n)$, than m inherits from n and this thesis writes $m < n$ as shorthand notation.

For example, Figure 2.1(b) depicts the graph H that is typed over graph TG. The type graph TG describes that classes consist of fields. Furthermore, fields consist of modifiers such as private and public visibilities. The dotted rectangle denotes the inheritance clan of the Modifier graph node that contains the graph nodes Modifier, Private, and Public. Furthermore, the Modifier graph node belongs to the set A of abstract graph nodes.

According to Definition 5, a graph is a typed graph, when each graph node and edge has a type. These types are defined by means of graph nodes and edges of a type graph.

Definition 5 (Typed Graph)
Given a graph $G = (N_G, E_G, s_G, t_G)$, a type graph $TG = (T, I, A)$ with $T = (N_T, E_T, s_T, t_T)$, and a graph morphism $type : G \to TG$, which consists of two functions $type_N : N_G \to N_T$ and $type_E : E_G \to E_T$ that assign graph nodes and edges of T to the graph nodes and edges of G, then graph G is a typed graph over TG, if the following conditions hold:

- $\forall e \in E_G : type_N(s_G(e)) < s_T(type_E(e))$ *and*
- $\forall e \in E_G : type_N(t_G(e)) < t_T(type_E(e))$

For example, Figure 2.1(b) shows dashed lines that are labeled with the function names $type_N$ and $type_E$. For example, the function $type_N$ assigns the Class node type of the type graph TG to the class graph node of graph H and the function $type_E$ assigns the member edge type of the type graph TG to the member1 and member2 graph edges of graph H.

Definition 6 extends the Definition 2 of graph morphisms in a manner that types of mapped graph nodes and edges must be compatible as well.

Definition 6 (Typed Graph Morphism)
Given two graph $G = (N_G, E_G, s_G, t_G)$ and $H = (N_H, E_H, s_H, t_H)$ over a type graph TG, a pair of functions (f_N, f_E) with $f_N : N_G \to N_H$ and $f_E : E_G \to E_H$ is a typed graph morphism, if the following conditions hold:

- $\forall e \in E_G : f_N(s_G(e)) = s_H(f_E(e)) \wedge f_N(t_G(e)) = t_H(f_E(e))$ *(structure compatibility)*
- $\forall n \in N_G : type_N(f_N(n)) < type_N(n)$ *(type compatibility)*
- $\forall e \in N_E : type_E(f_E(e)) = type_E(e)$ *(type compatibility)*

For example, Figure 2.2 shows the graphs G and H that are typed over graph TG. The graph G exists in graph H as denoted by the dashed lines that are labeled with the function name f_N. Additionally, the mapped graph nodes and edges have the same types as denoted by the dashed lines that are labeled with the function name $type_N$. For example, the function f_N maps the class1 graph node of graph G to the class2 graph node of graph H and the function $type_N$ describes that both graph nodes are of the same Class node type. For readability, Figure 2.2 neglects the functions f_E and $type_E$.

2.1.2. Graph Transformations

Graph transformations consists of graph transformation rules that modify graphs, when certain conditions are satisfied. These conditions are defined by means of graph patterns. The modifications of graphs are defined by rule applications.

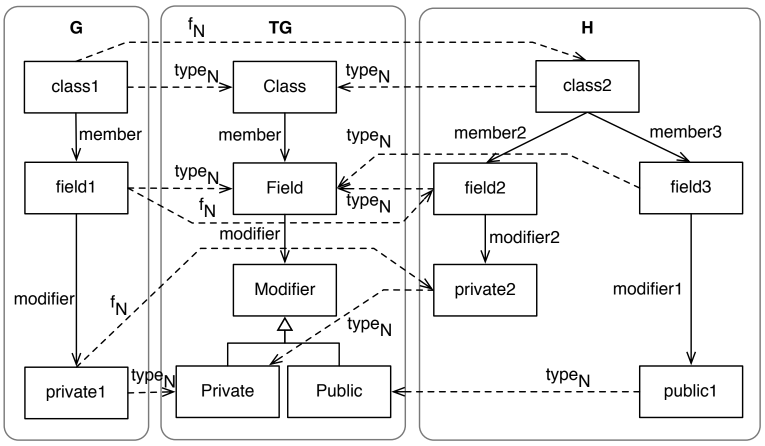


Figure 2.2.: Typed graph morphism

Definition 7 defines a graph pattern as a typed graph P for which copies in another graph G may exist and a set of typed graphs $\{N_j, j \in J\}$ that contain P as subgraph and for which copies must not exist in graph G.

Definition 7 (Graph Pattern)
A graph pattern $\Pi = (P, \{N_j, j \in J\})$ consists of a typed graph P and a finite set of typed graphs N_j that contain P as subgraph. For the graph pattern $(P, \emptyset)$, this thesis writes P.

This thesis refers to the graph nodes and edges of graph P and N_j as pattern nodes and pattern edges, respectively. This thesis refers to the graph P as positive application condition (PAC) and to N_j as negative application condition (NAC).

For example, Figure 2.3 shows four graph patterns. The examples neglect the type graph. Instead, the graph nodes are labeled with the name and type of the graph node separated by a colon. The examples omit the types of graph edges. The rounded rectangles denote the PACs and NACs. The graph pattern in Figure 2.3(a) searches for classes with private fields. The graph pattern in Figure 2.3(b) searches for classes with fields that are *not* private. The graph pattern in Figure 2.3(c) searches for classes that have *no* private fields.

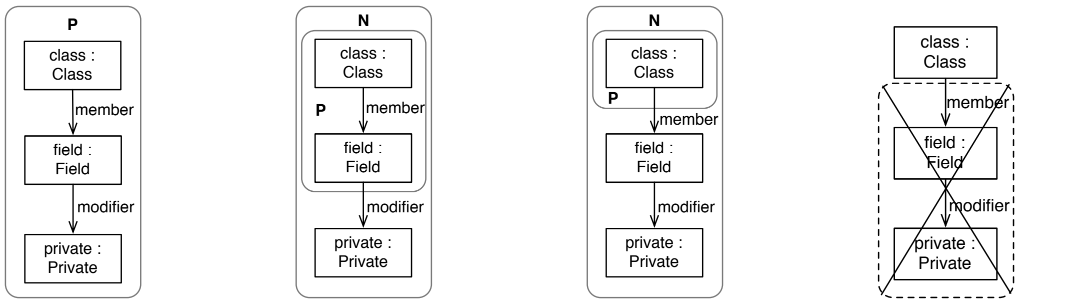
As a shorthand notation, this thesis crosses out graph nodes and edges of NACs that do not belong to PACs. Figure 2.3(d) depicts the pattern of Figure 2.3(c) in shorthand notation.

According to Definition 8, this thesis distinguishes simple and complex NACs.

Definition 8 (Simple and Complex NAC)
Given a graph pattern $\Pi = (P, \{N_j, j \in J\})$, N_j is a simple NAC, if all graph nodes of $N_j \setminus P$ are directly connected to graph nodes of subgraph P and no graph edges exist that connect the graph nodes of $N_j \setminus P$. N_j is a complex NAC, if at least one graph node of $N_j \setminus P$ is not directly connected to graph nodes of subgraph P or graph edges exist that connect the graph nodes of $N_j \setminus P$.

Figure 2.3(b) shows a simple NAC, because all graph nodes of the NAC $N \setminus P$ are connected to the graph nodes of the PAC P and the graph nodes of the NAC $N \setminus P$ own no additional graph edges that connect graph nodes of the NAC $N \setminus P$. Figure 2.3(c) shows a complex NAC, because the private graph node is *not* directly connected to a graph node of PAC P.

According to Definition 9, copies of a sample graph P in another graph G are called graph pattern match, when no copies of N_j containing P exist in graph G.



(a) PAC (b) Simple NAC (c) Complex NAC (d) Shorthand notation

Figure 2.3.: Sample graph patterns

Definition 9 (Graph Pattern Match)
Given a graph pattern $\Pi = (P, \{N_j, j \in J\})$ *and a graph* G*, then each injective typed graph morphism* $m : P \rightarrow G$ *such that there does not exist an injective typed graph morphism* $q : N_j \rightarrow G$ *with* q *being identical to* m *on* P*, is called a graph pattern match of the graph pattern* Π *in* G*. This thesis refers to a graph pattern match as match.*

For example, Figure 2.4 shows the two graph patterns P_1 and P_2 that have matches in graph H as denoted by the dashed lines with f_N and f_E label. The graph pattern P_1 (cf. Figure 2.3(a)) matches the graph nodes class2, field2, and private2 of graph H. The graph pattern P_2 (cf. Figure 2.3(b)) matches the graph nodes class2 and field3 of graph H.

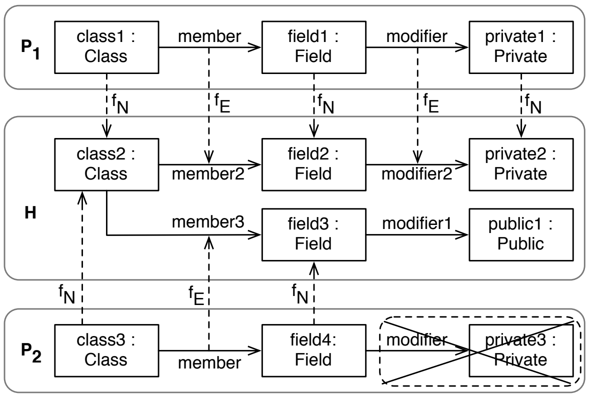
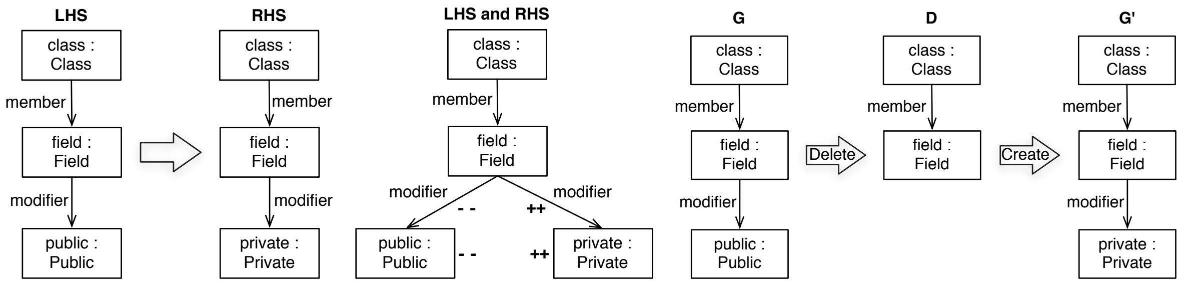


Figure 2.4.: Graph pattern match

A graph transformation rule describes how and under which conditions graphs are modified by adding or removing graph nodes and edges. A graph transformation rule consists of a left-hand side (LHS) graph pattern Π_{LHS} and right-hand side (RHS) graph pattern Π_{RHS}. The left-hand side graph pattern describes the pre-conditions under which the graph transformation rule modifies a graph. The right-hand side graph pattern describes the post-conditions that must be satisfied by a graph, after the graph transformation rule is applied.

Definition 10 (Graph Transformation Rule)
A graph transformation rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *consists a left-hand side* $\Pi_{LHS} = (L, \{N_j, j \in J\})$ *and right-hand side* $\Pi_{RHS} = R$ *graph pattern. The set* $del(r) = L \setminus (L \cap R)$ *is the part of the graph* G *that is deleted by the rule* r*. The set* $cre(r) = R \setminus (L \cap R)$ *is the part of the graph* G *that is created by the rule* r*.*

For example, Figure 2.5(a) shows the left-hand side and right-hand side graph pattern of a graph transformation rule. This rule transforms all public fields of a class into private fields. Figure 2.5(b) shows the shorthand notation of this rule. The shorthand notation combines the left-hand side and right-hand side graph pattern in one graph pattern that consists of graph node and edge modifiers. These modifiers describe which graph node and edge is deleted, created, or preserved, when the graph transformation rule is applied. The shorthand notation labels graph nodes and edges that are deleted or created with “--” and “++”, respectively. The shorthand notation does not label graph nodes and edges that are preserved, when the graph transformation is applied. Thus, the notation depicts the left-hand side pattern as graph nodes and edges with “--” labels and without labels. The notation depicts the right-hand side pattern as graph nodes and edges with “++” labels and without labels.



(a) Transformation rule (b) Shorthand notation (c) Rule application

Figure 2.5.: Graph transformation rule and application

A graph transformation rule is applicable, if a match for the left-hand side of the graph transformation rule exists. Definition 11 distinguishes the single-pushout (SPO) and the double-pushout (DPO) approach. The SPO approach permits implicit side-effects. When the right-hand side of the graph transformation rule deletes graph nodes, some graph edges may be dangling. The SPO approach deletes these dangling graph edges too, even if the graph transformation rule does not describe this deletion explicitly. The DPO approach does not permit implicit side-effects and does not apply the graph transformation rule, even if a match for the left-hand side of the graph transformation rule exists.

Definition 11 (Graph Transformation Rule Applicability)
A graph transformation rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *with* $\Pi_{LHS} = (L, \{N_j, j \in J\})$ *and* $\Pi_{RHS} = R$ *is applicable to a graph* G *in the double-pushout approach, if there exists a match* $m : L \to G$ *for* Π_{LHS} *in* G *that fulfills the dangling edge condition.*

The dangling edge condition $dan(m, r) = \{e | e \in E_G, s_G(e) \vee t_G(e) \in m(del(r)), e \notin m(del(r))\}$ *is the set of dangling edges in* G *for match* m *and rule* r*. The match* m *fulfills the dangling edge condition for rule* r*, if* $dan(m, r)$ *is empty.*

A graph transformation rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *with* $\Pi_{LHS} = (L, \{N_j, j \in J\})$ *and* $\Pi_{RHS} = R$ *is applicable to a graph* G *in the single-pushout approach, if there exists a match* $m : L \to G$ *for* Π_{LHS} *in* G*.*

When a match for the left-hand side of the graph transformation rule exists, the graph transformation rule is applied for this match. Then, the result of the rule application must satisfy the right-hand side of the graph transformation rule. According to Definition 12, this rule application consists of two construction steps. First, the rule application deletes all graph nodes and edges (incl. dangling graph edges) that are not preserved by the right-hand side of the graph transformation rule. The result is an intermediate graph. Second, the rule application creates graph nodes and edges in the graph by means of a gluing construction that copies the graph nodes that are created by the right-hand side of the graph transformation rule into the intermediate graph.

Definition 12 (Graph Transformation Rule Application)
A rule application $G \overset{r,m}{\Rightarrow} H$ from graph G to graph H by means of an applicable rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ with $\Pi_{LHS} = (L, \{N_j, j \in J\})$ and $\Pi_{RHS} = R$ and a match $m : L \to G$ is constructed as follows.

- *Delete graph nodes and edges incl. dangling graph edges: $D = G \setminus (m(del(r)) \cup dan(m, r))$*
- *Create graph nodes and edges: $H = D \cup i(R)$ with $i : R \to i(R)$ a graph isomorphism identical to m on elements of $L \cap R$ and disjoint with D on elements in $cre(r)$.*

For example, Figure 2.5(c) shows how the graph transformation rule of Figure 2.5(b) transforms graph G via the intermediate graph D into graph G'. First, this rule removes the public graph node from graph G and derives the intermediate graph D. Second, this rule creates the private graph node and links it to the field graph node by creating a modifier graph edge. The result is the derived graph G'.

The Definition 13 describes a graph transformation as a sequence of multiple graph transformation rule applications that transform a graph into another graph.

Definition 13 (Graph Transformation)
A graph transformation $G_0 \overset{*}{\Rightarrow} G_n$ is a sequence $G_0 \Rightarrow \ldots \Rightarrow G_n$ with $n \geq 0$ rule applications.

According to Definition 14, a graph pattern is a graph condition that can consist of other graph conditions by means of conjunctions, disjunctions, and negations.

Definition 14 (Graph Conditions)
A graph pattern $\Pi = (P, \{N_j, j \in J\})$ is a graph condition. Any combination of two graph conditions p and q of the form $p \wedge q$, $p \vee q$, and $\neg q$ is also a graph condition. A graph condition p satisfies a graph G (written $G \models p$), if

- *a match exists for graph pattern p in G,*
- *$p = p_1 \wedge p_2$, then p is satisfied, if $G \models p_1$ and $G \models p_2$*
- *$p = p_1 \vee p_2$, then p is satisfied, if $G \models p_1$ or $G \models p_2$*
- *$p = \neg p_1$, then p is satisfied, if $\neg(G \models p_1)$.*

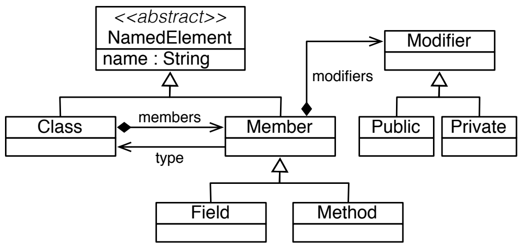
Definition 15 (Recursive Graph Conditions)
A graph condition p is a recursive graph condition, if it refers to graph condition p, e. g., $p = p \wedge q$ or $p = p \vee q$.

Definition 16 (Graph Query)
A graph query is a (recursive) graph condition. The query results are graph pattern matches.

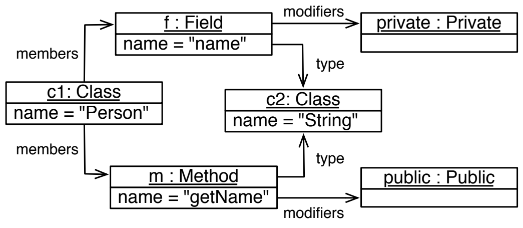
2.2. Models and Metamodels

In software engineering, models and metamodels are special kinds of graphs. A *metamodel* is an (attributed) type graph that defines additional properties such as graph edge cardinalities and containment hierarchies. A metamodel describes the concepts of a modeling language.

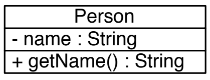
A *model* is a typed (attributed) graph that must adhere to a metamodel, i. e., a model is an instance of a metamodel. A model is expressed either in abstract syntax or concrete syntax. The *abstract syntax* shows all graph nodes and edges that belong to the model. The *concrete syntax* represents a model in a textual or graphical notation and neglects details of the abstract syntax.



(a) Metamodel (UML class diagram)



(b) Abstract syntax graph (UML object diagram)



(c) Graphical Notation (UML class diagram)

```
class Person {
  private String name;
  public String getName {
  ...
  };
}
```

(d) Textual Notation (Java Syntax)

Figure 2.6.: Metamodel and models of a simple object-oriented language

For example, Figure 2.6(a) depicts a simple metamodel for an object-oriented language by means of an Unified Modeling Language (UML) class diagram. The metamodel describes that the language consists of classes, which own members such as fields and methods that have a type. Furthermore, classes and members have a name. Additionally, members own modifiers such as public and private visibilities, respectively. Figure 2.6(b) depicts an instance of this metamodel by means of an UML object diagram. The model describes a Person class that owns a private name field and a public getName method. The data type of the field and the return type of the method is the String class. Figure 2.6(c) borrows the notation of UML class diagrams to depict the model in concrete syntax. Figure 2.6(d) employs the textual syntax of the Java programming language to describe the model.

Depending on the graph pattern language, models in abstract or concrete syntax are used to describe graph patterns (cf. Definition 9). Additionally, these pattern languages provide the Object Constraint Language (OCL) [77] and OCL-like expression languages for the definition of attribute constraints such as constraints that check the equality of attribute values. These languages enable to describe arithmetic and logical expressions. These expressions must be satisfied by graph nodes and edges that constitute graph pattern matches.

3. Running Example

The running example, which is used throughout this thesis, deals with the detection of employed software design patterns and recommended software refactorings. Software design patterns describe best-practices for the design of software architectures. Gamma et al. [35] describe creational, structural, and behavioral design patterns. Software refactorings aim for improving the internal structure of existing source code. Fowler et al. [33] describe software refactorings as architectural design flaws in source code. They recommend how such design flaws should be refactored to improve the design of the source code while preserving the external behavior of the software. Fowler et al. [33] describe several software refactorings such as how to deal with generalizations, make method calls simpler, or simplify conditional expressions. This thesis refers to software refactorings as software design anti-patterns. This thesis uses the term design pattern as umbrella term for design pattern and anti-pattern.

This thesis uses the example of software design patterns and anti-patterns, because a) a large number of software repositories exists that provide large-scale source code together with a real history of changes and b) real graph queries can be derived from the descriptions of design patterns [33, 35]. The running example employs abstract syntax graphs (ASGs) of object-oriented source code as base graphs and searches for locations in these ASGs, where software design patterns are employed and software refactorings should be employed. The running example aims for storing and maintaining such locations by means of graph pattern matches (cf. Definition 9) in graph views. For example, when a design pattern appears in ASGs due to changes of the source code, the design pattern must be also added to the view. When a design pattern disappears from ASGs due to changes of the source code, the design pattern must be also removed from the view.

Detecting such design patterns in ASGs is a difficult task, because multiple variants of software design patterns and anti-patterns exist in practice, which have to be considered by the detection procedure. For that purpose, the running example employs graph conditions (cf. Definition 14) that support nested disjunctions, conjunctions, and negations to express graph patterns that cover these variants of the design patterns. Then, the running example employs graph pattern matching (cf. Definition 9) to find subgraphs in ASGs that satisfy these graph conditions and, therefore, represent matches of design patterns. The graph views store and maintain these graph pattern matches.

For the detection of design patterns, this thesis adapts the approach of Niere et al. [75, 76]. Approaches for the detection of design patterns that employ heuristics or metrics [70, 79] are not in the scope of the running example. The running example employs an approximation for detecting design patterns. That means, the running example does not aim for detecting the employed design patterns fully correct and complete in all possible implementation variants. Instead, the running example serves as means to describe the definition and maintenance of graph views. Appendix F provides a list of all employed graph patterns.

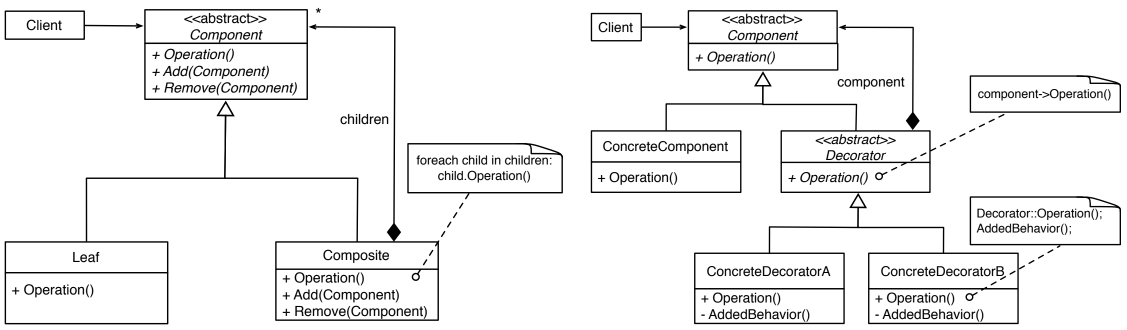
Section 3.1 describes examples for design patterns. Section 3.2 describes the composition of design patterns. Section 3.3 describes the used type graph and graph conditions for detecting design patterns. Section 3.4 summarizes this chapter.

3.1. Software Design Patterns and Anti-Patterns

This section describes examples for software design patterns and anti-patterns that are used for explanations in this thesis. Figure 3.1 shows the Composite and Decorator software design patterns as defined by Gamma et al. [35]. Figure 3.2 depicts the Extract Interface design anti-pattern as defined by Fowler et al. [33]. Both figures employ UML class diagrams.

Composite Design Pattern

Figure 3.1(a) shows the Composite design pattern that has the intention to *"compose objects into tree structures to represent part-whole hierarchies"* [35, p. 163]. Furthermore, the *"Composite lets clients treat individual objects and compositions of objects uniformly"* [35, p. 163]. The Composite design pattern consists of four parts called component, composite, leaf, and client. The component defines an interface and implements the default behavior for all objects in the composition. The composite describes the behavior of all objects in the composition such as other composites or leafs. For that purpose, the composite stores all children components as denoted by the children aggregation between the Composite class and Component class. The children aggregation describes the part-whole hierarchy of the composition. The composite and leaf are themselves specializations of the component as denoted by the inheritance relationship between the Component class and Composite class as well as the Component class and Leaf class. Leafs are objects in the composition that do not have children. Clients manipulate the tree-structure of the composition. For that purpose, composites enable to add and remove other composites and leafs to / from the tree structure. Furthermore, when clients call certain operations, composites propagate these operation calls to all children in the composition.



(a) UML class diagram of the Composite design pattern (adapted from [35, p. 164])

(b) UML class diagram of the Decorator design pattern (adapted from [35, p. 177])

Figure 3.1.: Running example for software design patterns

In practice, several variants of the Composite design pattern exist. For example, the Component class can be replaced by an interface that defines the methods that must be implemented by the Leaf and Composite class. Furthermore, also multi-level inheritance between classes in the Composite design pattern can be employed. Note that also interfaces and multi-level inheritance can be combined to implement variants of Composite design patterns. Furthermore, the Leaf class is an optional part of the Composite design pattern and can be missing. Moreover, Gamma et al. [35] describe that the children association between the Composite class and Component class has an unbounded multiplicity. However, in practice also aggregations with bounded multiplicity can be employed, when the number of children in the composites has an upper bound.

Decorator Design Pattern

Figure 3.1(b) shows the Decorator design pattern that has the intention to *"add additional responsibilities to an object dynamically"* [35, p. 175]. The concept of decorators is an alternative to the concept of subclasses, when the functionality of a class has to be extended. The Decorator design pattern consists of five parts called component, concrete component, decorator, concrete decorator, and client. The component defines a common interface for all objects that can be extended by additional functionality. The concrete component is a component that defines objects that can be extended with additional functionality by adding decorators. For that purpose, the ConcreteComponent class is a specialization of the Component class. The decorator is a component that implements the additional functionality that is added to concrete components. For that purpose, the Decorator class is a specialization of the Component class and the Decorator class owns a component association that targets the component that is extended by the decorator. Furthermore, several concrete decorators can exist that extend the default decorator. Concrete decorators add additional functionality to the default decorator by overriding inherited methods. Note that decorators are also components and can be extended by additional decorators, too.

As illustrated by Figure 3.1, the Composite and Decorator design patterns have structural properties in common such as the inheritance hierarchies between the Component class and Composite class as well as the Component class and Decorator class, respectively. Furthermore, variants of the Decorator design pattern exist that are similar to the variants of the Composite design pattern. Therefore, the Decorator design pattern can be considered as extension of the Composite design pattern. However, in comparison to the Composite design pattern the Decorator design pattern employs a component association with a bounded multiplicity of one, while the Composite design pattern employs a children association with an unbounded multiplicity. This difference must be considered, when reusing detected Composite design patterns for detecting Decorator design patterns.

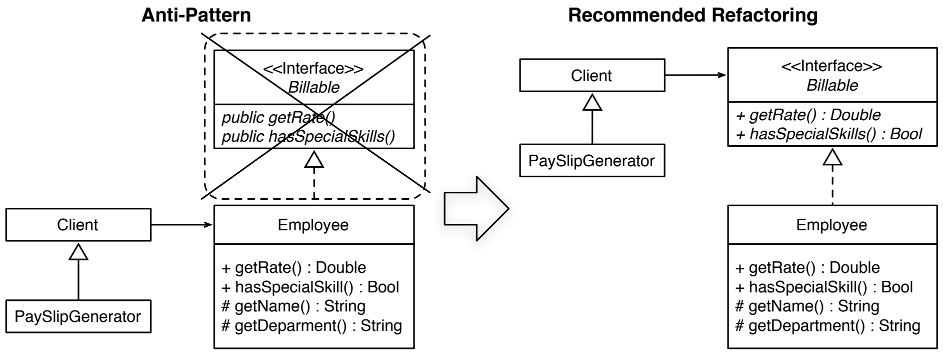


Figure 3.2.: UML class diagram of the Extract Interface design anti-pattern (cf. [33, p. 341])

Extract Interface Design Anti-Pattern

Figure 3.2 shows the Extract Interface software design anti-pattern that has the intention to derive a common interface, when *"several clients use the same subset of a class's interface, or two classes have parts of their interfaces in common"* [33, p. 341]. An explicit interface should be extracted to enable certain groups of clients to use a certain subset of the methods in a uniform way. The Extract Interface software design anti-pattern does not describe which methods should be extracted and how many interfaces should be derived. This decision is left to the person, who performs the software refactoring.

Figure 3.2 depicts an Employee class that consist of the public methods getRate and hasSpecialSkills among other methods. A pay slip generator uses the public getRate and hasSpecialSkills methods to compute the salary of employees. Therefore, the Billable interface should be extracted that defines the signature of the getRate and hasSpecialSkills methods. Then, the classes that implement the Billable interface can be processed by the pay slip generator or different kinds of pay slip generators in a uniform way.

3.2. Composition of Design Patterns and Anti-Patterns

Software design patterns and anti-patterns are high-level graph patterns that can be composed of lower-level graph patterns such as patterns for generalizations or associations. When graph pattern matches for high-level patterns have to be found, graph pattern matches for lower-level patterns can be looked up first and, afterwards, these matches can be reused to lookup matches for high-level patterns. Thus, graph views that store matches of high-level patterns can reuse the matches of lower-level patterns that are stored by other views.

Figure 3.3 shows kinds of graph patterns by means of a type graph (cf. Definition 3) for design patterns. The type graph depicts specializations of graph patterns by means of a type hierarchy and which kinds of patterns reuse other kinds of patterns. Also other compositions are possible, but they are not in the scope of the running example. Figure 3.3 shows the decomposition of the Composite and Decorator design patterns as well as the Extract Interface design anti-pattern into lower-level patterns.

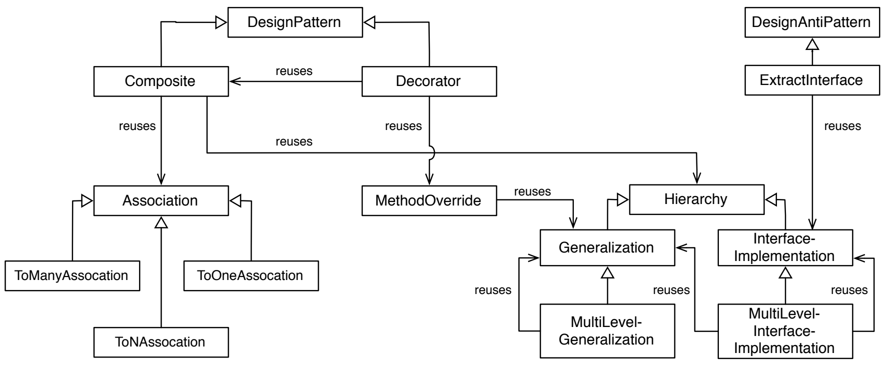


Figure 3.3.: UML class diagram that describes the decomposition of design (anti-)patterns

The Composite graph pattern refers to the Composite design pattern. The Composite pattern is a high-level pattern, because it reuses the Association and Hierarchy patterns. As described in Section 3.1 multiple variants of the Composite design pattern exist. The Composite pattern deals with these variants, because it reuses the Association and Hierarchy patterns that consider variants of associations, generalizations, and interface implementations.

The Association pattern is a low-level pattern. The ToOne Association, ToN Association, and ToMany Association patterns are specializations of the general Association pattern. The ToOne Association pattern refers to associations with a multiplicity of one. The ToN Association pattern refers to associations with a bounded multiplicity of N. The ToMany Association pattern refers to associations with an unbounded multiplicity.

The Hierarchy graph pattern refers to several kinds of hierarchies. This thesis distinguishes Generalization and Interface Implementation patterns as specializations of the general Hierarchy pattern. The Generalization pattern refers to classes that extend other classes without any intermediate classes in between. This thesis refers to this kind of generalizations as single-level generalizations. The Interface Implementation pattern refers to classes that implement an interface without any intermediate classes in between. This thesis refers to this kind of interface implementations as single-level interface implementations.

The Multi-Level Generalization pattern is a specialization of the Generalization pattern. The Multi-Level Generalization pattern refers to classes that extend other classes with at least one intermediate class in between. This thesis refers to this kind of generalizations as multi-level generalizations. The Multi-Level Generalization pattern reuses the Generalization patterns (incl. Multi-Level Generalization pattern), because multi-level generalizations are recursively composed of single-level and multi-level generalizations.

The Multi-Level Interface Implementation pattern is a specialization of the Interface Implementation pattern. The Multi-Level Interface Implementation pattern refers to classes that are subclasses of other classes that implement an interface. This thesis refers to this kind of interface implementations as multi-level interface implementations. Therefore, the Multi-Level Interface Implementation pattern reuses the Generalization patterns (incl. Multi-Level Generalization pattern) and the Interface Implementation pattern.

The Decorator graph pattern refers to the Decorator design pattern. The Decorator pattern is a high-level pattern, because it reuses the Composite pattern and Method Override pattern. The Method Override pattern refers to methods that are overridden in subclasses. Therefore, the Method Override pattern reuses the Generalization patterns.

The Decorator design pattern is an extension of the Composite design pattern as described in Section 3.1. Therefore, the Decorator pattern reuses the Composite pattern due to their structural similarities. The Decorator pattern considers variants of the Decorator design pattern, because the Composite pattern considers these variants already.

The Extract Interface pattern refers to the Extract Interface design anti-pattern. The Extract Interface pattern is a high-level pattern, because it reuses the Interface Implementation patterns.

3.3. Query Software Design Patterns and Anti-Patterns

This section describes which kinds of graph nodes and edges are stored by base graphs and which kinds of graph pattern matches are stored by graph views. Section 3.3.1 describes the type graph of base graphs. Afterwards, Section 3.3.2 describes graph patterns that are employed to query the base graphs for design patterns. Later on, these patterns are used to specify the content of graph views.

3.3.1. Type Graph of Base Graphs

Figure 3.4 depicts the type graph of base graphs by means of a metamodel (cf. Section 2.2). The metamodel describes general concepts of object-oriented languages such as classes, attributes and operations. The metamodel is derived from the Java Model Parser and Printer (JaMoPP) metamodel [53] that describes the language concepts of the Java programming language. For the sake of simplicity, the metamodel is an excerpt and simplified version of the JaMoPP metamodel and assumes that all references can be traversed in forward and backward direction. The metamodel is designed in a way that it is not dedicated to any object-oriented language.

The metamodel in Figure 3.4 distinguishes Types and TypeReferences. Types describe kinds of object-oriented concepts and, therefore, consist of several specializations. Types are either primitive types, e. g. integers and booleans, or classifiers, e. g. classes and interfaces. Furthermore,

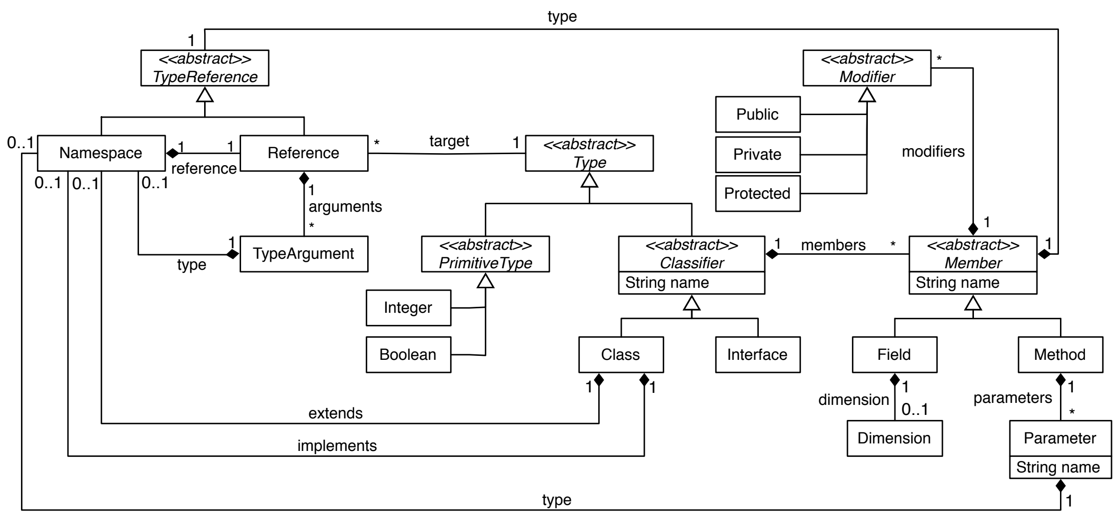


Figure 3.4.: UML class diagram that describes the type graph of base graphs as metamodel

classifiers consist of a name that describes the intention of the type. Type references enable other concepts to refer to types that are defined in the same or a different namespace.

The classifiers own members that describe attributes and operations of classifiers. Therefore, members have a name, type, and modifier. The member name describes the intention of the member. The member type enables to refer to primitive types or classifiers that describe the type of a field or the type of a return value of a method. The member modifier describes the visibility and accessibility of members. For example, the metamodel distinguishes public, private, and protected visibilities. Fields are specializations of members and describe attributes or associations of classes. Furthermore, fields can consist of a dimension that describes a bounded number of elements that are stored by a field. Methods are specializations of members and describe operations that are provided by classifiers. Methods can own parameters that consist of a name and type. Classes, members, and parameters refer to other types by means of type references. Classes refer to other types, when they extend another class or implement an interface. Members use type references to describe the type of fields and return values of methods. Parameters use type references to describe their types.

The metamodel distinguishes type references into namespaces and references. Namespaces enable to refer to the namespaces of types. References enable to refer to types in a certain namespace. Furthermore, references can consist of type arguments to describe parameterized types. For example, when a reference refers to a type that describes a data structure, a type argument is used to describe the type of the elements that are stored in the data structure.

3.3.2. Graph Patterns for Graph Views

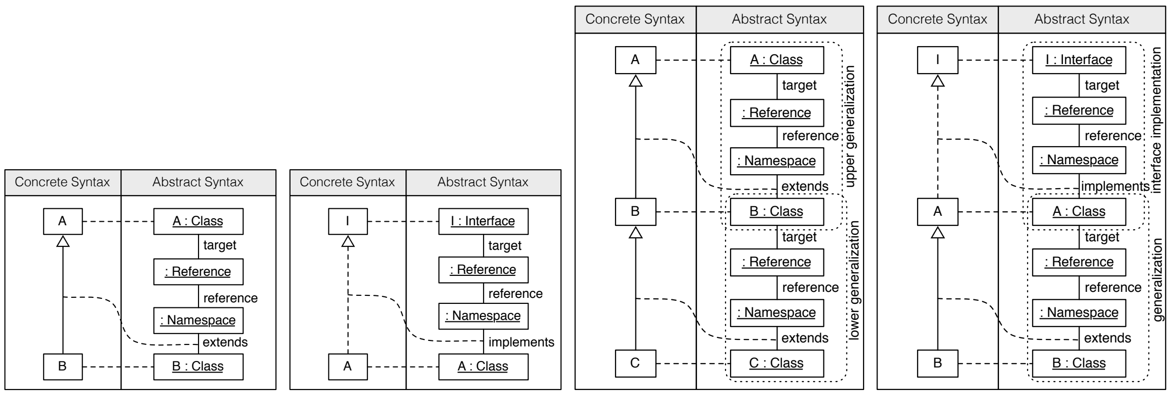
This section describes graph patterns that represent software design patterns and anti-patterns. First, this section describes low-level patterns. Afterwards, this section reuses these low-level patterns to compose high-level patterns. The graph patterns base on the type graph that is described in Section 3.3.1. The rest of this thesis, refers to these patterns to query base graphs for design patterns and to describe the kinds of graph pattern matches that are maintained by graph views. These patterns are approximations of design patterns, which rather aim for conveying the developed concepts than supporting an accurate and complete detection of design patterns. The following paragraphs describe the graph patterns in concrete and abstract syntax by means of UML class diagrams and UML object diagrams, respectively. Chapter 2 introduces the notation of the concrete and abstract syntax.

Low-Level Graph Patterns

The following sections describe low-level patterns such as patterns for querying hierarchies, associations, and overriding methods.

Hierarchy patterns

Figure 3.5 shows the Generalization, Multi-Level Generalization, Interface Implementation, and Multi-Level Interface Implementation patterns as variants of the general Hierarchy pattern (cf. Section 3.2). These variants can be composed by means of disjunctions in high-level graph patterns to handle the detection of implementation variants of design patterns, e. g., Composite design patterns that employ single-level or multi-level generalizations.



(a) Generalization (b) Interface Implementation (c) Multi-Level Generalization (d) Multi-Level Interface Implementation

Figure 3.5.: Kinds of Hierarchy patterns as running example

Figure 3.5(a) depicts the Generalization pattern, which describes a single-level generalization between a superclass A and a subclass B. In the abstract syntax, the subclass references the superclass by means of an extends edge that refers to the namespace that owns the superclass.

Figure 3.5(b) depicts the Interface Implementation pattern, which describes a class A that implements an interface I. In the abstract syntax, the Interface Implementation pattern is similar to the Generalization pattern, but differs in the implements edge, which references the namespace that contains the interface.

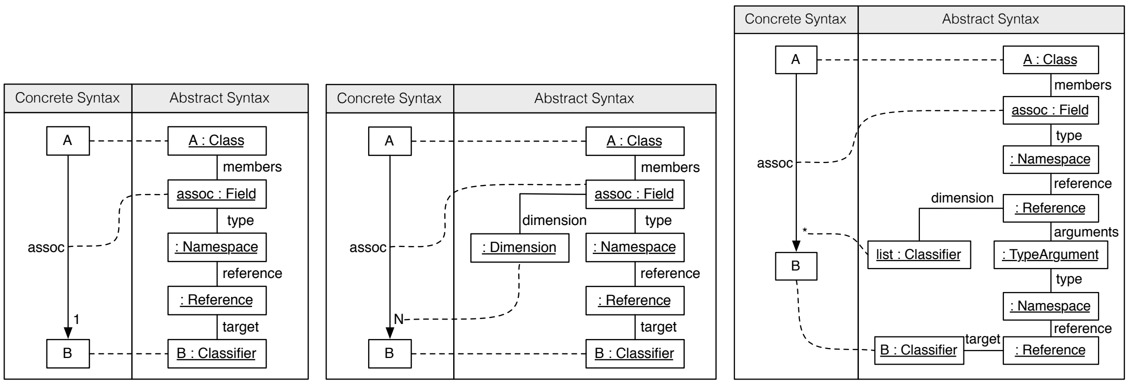
Figure 3.5(c) depicts the Multi-Level Generalization pattern, which describes a generalization with at least two inheritance levels between the outermost classes A and C. A multi-level generalization exists, when the subclass B of the upper generalization and the superclass B of the lower generalization are the same. The embedded Generalization patterns can be also replaced by the Multi-Level Generalization patterns with two or more inheritance levels. Then, the same condition must hold for the lower and upper multi-level generalization.

Figure 3.5(d) depicts the Multi-Level Interface Implementation pattern, which describes that class A implements an interface I and class B extends class A at the same time. A multi-level interface implementation exists, when a superclass A of a generalization implements an interface

I. Then, class B implements interface I, too. Note that the Generalization pattern can be also replaced by the Multi-Level Generalization pattern with two or more inheritance levels.

Association patterns

Figure 3.6 shows the ToOne Association, ToN Association, and ToMany Association patterns as variants of the general Association pattern (cf. Section 3.2). High-level graph patterns can compose these variants by means of disjunctions to handle implementation variants of design patterns that employ different kinds of associations. For example, the Composite design pattern can employ associations with bounded or unbounded multiplicity.



(a) ToOne Association (b) ToN Association (c) ToMany Association

Figure 3.6.: Kinds of Association patterns as running example

Figure 3.6(a) depicts the ToOne Association pattern, which describes an association with a multiplicity of one. A class owns such an association, when it consists of a field that refers to a namespace, which contains the classifier that describes the type of the field.

Figure 3.6(b) depicts the ToN Association pattern, which describes an association with a bounded multiplicity that is greater than one. The ToN Association pattern is similar to the ToOne Association pattern. When the field additionally owns a dimension, which describes the number of elements that are stored by the field, then the association has a bounded multiplicity that is greater than one.

Figure 3.6(c) depicts the ToMany Association pattern, which describes an association with an unbounded multiplicity. The ToMany Association pattern is an extension of the ToN Association pattern. Additionally, the type of the field must be a classifier of a data structure, e. g. a list data structure, that can store other elements. The type of these elements is described by a type argument, which references the namespace that contains this type.

Method override pattern

Figure 3.7 shows the Method Override pattern (cf. Section 3.2). The graph pattern describes that class A is the superclass and class B is the subclass in a generalization. Furthermore, both classes own methods that have the same names[1] and do not have private visibilities.

[1] The running example only checks the equality of the method names, because the complexity of the OCL expressions (cf. Definition 2.2) that check the equality of the parameter types and the parameter orders are very complex and are neglected in this example for the sake of simplicity.

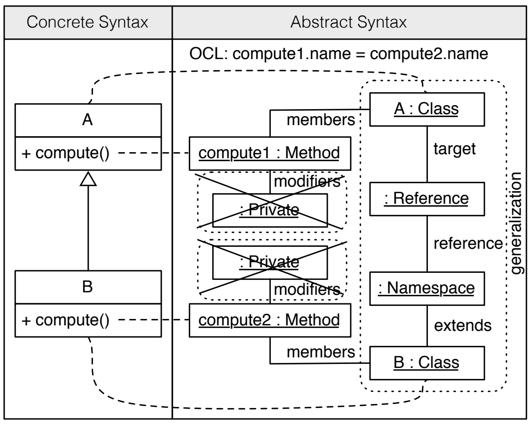


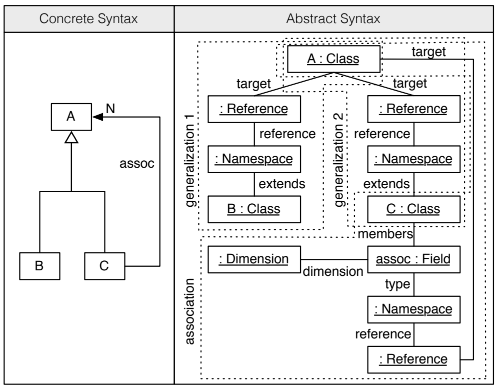
Figure 3.7.: Method Override pattern as running example

High-Level Graph Patterns

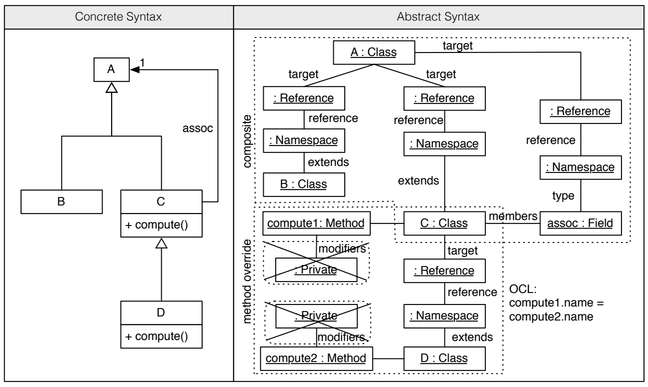
The following sections compose low-level graph patterns to high-level graph patterns that represent software design patterns and anti-patterns. For the sake of simplicity, these patterns only consider structural properties of design patterns. Behavioral properties are not in the scope of the running example.

Composite Design Pattern

Figure 3.8(a) depicts the Composite pattern (cf. Figure 3.1(a)). According to the dependency graph in Figure 3.3, the pattern reuses the Generalization pattern as denoted by the dotted generalization 1 and generalization 2 rectangles as well as the ToN Association pattern as denoted by the dotted association rectangle. The Generalization patterns overlap in the common superclass A. Furthermore, the Association pattern overlaps in class C that owns the association and class A that is the target of the association. Note that any combination of the Hierarchy and Association graph patterns can be employed.



(a) Composite



(b) Decorator

Figure 3.8.: Kinds of Design Pattern patterns as running example

Decorator Design Pattern

Figure 3.8(b) shows the Decorator pattern (cf. Figure 3.1(b)). The Decorator pattern has several structural properties in common with the Composite pattern such as the hierarchies and the association between the subclass C and its superclass A. Therefore, the Decorator pattern reuses the Composite pattern as denoted by the dotted composite rectangle. However, only Composite patterns are reused that employ the ToOne Association pattern, because the Decorator software design pattern employs an association with a multiplicity of one. Furthermore, the Decorator pattern reuses the Method Override pattern (cf. Figure 3.7) as denoted by the dotted method override rectangle to search for overridden methods in the subclass D. Note that similar variants of the Decorator pattern must be considered as for the Composite pattern such as variants that employ multi-level generalizations and multi-level interface implementations.

Extract Interface Anti-Pattern

Figure 3.9 shows the Extract Interface pattern (cf. Figure 3.2). Figure 3.9 shows that class A implements a public method and does *not* implement an interface as denoted by the crossed out interface I. In the abstract syntax, the Interface Implementation pattern is a complex NAC (cf. Definition 8) as denoted by the crossed out namespace, reference, and interface graph nodes (cf. Figure 2.3(d)). The Interface Implementation pattern can be also replaced by the Multi-Level Interface Implementation pattern to cover variants of the Extract Interface pattern that require the absence of multi-level interface implementations.

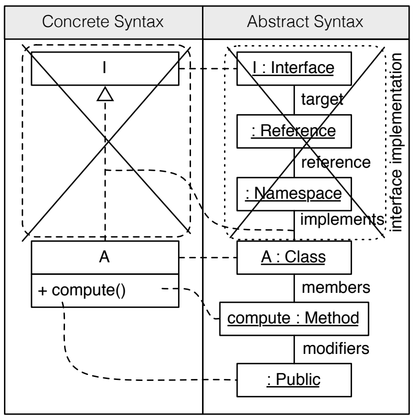


Figure 3.9.: Extract Interface Design Anti-Pattern

3.4. Summary

This chapter introduces the detection of employed software design patterns and anti-patterns as running example. For that purpose, the running example introduces a subset of possible graph patterns that describe design patterns. The running example, composes high-level patterns by means of lower-level patterns. This composition enables the reuse of lower-level patterns by different high-level patterns.

The design of the running example enables to setup multiple graph views for different kinds of software design patterns. From the perspective of evaluating the concepts that are presented in this thesis, the ASGs can be easily derived from the source code of software repositories. Moreover, these software repositories consist of change histories that can be used to modify the ASGs accordingly. Then, these changes demand a maintenance of the graph views that store the matches of the described graph patterns.

4. Overview

This chapter gives an overview of the framework for incremental view graph maintenance. First, user needs have to be identified that constitute the foundation for requirements towards a framework for incremental view graph maintenance. This thesis distinguishes two kinds of user groups: developers and end-users. *Developers* aim for increasing the throughput of graph queries by means of graph views. For that purpose, developers are responsible to describe which kinds of graph pattern matches are maintained by graph views and compose low-level views to high-level views. *End-users* want to query graphs interactively and, thus, need low response times of their queries. For that purpose, end-users make use of the graph pattern matches that are maintained by graph views.

This thesis focuses on the developer user group, because this thesis primarily aims for the effective definition as well as the efficient and scalable maintenance of graph views to increase the throughput of graph queries that are stated by end-users.

Section 4.1 identifies the needs of developers. Section 4.2 uses these needs to derive requirements towards a framework for incremental graph view maintenance. Section 4.3 gives an overview of the framework and outlines why the framework satisfies the requirements.

4.1. Needs

This section describes the needs of the developers towards a framework for the incremental maintenance of graph views. Section 4.1.1 describes needs concerning the definition of views. Section 4.1.2 describes needs concerning the computation of the view content. Section 4.1.3 describes needs concerning the maintenance of views.

4.1.1. Definition of Graph Views (N1)

The following paragraphs describe identified user needs concerning the definition of views.

Native Graph Views (N1a)

Developers want to define graph views that keep ready graph pattern matches for graph queries that are stated by end-users to increase the throughput of these graph queries. By means of these graph views, developers want to enrich graph query results, hide details of query results, and restructure query results [101]. Furthermore, developers want to employ native graph data models to store graph pattern matches, because these graph data models enable to employ graph operations for the definition and maintenance of graph views instead of relational operations that can decrease the query performance [83]. Moreover, developers and end-users want to query base graphs and graph views uniformly with graph queries. Thus, they need views that are graphs, too. These graphs should store query results in a memory-efficient way without copying graph nodes and edges from base graphs to views.

Creation of Graph Views (N1b)

Developers want to define single graph views that store matches of certain graph patterns. They want to employ graph query languages they are familiar with and do not want to be forced to make use of a dedicated query language for the definition of graph views.

According to the running example, developers want to define views that store matches for software design patterns and anti-patterns. For example, one view has to store all matches of the Composite pattern (Figure 3.8(a)), while another view has to store all matches of the Extract Interface pattern (Figure 3.9).

Composition of Graph Views (N1c)
Developers want to define graph views on top of other views to avoid redundant definitions of graph patterns and redundant searches for graph pattern matches. They want to freely choose a granularity level for the composition of low-level graph views to high-level graph views that fits best for their graph queries to be able to optimize the maintenance of graph views concerning memory consumption and maintenance execution time. They do not want to be restricted to a certain granularity level that is imposed by the composition approach.

According to the running example, developers compose graph views that store matches of the Multi-Level Interface Implementation pattern (cf. Figure 3.5(d)) by reusing views that store matches of the Generalization (cf. Figure 3.5(a)), Multi-Level Generalization (cf. Figure 3.5(c)), and Interface Implementation (cf. Figure 3.5(b)) pattern as depicted by Figure 3.3.

Referring to Graph Nodes and Edges of Graph Pattern Matches (N1d)
Developers and end-users want to retrieve graph pattern matches that are stored by graph views. Furthermore, they want to efficiently retrieve graph nodes and edges with certain roles in these matches without the need to match these graph nodes and edges a second time to determine their roles. End-users want to use these roles to post-process them in their applications. Developers want to make use of these roles, when they define views that need to refer to the roles of graph nodes and edges in reused matches.

According to the running example, developers and end-users want to retrieve matches of the Generalization pattern (cf. Figure 3.5(a)). End-users want to know which of both classes in a match of the Generalization pattern acts as superclass and subclass, respectively.

Developers must be able to refer to the superclass and subclass in matches of the Generalization pattern to describe that the lower and upper generalization of a Multi-Level Generalization pattern must have a class in common as depicted by Figure 3.5(c).

Reasonable Expressive Power (N1e)
Current graph query languages support pattern-based and path-based search [2]. Therefore, developers want to employ the same kinds of query languages to specify the content of views. The employed query languages for pattern-based search often have the expressiveness of graph conditions (cf. Definition 14). Thus, developers want to use conjunctions, disjunctions, and negations to describe the kinds of matches that are stored by graph views. For path-based searches, developers want to employ recursive graph conditions (cf. Definition 15).

According to the running example, when developers define a view that stores matches of the Composite pattern (cf. Figure 3.8(a)), they have to describe a disjunction of Association patterns (cf. Figure 3.6), a disjunction of Hierarchy patterns (cf. Figure 3.5), and a conjunction of Hierarchy and Association patterns.

When developers define a view that stores matches of the Extract Interface pattern (cf. Figure 3.9), they have to describe that matches of the Interface Implementation pattern (cf. Figure 3.5(b)) must not exist to detect Extract Interface design anti-patterns.

When developers define a view that stores matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)), they have to describe that matches of the Multi-Level Generalization pattern base on other matches of the Multi-Level Generalization pattern, recursively.

4.1.2. Graph Data Models and Graph Query Languages (N2)

The following paragraphs describe identified user needs concerning the graph data models and graph query languages that should be employed for the maintenance of graph views.

User's Choice of Graph Data Model (N2a)

Angles [2] states that different kinds of graph data models are employed to store entities and their relationships such as simple graphs or attributed graphs, with or without labels of nodes and edges, and directed or undirected edges. The developers expect that a framework for the maintenance of graph views can handle different kinds of graph data models uniformly, e. g., the framework should support typed attributed graphs and hypergraphs[1] in a uniform way.

User's Choice of Graph Query Language (N2b)

Angles [2] states that different kinds of graph query languages exist. For example, Neo4j [82] employs the declarative query language Cypher [82], AllegroGraph [34] employs SPARQL [2], while other graph query approaches employ *"SQL-based query languages with special instructions for querying graphs"* [2]. The developers want to employ query languages of their choice, e. g., imperative as well as declarative query languages, and do not want to be restricted to a certain query language for the definition of graph views.

Execute View Definitions (N2c)

Developers want to execute the definitions of graph views to initially create and, afterwards, maintain the matches that are stored by views. In combination with need N2b - Query Languages, different query languages must be supported by the execution.

Modify Graphs (N2d)

End-users want to modify base graphs and expect that the framework automatically maintains the graph views to ensure that the views store correct and complete graph pattern matches. Thus, developers need a framework that handles modifications of base graphs and triggers the maintenance of graph views that are impacted by changes of base graphs.

4.1.3. Maintenance of Graph Views (N3)

The following sections describe the user needs concerning the maintenance of graph views.

Interactive Usage of Graph Views (N3a)

When end-users make use of graph views, they expect that the average response time of queries is less than the response time without views. Thus, developers need to define graph patterns that specify the content of views to decrease the response time of graph queries.

Consistent Graph Views (N3b)

When end-users state graph queries that make use of graph views, they expect that views are consistent with the graphs from which they are derived. That means, the end-users expect that queries retrieve the same result, when views are employed and are not employed to answer queries. The end-users always need consistent query results.

General-Purpose Maintenance Algorithms (N3c)

The end-users and developers do not want to be responsible to keep views consist on their own. Instead, they need general-purpose maintenance algorithms that maintain graph views, which are derived with the help of a query language that is chosen by developers.

[1]Hypergraphs are graphs with n-ary graph edges.

4.2. Requirements

The identified user needs lead to the following requirements. The framework must provide a lightweight approach for graph views, which employs graphs to store graph pattern matches and provides capabilities to define views and their interrelationships. Furthermore, the framework must provide capabilities to specify the content of views and to maintain these views to keep them consistent with the base graphs from which they are derived.

Section 4.2.1 describes how graph views should store graph pattern matches. Section 4.2.2 describes how the framework should enable to describe views and their interrelationships. Section 4.2.3 describes how the framework should compute the content of views. Section 4.2.4 describes how the framework should maintain views. Section 4.2.5 summarizes the elicited requirements and discusses why they cover the user needs.

4.2.1. Lightweight Graph Views (R1)

The framework must provide a lightweight approach that enables developers to store graph pattern matches natively as graphs. The following paragraphs describe requirements concerning a native graph data model for graph views (R1a), how views should store matches (R1b), and how views should store additional properties of matches (R1c).

Native Graph Data Model for Graph Views (R1a)

This thesis employs graph data models to store entities and relationships between these entities. According to need N1a - Native Views, developers want to employ a native graph data model for storing graph pattern matches in graph views. This graph data model must be generic enough to support common use cases of views such as enrich, hide, and restructure query results. Furthermore, this graph data model must overcome expensive join operations of relational data models, when graph edges between graph nodes have to be traversed.

Memory-Efficient Graph Views (R1b)

According to need N1a - Native Views, the graph views should store graph pattern matches in a memory-efficient way. Thus, the framework should not copying graph nodes and edges from base graphs to graph views to keep the memory consumption low. Instead, marking graph pattern matches is much more efficient than copying graph nodes and edges.

Additional Properties for Stored Graph Pattern Matches (R1c)

According to need N1a - Native Views, developers want to enrich graph pattern matches with additional knowledge that is derived from the graph nodes and edges of these matches. Thus, the graph views must also enable to store additional data values for each found match. These values must be maintained by the queries that define the content of views.

4.2.2. Model Graph Views (R2)

One major requirement is the effective modeling of views and their interrelationships. Modeling of views means that the developers describe by means of a modeling language which kinds of views exist (N1b - Single View), how these views are related (N1c - Composed Views), and how end-users and developers can retrieve maintained matches effectively (N1d - Accessibility). The modeling approach must support a reasonable expressiveness for views (N1e - Expressiveness).

The following paragraphs describe the requirements concerning the encapsulation of graph queries (R2a), the effective storing of graph pattern matches (R2b), the reusability of stored matches (R2c), and the expressiveness of the modeling approach (R2d, R2e).

Encapsulation of Graph Queries (R2a)
According to the needs N2a - Graph Models and N2b - Query Languages, the framework must support definitions of graph views that encapsulate graph queries in a way that the framework abstracts from graph data models and graph query languages. This abstraction makes the framework applicable to different kinds of graph data models and query languages, because the framework can handle the encapsulated queries uniformly. For example, the modeling language must enable the framework to handle imperative and declarative queries uniformly.

Effectiveness of Graph Views (R2b)
According to need N1d - Accessibility, the graph views must store graph pattern matches effectively. That means, the roles of graph nodes and edges of graph pattern matches must be preserved, when storing matches, to enable users to reuse these roles for post-processing, e. g., when they define views that build on the matches that are stored by other views.

For example, when developers create views that store matches of the Generalization pattern (cf. Figure 3.5(a)), end-users are interested in the super- and subclasses of generalizations. Thus, each stored match must keep track of which class acts as super- and subclass.

Reusability of Graph Views (R2c)
According to need N1b - Single View and N1c - Composed Views, the framework must enable graph views to reuse the graph pattern matches that are stored by other graph views for graph pattern matching. For that purpose, the modeling approach must enable to describe dependencies between definitions of graph views. In combination with the encapsulation of queries (R2a - Encapsulation), the modeling approach must enable developers to specify interfaces that describe the input and output of graph views.

For example, the graph view that stores matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)) must be able to efficiently access the super- and subclasses of matches, which are stored by the Generalization view, to find matches of the Generalization pattern (cf. Figure 3.5(a)) that overlap in the subclass of one match and the superclass of another match.

Nesting of Graph Conditions (R2d)
According to need N1e - Expressiveness, the language must support the modeling of graph views that have the same expressive power as graph conditions (cf. Definition 14). In combination with the reuse of graph views (R2c - Reusability), the language must enable developers to model conjunctions, disjunctions, and negations, when they combine views to complex views.

According to the running example, the Composite graph view must implement disjunctions of Hierarchy patterns, disjunctions of Association patterns, as well as conjunctions of Hierarchy patterns and Association patterns to detect implementation variants of Composite design patterns that employ single-level or multi-level generalizations and n-ary associations.

The graph view for the Extract Interface pattern (cf. Figure 3.9) reuses graph pattern matches of the Interface Implementation pattern (cf. Figure 3.5(b)) in negative sense. That means, the graph view requires the non-existence of matches of the Interface Implementation pattern to detect Extract Interface design anti-patterns.

Recursive Graph Conditions (R2e)
According to the need N1e - Expressiveness, the modeling language must enable developers to describe recursive graph conditions (cf. Definition 15).

According to the running example, the view that stores matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)) depends on itself, because matches of the Multi-Level Generalization pattern can result in additional matches of the Multi-Level Generalization pattern.

4.2.3. Embedding of Graph Queries (R3)

The framework must enable developers to employ graph queries as operationalization for the definition of graph views (N2c - View Execution). These graph queries must search for graph pattern matches that are maintained by graph views.

The following paragraphs describe requirements concerning the support of different kinds of graph query languages (R3a) and the execution of embedded queries (R3b).

Support Different Kinds of Graph Pattern Matching (R3a)

According to the need N2b - Query Languages, the framework must enable developers to employ different kinds of graph query languages to describe the graph pattern matches that are stored by views. These query languages can employ different algorithms for pattern matching such as injective and non-injective graph pattern matching. Thus, the framework must support these different kinds of pattern matchings.

Employ Graph Transformations (R3b)

According to the need N1a - Native Views, views must employ a native graph data model to store graph pattern matches. One natural way to add, remove, and update graph nodes and edges is to employ graph transformations (cf. Definition 13). Thus, the framework should employ graph transformations to natively maintain graph views.

4.2.4. Maintenance of Graph Views (R4)

The framework must maintain graph views, when base graphs change, to ensure that the derived views are consistent with these base graphs (N3b - Consistency). This maintenance must be efficient and scalable in time and space to enable an interactive usage of views (N3a - Interactivity). The maintenance algorithms must be general enough to support arbitrary graph query languages that are employed for the definition of views (N3c - Generic Algorithm).

The following paragraphs describe requirements towards the monitoring of graph changes (R4a), the pruning of search spaces (R4b), and the change propagation between views (R4c).

Monitoring of Graph Changes (R4a)

According to the need N2d - Graph Modification, the framework must track changes of base graphs to ensure the consistency of graph views (N3b - Consistency) by means of a graph view maintenance (N3 - View Maintenance). For that purpose, the framework must monitor base graphs and track how these base graphs change over time. The framework must relate these changes to stored graph pattern matches of views and must trigger the view maintenance, when the base graph changes have an impact on these graph pattern matches.

According to the running example, the view that enumerates matches for the Generalization pattern (cf. Figure 3.5(a)) must be updated, when classes are added to or removed from base graphs, because these classes may satisfy or dissatisfy the Generalization pattern.

Efficiency of Graph View Maintenance (R4b)

According to the needs N3a - Interactivity and N3b - Consistency, the framework must maintain graph views efficiently and scalable. The efficient maintenance must reduce the time that is required to maintain the graph pattern matches that are stored by graph views. The scalable maintenance must keep the required time to maintain graph views independent from the number of graph nodes and edges that are stored by base graphs.

According to the running example, when classes are added to the base graphs, it is sufficient to only search within the scope of these classes for new matches of the Generalization pattern (cf. Figure 3.5(a)), because these classes are either superclasses of other classes, subclasses of other classes, or do not belong to a generalization at all.

Propagation of Graph View Changes (R4c)

According to the need N1c - Composed Views, graph views depend on each other and reuse graph pattern matches that are maintained by other graph views (R2c - Reusability). Therefore, the framework must propagate matches between graph views to ensure that dependent graph views maintain their stored graph pattern matches as well, when base graphs change.

According to the running example, the Multi-Level Generalization view depends on the Generalization view. When matches of the Generalization pattern (Figure 3.5(a)) are added to or removed from the Generalization view, then these added and removed graph pattern matches must be propagated to the dependent Multi-Level Generalization view as well to update the stored matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)).

4.2.5. Summary

Table 4.1 shows which requirements cover which user needs. The black circles denote that a requirement covers a certain need. The black / white circles denote that a requirement contributes to the covering of a need, but does not cover the need completely. The white circles denote that a requirement does not cover a certain need.

The requirements concerning lightweight graph views such as a R1a - Nativeness, R1b - Memory-Efficiency, R1c - Match-Properties, cover the need N1a - Native Views, because these requirements aim for graph views that are graphs, too, and store matches efficiently.

The requirements concerning the modeling of views such as R2a - Encapsulation, R2b - Effectiveness, R2c - Reusability, R2d - Nesting, and R2e - Recursion cover the user needs N1a - Native Views, N1b - Single View, N1c - Composed Views, N1d - Accessibility, and N1e - Expressiveness, because these requirements aim for a modeling language that enable developers to define and combine graph views in a reusable manner with the same expressiveness as nested and recursive graph conditions. Furthermore, the requirement R2a - Encapsulation (partially) covers the user needs N2a - Graph Models, N2b - Query Languages, N2c - View Execution, and N3c - Generic Algorithm, because this requirement aims for encapsulated graph queries that enable the framework to handle arbitrary queries uniformly.

The requirements concerning the embedding of graph queries into view definitions such as R3a - Languages and R3b - Transformations cover the user needs N2b - Query Languages and N2c - View Execution, because these requirements aim for the support of arbitrary graph query languages as operationalization of graph views.

The requirements concerning the maintenance of views such as R4a - Monitoring, R4b - Time-Efficiency, and R4c - Propagation cover the need N3a - Interactivity, N3b - Consistency, and N3c - Generic Algorithm, because these requirements aim for maintenance algorithms that efficiently update graph views based upon base graph changes in a scalable manner. The requirements R4b - Time-Efficiency and R4c - Propagation cover the needs N1b - Single View and N1c - Composed Views, because they aim for a time-efficient maintenance of single and composed views. The requirements R4a - Monitoring, R4b - Time-Efficiency, and R4c - Propagation cover the user needs N2c - View Execution and N2d - Graph Modification, because they aim for an efficient execution of view definitions and change propagation between graph view, when base graphs change.

Table 4.1.: Mapping needs of developers to requirements

		N1 - View Definition					N2 - Query Definition				N3 - View Maintenance		
		N1a - Native Views	N1b - Single View	N1c - Composed Views	N1d - Accessibility	N1e - Expressiveness	N2a - Graph Models	N2b - Query Languages	N2c - View Execution	N2d - Graph Modification	N3a - Interactivity	N3b - Consistency	N3c - Generic Algorithm
R1 - Lightweight Views	R1a - Nativeness	●	○	○	○	○	○	○	○	○	○	○	○
	R1b - Memory-Efficiency	●	○	○	○	○	○	○	○	○	○	○	○
	R1c - Match-Properties	●	○	○	○	○	○	○	○	○	○	○	○
R2 - View Modeling	R2a - Encapsulation	○	●	○	○	◐	●	●	◐	○	○	○	◐
	R2b - Effectiveness	●	○	◐	●	○	○	○	○	○	○	○	○
	R2c - Reusability	○	●	●	●	◐	○	○	○	○	○	○	○
	R2d - Nesting	○	○	●	○	●	○	○	○	○	○	○	○
	R2e - Recursion	○	○	●	○	●	○	○	○	○	○	○	○
R3 - Embed Queries	R3a - Languages	○	○	○	○	○	○	●	○	○	○	○	○
	R3b - Transformations	●	○	○	○	○	○	○	●	○	○	○	○
R4 - Maintenance	R4a - Monitoring	○	○	○	○	○	○	○	○	●	○	●	●
	R4b - Time-Efficiency	○	●	●	○	○	○	○	●	○	●	●	●
	R4c - Propagation	○	○	●	○	○	○	○	○	●	○	●	●

covered: ●; partially covered: ◐; not covered: ○

4.3. Overview of the Approach

Figure 4.1 depicts an overview of the proposed framework. First, Section 4.3.1 describes the main system components of the framework. Afterwards, Section 4.3.2 discusses why the proposed framework covers the elicited requirements of Section 4.2.

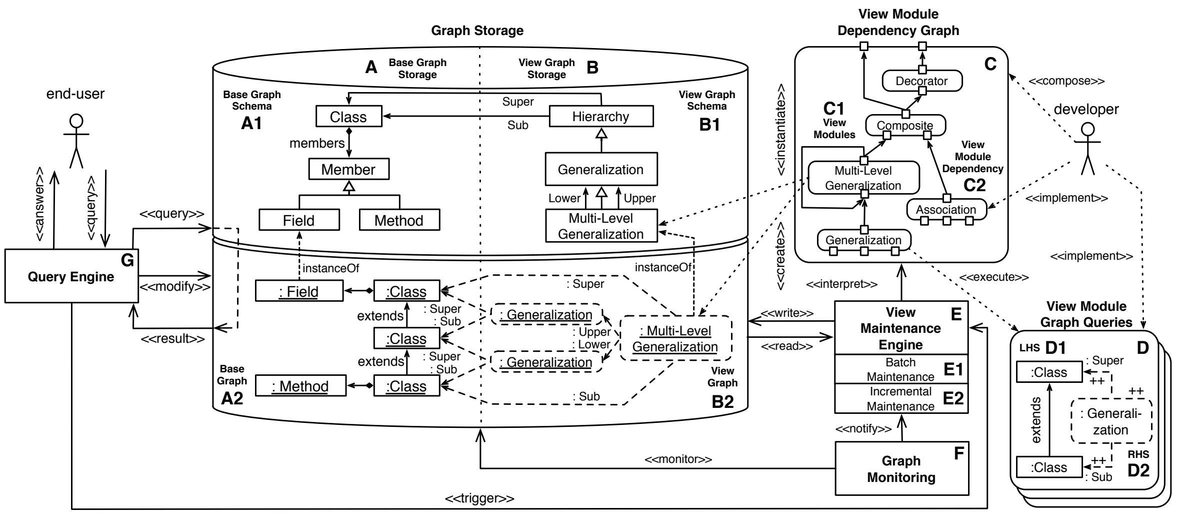


Figure 4.1.: Overview of the approach

4.3.1. System Components and Parts

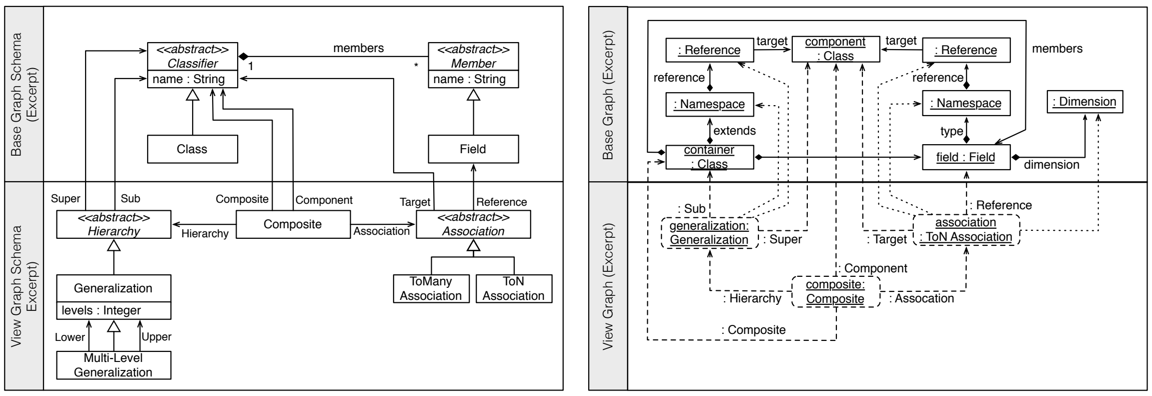
The framework distinguishes six main components called graph storage (A, B), view module dependency graph (C), view module queries (D), view maintenance engine (E), graph monitoring (F), and query engine (G). The next sections describe each component in detail.

Graph Storage

The graph storage (A, B) consists of a base graph storage (A) and a view graph storage (B).

The base storage (A) stores base graphs (A2). Base graphs (A2) are typed attributed graphs (cf. Section 2.1.1) that are created and modified by end-users. Base graphs (A2) must conform to a base graph schema (A1). The base graph schema is an attributed type graph (cf. Section 2.1.1) that describes which kinds of graph nodes and edges can exist in base graphs.

The view graph storage (B) stores view graphs (B2). View graphs (B2) are typed attributed graphs (cf. Section 2.1.1) that constitute graph views and store graph pattern matches. Graph nodes of view graphs represent markings of graph pattern matches. Graph edges of view graphs reference the graph nodes that belong to these matches and describe the role of the graph nodes in these matches. View graphs (B2) must conform to a view graph schema (B1). The view graph schema (B1) is an attributed type graph (cf. Section 2.1.1) that extends the base graph schema (A1) to describe which kinds of graph pattern matches are marked by graph nodes and edges of view graphs. Node types of the view graph schema describe which kinds of markings for graph pattern matches can exist in view graphs. Edge types of the view graph schema describe which kinds of graph edges are used to mark graph nodes of matches. These edge types are used to denote the role of graph nodes in matches. Furthermore, the view graph schema describes kinds of attributes that can be stored by graph nodes to enrich the matches with additional data values.



(a) Excerpt of base graph schema and view graph schema as UML class model (b) Excerpt of base graph and view graph as UML object model

Figure 4.2.: Kinds of employed graphs

For example, Figure 4.2(a) shows an excerpt of the base graph schema (A1) and view graph schema (B1). According to the running example, the base graph schema on top describes that classifiers, such as classes, own members, such as fields. The view graph schema at the bottom describes that view graphs store matches of the Generalization, Multi-Level Generalization, ToN Association, ToMany Association, and Composite patterns.

The Hierarchy type describes that markings for matches of Hierarchy patterns (cf. Figure 3.5) mark the super- and subordinate classifiers in the matches, such as the super- and subclasses of generalizations, by means of graph edges with Super and Sub type. Additionally, the markings for matches of the Multi-Level Generalization pattern mark the reused generalization matches by means of graph edges with Lower and Upper type.

The Association type describes that markings for matches of the Association patterns (cf. Figure 3.6) mark the fields and classifiers that act as reference and target of the reference by means of graph edges with Reference and Target type.

The Composite type describes that markings for matches of the Composite pattern mark the reused matches of the Hierarchy and Association patterns by means of graph edges with Hierarchy and Association type. Furthermore, these markings reference the composite and component classifiers of the detected Composite design pattern by means of graph edges with Composite and Component type.

The top of Figure 4.2(b) shows a base graph, which consists of a container class that extends a component class. Furthermore, the container class owns an one-dimensional field that targets at the component class. The bottom of Figure 4.2(b) shows a view graph that contains markings for matches of the Generalization, ToN Association, and Composite patterns.

The generalization node with Generalization type marks a match of the Generalization pattern (cf. Figure 3.5(a)). According to the view graph schema of Figure 4.2(a), the generalization node owns two edges with Super and Sub type that mark the component class as superclass and the container class as subclass of the generalization. Furthermore, the generalization node marks the nodes with Namespace and Reference type in between, because they belong to the match of the Generalization pattern as well.

The association node with ToN Association type marks a match of the ToN Association pattern (cf. Figure 3.6(b)). According to the view graph schema of Figure 4.2(a), the association node owns two edges with Reference and Target type that mark the field as reference and the component class as target type of the reference. Furthermore, the association node marks the nodes with Namespace, Reference, and Dimension type in between, because they belong to the match of the ToN Association pattern as well.

The composite node with Composite type marks a match of the Composite pattern (cf. Figure 3.8(a)). According to the view graph schema of Figure 4.2(a), the composite node owns four edges with Hierarchy, Association, Composite, and Component type. The edge with Hierarchy type marks the reuse of the Generalization match that is marked by the generalization node. The edge with Association type marks the reuse of the Association match that is marked by the association node. The edges with Component and Composite type reference the component and container classes to mark that the component class acts as component and the container class acts as composite in the detected Composite software design pattern.

View Module Dependency Graph

The view module dependency graph (C) describes which kinds of graph views are maintained by the framework. For that purpose, the view module dependency graph consists of view modules (C1) and view module dependencies (C2).

View modules (C1) enable developers to encapsulate graph queries behind interfaces that can be used uniformly by the framework. These interfaces describe the input and output of view modules, i. e., which kinds of graph nodes are processed by the encapsulated queries and which kind of graph nodes are created in the view graph, when the encapsulated queries find graph pattern matches. The developers use the node types that are defined in the base graph schema and view graph schema to describe the input and output of view modules.

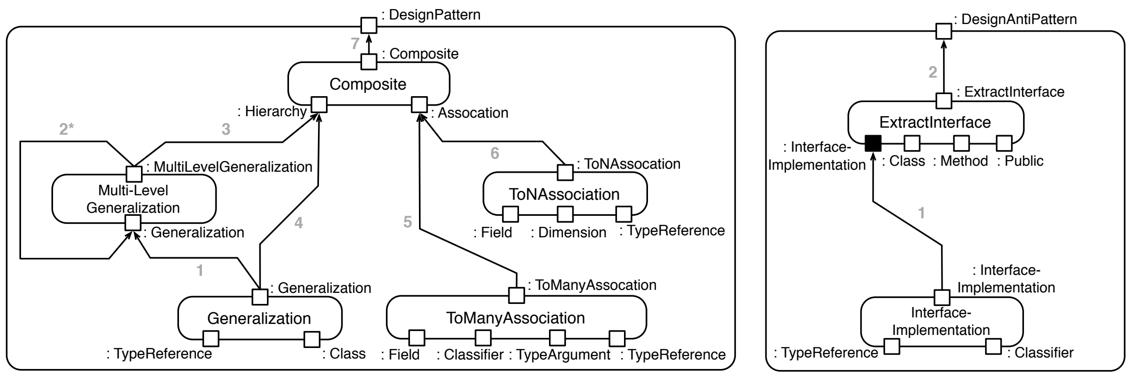
View module dependencies (C2) enable developers to describe how view modules (C1) reuse markings of graph pattern matches that are stored and maintained by other views.

According to the running example, Figure 4.3 shows two view module dependency graphs for design patterns and anti-patterns. Both view module dependency graphs depict view modules as rounded rectangles and view module dependencies as solid lines. The arrow heads of the solid lines denote the dependent view module. The small rectangles that are attached to the view modules denote the interfaces of view modules.

Figure 4.3(a) shows an excerpt of the view module dependency graph for design patterns. Figure 4.3(a) depicts the Composite, Generalization, Multi-Level Generalization, ToN Association, and ToMany Association view modules. These view modules maintain matches of the Generalization (cf. Figure 3.5(a)), Multi-Level Generalization (cf. Figure 3.5(c)), ToN Association (cf. Figure 3.6(b)), ToMany Association (cf. Figure 3.6(c)), and Composite (cf. Figure 3.8(a)) patterns. These patterns describe which kinds of graph nodes are required by the graph queries that are encapsulated by the view modules.

According to the running example, the Generalization pattern (cf. Figure 3.5(a)) consists of pattern nodes with type Class, Namespace, and Reference. Therefore, the Generalization view module requires graph nodes of type Class and TypeReference as input and produces nodes of type Generalization in the view graph as output. According to Figure 3.4, the TypeReference type is the supertype of the Namespace and Reference type.

View modules that maintain matches for high-level patterns require graph nodes of view graphs. For example, the Multi-Level Generalization view module requires nodes of type Generalization as input and produces nodes of type Multi-Level Generalization in the view graph as output. For that purpose, Figure 4.3(a) shows a view module dependency between the Generalization view module and Multi-Level Generalization view module. Furthermore, Figure 4.3(a) shows a view module dependency between the output and input connector of the Multi-Level Generalization view module to describe that markings for matches of the Multi-Level



(a) Design pattern (b) Design anti-pattern

Figure 4.3.: View module dependency graphs (the numbers denote a possible execution order)

Generalization pattern can lead to additional matches of the Multi-Level Generalization pattern. Thus, the Multi-Level Generalization view module implements a recursion.

The input connectors of the Composite view module describe that the Composite view module requires graph nodes with Hierarchy and Association type to find matches of the Composite pattern. Therefore, the Composite view module implements a conjunction of matches for Hierarchy and Association patterns. Furthermore, the Hierarchy and Association input connectors have two incoming view module dependencies. The two incoming view module dependencies of the Hierarchy input connector describe that the Composite view module implements a disjunction of matches for the Generalization and Multi-Level Generalization pattern. Accordingly, the two incoming view module dependencies of the Association input connector describe that the Composite view module implements a disjunction of matches for the ToMany Association and ToN Association pattern.

Figure 4.3(b) shows an excerpt of the view module dependency graph for design anti-patterns. Figure 4.3(b) depicts the Interface Implementation and Extract Interface view module. These view modules maintain matches of the Interface Implementation (cf. Figure 3.5(b)) and Extract Interface (cf. Figure 3.9) patterns. The Interface Implementation view module requires graph nodes with Classifier type (supertype of the Class and Interface node type (cf. Figure 3.4)) and graph nodes with TypeReference type (supertype of the Namespace and Reference node type (cf. Figure 3.4)) to find matches of the Interface Implementation pattern. The Extract Interface view module requires graph nodes with InterfaceImplementation, Class, Method, and Public type. For that purpose, Figure 4.3(b) depicts a view module dependency between the Interface Implementation view module and the Extract Interface view module. The InterfaceImplementation input connector of the Extract Interface view module is negated as denoted by the black filled rectangle to describe that the Extract Interface view module requires the non-existence of matches of the Interface Implementation pattern to find matches of the Extract Interface pattern. Therefore, the Extract Interface view module implements a negation of matches for the Interface Implementation pattern.

View Module Graph Queries

View modules (C1) encapsulate graph queries (D) as graph transformation rules (cf. Section 2.1.2) that search for all matches of certain graph patterns (D1) and create markings for these matches in the view graph (D2). A marking of a match consists of a graph node, which describes the kind of the match, and owns edges, which mark all graph nodes of base and view graphs that belong to the match. The edge types, which are used to mark the graph nodes of the match, describe the roles of the graph nodes in the match.

According to the running example, Figure 4.4 shows the implementation of the view modules. Figure 4.4 employs the shorthand notation for graph transformation rules as described in Section 2.1.2. Figure 4.4 depicts the left-hand sides of the transformation rules in black color and the side-effects of the right-hand sides of the transformation rules in gray color. Furthermore, Figure 4.4 depicts pattern nodes and edges that refer to the base graphs as solid rectangles and lines. For better readability, Figure 4.4 depicts pattern nodes and edges that refer to the view graphs as dashed rectangles and lines. Figure 4.4(a) shows an excerpt of the graph transformation rules for design patterns. Figure 4.4(b) shows an excerpt of the graph transformation rules for design anti-patterns.

The left-hand side of the Generalization transformation rule implements the Generalization pattern (cf. Figure 3.5(a)). For each found match, the right-hand side of the transformation rule creates a graph node with Generalization type in the view graph to mark the super- and subclass in the match by means of graph edges with Super and Sub type. Additionally, the right-hand side of the transformation rule creates two edges with a default type that mark the graph nodes with Namespace and Reference type, because they belong to the match as well.

The left-hand side of the Multi-Level Generalization transformation rule implements the Multi-Level Generalization pattern (cf. Figure 3.5(c)) by reusing graph nodes that mark matches of the Generalization pattern (cf. Figure 3.5(a)). For each found match, the right-hand side of the transformation rule marks the outermost classes of the multi-level generalization as super- and subclass by means of graph edges with Super and Sub type. Furthermore, the transformation rule marks the reused matches of the Generalization pattern by means of graph edges with Lower and Upper type.

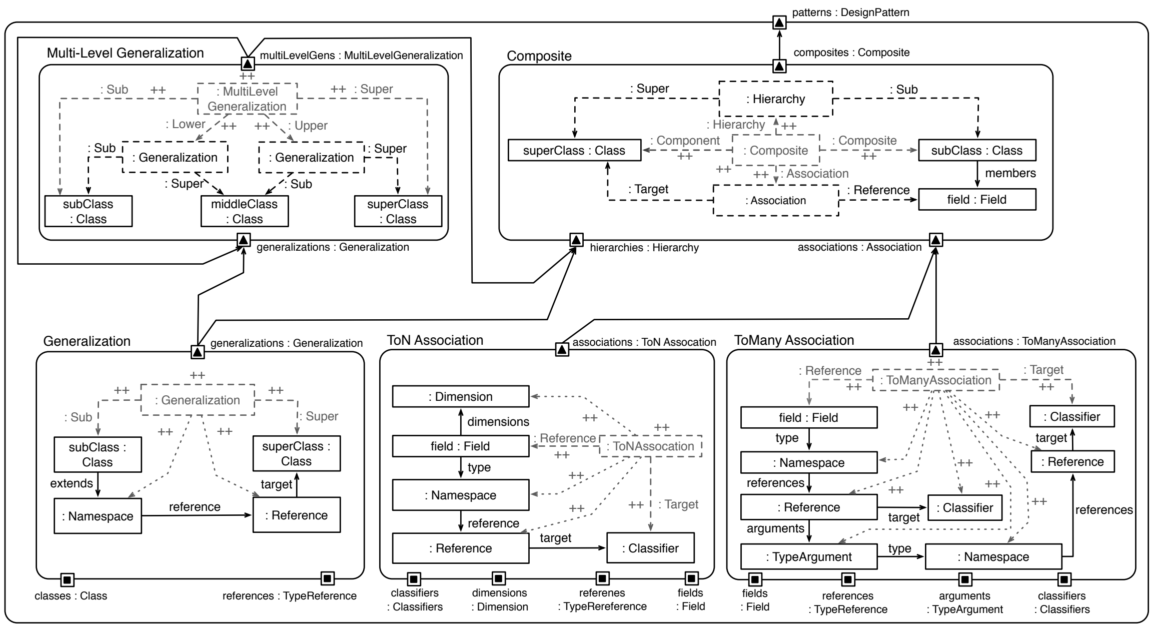
Since the two graph nodes with Generalization type, which belong to the left-hand side of the transformation rule, can match graph nodes that mark matches of the Multi-Level Generalization pattern as well, the Multi-Level Generalization transformation rule also find matches of the Multi-Level Generalization pattern, which reuse matches of multi-level generalizations. Thus, the Multi-Level Generalization view module implements recursion.

The left-hand side of the Composite transformation rule implements the Composite pattern (cf. Section 3.8(a)) by reusing matches of the Hierarchy and Association graph patterns. For each found match, the right-hand side of the transformation rule marks the composite and component of the detected Composite design pattern by means of graph edges with Composite and Component type. Furthermore, the right-hand side of the transformation rule marks the reused matches of the Hierarchy and Association graph patterns by means of graph edges with Hierarchy and Association type.

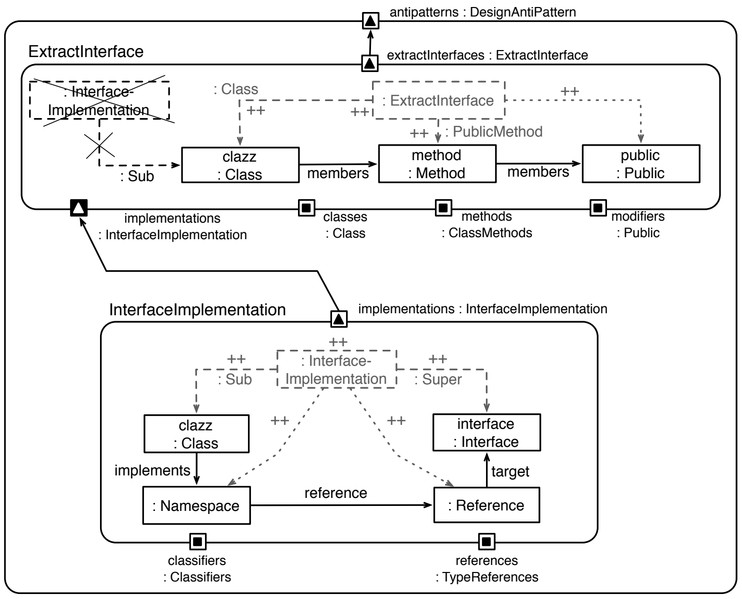
The Composite view module implements disjunctions of matches for Hierarchy and Association patterns, because the graph nodes with Hierarchy and Association type can match graph nodes that have a subtype of the Hierarchy and Association type.

The Composite view module implements conjunctions of matches for Hierarchy and Association patterns, because the Composite view module requires both kinds of matches to lookup matches for the Composite pattern.

Figure 4.4(b) shows the Interface Implementation view module and its graph transformation rule that is similar to the Generalization view module. The left-hand side of the Extract Interface transformation rule implements the Extract Interface pattern (cf. Figure 3.9) by reusing matches of the Interface Implementation pattern (cf. Figure 3.5(b)) in negated manner. Thus, the transformation rule implements a complex NAC (cf. Definition 8). For each found



(a) Design pattern



(b) Design anti-pattern

Figure 4.4.: Implementation of view modules by means of graph transformation rules

match, the right-hand side of the transformation rule marks the class and the method for which an interfaces should be extracted by means of graph edges with Class and PublicMethod type. The right-hand side of the transformation rule also marks the public node.

View Maintenance Engine

The view maintenance engine (E) interprets the view module dependency graph (C) and executes the graph transformations (D) that are encapsulated by view modules (C1) to initially instantiate the view graph schema (B1) and, later on, maintain the instantiated view graph (B2), when end-users modify base graphs (A2) with the help of the query engine (G).

The view maintenance engine (E) provides a batch maintenance algorithm (E1) and an incremental maintenance algorithm (E2). The batch maintenance algorithm (E1) processes the complete base graphs and view graphs during view graph maintenance. In contrast, the incremental maintenance algorithm (E2) takes modifications of the base graphs into account to process only portions of the base graphs and view graphs that underwent modifications. For that purpose, the framework derives candidate sets of graph nodes from changes of base graphs and view graphs and passes these candidate sets as pruned search space to view modules. The framework computes these candidate sets with the help of a reachability test that checks which graph nodes of the base graphs and view graphs are reachable from created, deleted, and modified graph nodes of the base graphs and view graphs.

As rule of thumb, the view module that creates a marking of a graph pattern match in the view graph is also responsible for its maintenance. For that purpose, the view modules are able to create missing markings, delete obsolete markings, and update suspicious markings. Furthermore, the framework propagates the changes of view graphs between view modules.

According to the running example, when end-users create a Class node in a base graph, the view maintenance engine must execute the Generalization view module to check whether new matches of the Generalization pattern (cf. Figure 3.5(a)) result from the added Class node. If yes, the Generalization view module must create new markings for these matches.

When end-users remove a Class node from the base graph, the view maintenance engine must check whether this graph node was part of a match of the Generalization pattern. For that purpose, the framework looks up the impacted graph nodes with Generalization type in the view graph and passes these graph nodes to the responsible Generalization view module for maintenance. If these graph nodes do not mark matches of the Generalization pattern anymore, the view module deletes the graph node and their edges from the view graphs.

When end-users modify a Class node in the base graph, the view maintenance engine must check whether this graph node is part of a match of the Generalization pattern and whether the modification dissatisfies the pattern. For that purpose, the framework looks up the impacted graph nodes with Generalization type in the view graph and passes these graph nodes to the responsible Generalization view module for maintenance. Then, the view module checks whether these graph nodes still mark matches of the Generalization pattern. If these graph nodes still mark matches of the Generalization pattern, the Generalization view module preserves these graph nodes. If these graph nodes do not mark matches of the Generalization pattern anymore, the Generalization view module flags these graph nodes for deletion.

Graph Monitoring

End-users state queries to apply side-effects to base graphs. The graph monitoring (F) tracks added, deleted, and modified graph nodes and edges of the base graphs (A2). The graph monitoring (F) notifies the view maintenance engine (E) to trigger the view graph maintenance, when the base graph change.

Query Engine
The query engine (G) enables end-users to query base graphs (A2) and view graphs (B2). Furthermore, the query engine (G) enables end-users to modify the base graphs (A2). The framework supports *immediate* view graph maintenance and *deferred* view graph maintenance. When the framework employs an immediate maintenance, the view maintenance engine (E) processes monitored graph changes immediately. When the framework employs a deferred maintenance, the query engine triggers the view maintenance, when users state graph queries.

4.3.2. Summary

This section summarizes how the framework addresses the elicited requirements of Section 4.2. The following sections describe why the system components of the framework meet the elicited requirements. Table 4.2 maps the requirements to the system components and their parts.

Graph Storage
The graph storage (A, B) employs typed attributed graphs with inheritance as native graph data model (R1a - Nativeness) for base graphs (A2) and view graphs (B2), because it is a general graph data model, which covers the graph data models that are employed in practice to store entities and their relationships [2].

The graph nodes of the view graphs represent matches of graph patterns. Each graph node of view graphs marks all graph nodes that belong to a graph pattern match. The types of these graph nodes in the view graphs describe the kinds of the matches that are marked by the graph nodes. Thus, all graph nodes that satisfy a pattern are stored memory-efficient by the view graph (R1b - Memory-Efficiency), because the view graph does not store copies of graph nodes and edges. Furthermore, the nodes of the view graph own typed graph edges that describe the role of graph nodes in matches. Thus, the typed graph nodes and edges of view graphs support an effective marking of matches (R2b - Effectiveness).

The graph nodes in the view graph can own attributes that store additional data values (R1c - Match-Properties). These data values can be used to enrich matches with additional information, e. g., similarity values of two entities.

View Module Dependency Graph
View modules describe interfaces for graph queries and, therefore, encapsulate queries (R2a - Encapsulation). The interfaces of view modules provide information about which kinds of graphs nodes are required by queries and which kinds of graph nodes are created in the view graph to mark matches of patterns (R1b - Memory-Efficiency). Thus, the framework is not aware of graph query implementations and can handle the encapsulated queries uniformly.

The interfaces of view modules (C1) and the view module dependencies (C2) between view modules enable developers to describe the reuse of matches (R2c - Reusability). Due to the view graph schema (B1) that describes which kinds of graph nodes can act in certain roles in matches, users and other view modules know which kinds of graph edges can be used to retrieve graph nodes with certain roles in matches. Since the view graph schema describes node types and their edge types that are used to refer to graph nodes with certain roles in matches, dependent view modules are enabled to receive and post-process graph nodes that mark certain kinds of matches (R2c - Reusability).

The view module dependencies together with the view modules enable developers to nest graph conditions (R2d - Nesting). The view module dependency graph enables to express disjunctions, conjunctions, and negations of graph conditions.

Table 4.2.: Mapping requirements to system components and their parts

		R1 - Lightweight Views			R2 - View Modeling					R3 - Embed Queries		R4 - Maintenance		
		R1a - Nativeness	R1b - Memory-Efficiency	R1c - Match-Properties	R2a - Encapsulation	R2b - Effectiveness	R2c - Reusability	R2d - Nesting	R2e - Recursion	R3a - Languages	R3b - Transformations	R4a - Monitoring	R4b - Time-Efficiency	R4c - Propagation
A, B - Graph Storage	A - Base Graph Storage	●	○	○	○	○	○	○	○	○	○	○	○	○
	B - View Graph Storage	●	●	●	○	●	○	○	○	○	○	○	○	●
	A1 - Base Graph Schema	●	○	○	○	○	○	○	○	○	○	○	○	○
	B1 - View Graph Schema	●	◐	◐	○	◐	●	○	○	○	○	○	○	○
	A2 - Base Graph	●	○	○	○	○	○	○	○	○	○	○	○	○
	B2 - View Graph	●	●	●	○	●	○	○	○	○	○	○	○	●
C1 - View Module	C1 - View Module	○	○	○	●	○	◐	◐	○	●	●	○	○	○
	C2 - View Dependency	○	○	○	○	○	●	●	●	○	○	○	○	●
D - Graph Queries	D1 - Transformation LHS	○	○	○	○	○	○	○	○	●	●	○	○	○
	D2 - Transformation RHS	○	●	●	○	○	○	○	○	○	●	○	○	○
E - View Maintenance Engine	E1 - Batch Maintenance	○	●	●	○	○	○	○	○	●	●	○	○	●
	E2 - Incremental Maintenance	○	●	●	○	○	○	○	○	●	●	○	●	●
F - Graph Monitoring		○	○	○	○	○	○	○	○	○	○	●	◐	○
G - Query Engine		○	○	○	○	○	○	○	○	◐	◐	○	○	○

covered: ●; partially covered: ◐; not covered: ○

The view module dependency graph supports cyclic dependencies between view modules. These cycles can consist of multiple view modules and cycles can contain other cycles. Thus, the view module dependency graph supports recursive graph conditions (cf. Definition 15).

Moreover, the dependencies between view modules (C2) as well as the dependencies between graph nodes that mark graph pattern matches (B2) are used to propagate changes between view graphs (R4c - Propagation).

View Module Graph Transformation
View modules encapsulate the implementation details (R2a - Encapsulation) of queries (D). Therefore, different kinds of pattern matchings can be employed by queries (R3a - Languages).

View modules (C1) employ pattern matching to find matches for the left-hand sides of transformation rules (D1). The right-hand sides of the transformation rules (D2) create memory-efficient markings of graph pattern matches (R1b - Memory-Efficiency) (incl. additional match properties (R1c - Match-Properties)) in the view graph, when the left-hand sides of the transformation rules (D1) are satisfied. Furthermore, the transformation rules are used to delete and update graph nodes in view graphs, when matches do not satisfy patterns anymore or changed. Moreover, different graph transformation languages can be employed (R3b - Transformations), because view modules encapsulate queries (R2a - Encapsulation).

View Maintenance
The view maintenance engine (E) maintains graph nodes of view graphs that efficiently mark matches of graph patterns (R1b - Memory-Efficiency). This maintenance includes the maintenance of graph node attributes of view graphs (R1c - Match-Properties).

The framework prunes search spaces based upon modification events of base graphs (R4b - Time-Efficiency) and propagates changes of view modules to dependent view modules (R4c - Propagation) as described by the view module dependency graph (C2).

Moreover, the view maintenance engine provides a native realization, because the view modules employ pattern matching (R3a - Languages) and graph transformations (R3b - Transformations) to maintain graph nodes of view graphs.

Graph Monitoring
The framework employs a graph monitoring (F) to report added, deleted, and modified graph nodes and edges to the view maintenance engine (E). This monitoring of changes (R4a - Monitoring) enables the view maintenance engine to prune the search spaces of view modules during view graph maintenance (R4b - Time-Efficiency).

Query Engine
The query engine (G) enables end-users to retrieve matches of patterns (R3a - Languages) and to the modify base graphs with the help of graph transformations (R3b - Transformations).

Thus, all requirements are covered by the architecture of the proposed framework. The following chapters describe each component of the architecture in detail to show how the requirements are satisfied.

5. Graph Views

This chapter explains the notion of graph views and how they store graph pattern matches as result of graph queries efficiently and effectively. Section 5.1 describes how the framework stores entities and their relationships. Section 5.2 describes how graph views store graph pattern matches. Section 5.3 describes the notation of graph views. Section 5.4 discusses why the concepts satisfy the elicited requirements of Section 4.2.

5.1. Base Graphs

Base graphs store entities of the real world and their relationships. Base graphs are typed attributed graphs, which must adhere to an attributed type graph (cf. Section 2.1.1) that is called base graph schema. Note that also other kinds of typed graphs can be employed as base graphs such as hypergraphs with typed graph nodes and n-ary graph edges.

Type Graph

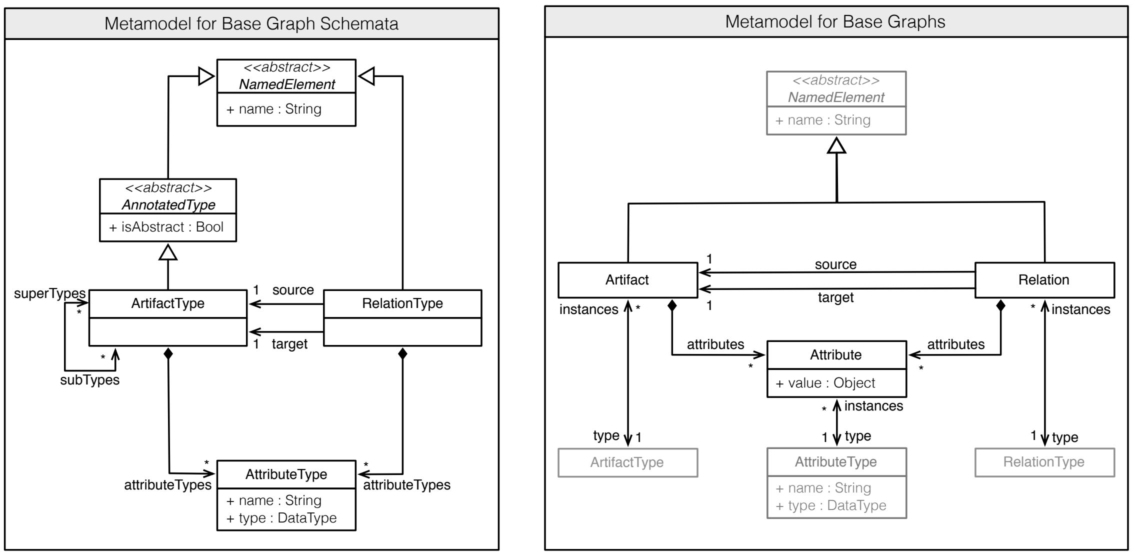
Base graph schemata describe which kinds of graph nodes and edges are stored by base graphs. This thesis employs the following terminology to refer to graph nodes and edges of base graph schemata. A graph node of a base graph schema is called artifact type and describes a kind of graph nodes that are stored by base graphs. A graph edge of a base graph schema is called relation type and describes a kind of graph edges that are stored by base graphs. A graph node and graph edge attribute of a base graph schema is called attribute type and describes a kind of graph node and graph edge attributes that are stored by base graphs.

Figure 5.1(a) depicts the metamodel that describes the concepts of base graph schemata. The metamodel describes that artifact types, relation types, and attribute types consists of distinguishable names that denote the purposes of these types. Artifact types constitute a type hierarchy with multiple inheritance. Artifact types can be abstract, i. e., cannot be instantiated in base graphs, but represent a common artifact supertype for a set of artifact types and their common properties. Relation types describe the direction of graph edges in base graphs as well as the kinds of graph nodes that are the source and target of these graph edges. Furthermore, artifact types and relation types own attribute types that describe the name and the data type of attribute values in base graphs.

Instance Graph

Base graphs are instances of base graph schemata. This thesis employs the following terminology to refer to graph nodes and edges of base graphs. A graph node of a base graph is called artifact and is an instance of a certain artifact type. A graph edge of a base graph is called relation and is an instance of a certain relation type. A graph node attribute and graph edge attribute of a base graph is called attribute and is an instance of a certain attribute type.

Figure 5.1(b) depicts the metamodel that describes the concepts of base graphs. The metamodel depicts classes that are already introduced in gray color. Artifacts and relations have a name and are instances of artifact types and relation types, respectively. Relations have



(a) Metamodel for base graph schemata (b) Metamodel for base graphs

Figure 5.1.: Metamodel for base graph schemata and base graphs as UML class diagrams

a direction and connect artifacts. Artifacts and relations own attributes that are instances of attribute types. These attributes store attribute values.

Note that artifact types and relation types serve as graph indexes, which enumerate all instances of a certain artifact type and relation type.

5.2. View Graphs

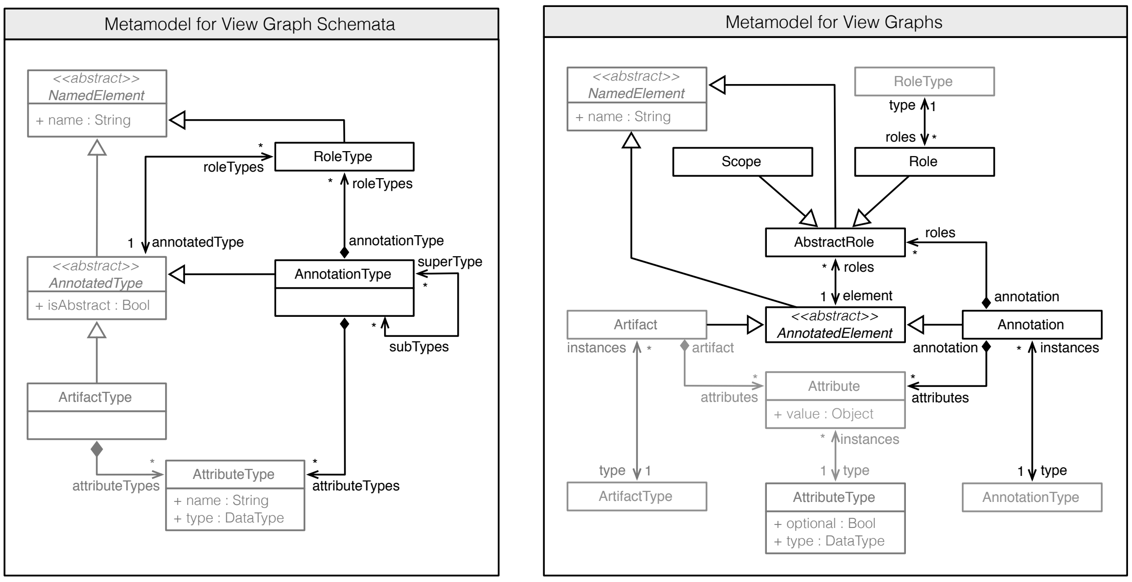
View graphs store markings of graph pattern matches effectively and efficiently. View graphs are typed attributed graphs, which must adhere to an attributed type graph (cf. Section 2.1.1) that is called view graph schema.

Type Graph

View graph schemata describe which kinds of graph pattern matches are stored by view graphs and which kinds of graph nodes belong to these matches. Furthermore, they describe kinds of graph edges that are used to denote the role of graph nodes in these matches.

This thesis employs the following terminology to refer to graph nodes and edges of view graph schemata. A graph node of a view graph schema is called annotation type and describes a kind of graph pattern matches that are stored by view graphs. A graph edge of a view graph schema is called role type and describes a kind of roles that graph nodes can have in graph pattern matches. A graph node and graph edge attribute of a view graph schema is called attribute type and describes a kind of attributes that can enrich graph pattern matches with additional properties.

Figure 5.2(a) depicts the metamodel that describes the concepts of view graph schemata. The metamodel extends the metamodel for base graph schemata and depicts parts that are already introduced in gray color. Annotation types, role types, and attribute types consist of distinguishable names that describe the purpose of these types. Annotation types constitute a type hierarchy with multiple inheritance. Annotation types can be abstract, i. e., cannot be instantiated in view graphs, but represent a common annotation supertype for a set of annotation types that have properties in common. Role types describe kinds of directed graph edges, which are used in view graphs to mark the role of the graph nodes that belong to graph pattern matches. The source of a role type is always an annotation type. The target of a role type is either an artifact type or an annotation type. Thus, graph nodes of base graphs and view graphs can be part of graph pattern matches. Furthermore, annotation types can own attribute types that describe the name and data type of attribute values in view graphs. Annotation types hand their attribute types down to their subtypes.



(a) Metamodel for view graph schemata (b) Metamodel for view graphs

Figure 5.2.: Metamodel for view graph schemata and view graphs as UML class diagram

Instance Graph

View graphs are instances of view graph schemata. View graphs store graph nodes that represent graph pattern matches and mark the graph nodes, which belong these matches, by means of graph edges. The types of the graph nodes in view graphs describe the kinds of the marked graph pattern matches. The types of the graph edges in view graphs describe the roles of graph nodes that belong to graph pattern matches.

This thesis employs the following terminology to refer to graph nodes and edges of view graphs. A graph node of a view graph is called annotation and is an instance of a certain annotation type. View graphs distinguish two kinds of graph edges. One kind of graph edges is called role and is an instance of a certain role type. The other kind of graph edges is called scope and consist of a default type, because the graph nodes that are marked by scopes do not have to be distinguishable. A graph node attribute and graph edge attribute of a view graph is called attribute and is an instance of a certain attribute type.

Figure 5.2(b) depicts the metamodel, which describes the concepts of view graphs. The metamodel extends the metamodel for base graphs and depicts classes that are already introduced in gray color. Annotations are instances of annotation types and represent markings of graph pattern matches. For that purpose, annotations own roles and scopes, which reference artifacts or other annotations that satisfy a certain graph pattern. Roles are instances of role types. These role types denote the role of artifacts or annotations in graph pattern matches. In contrast to roles, scopes reference artifacts or other annotations that have no special role in graph pattern matches. Therefore, scopes have a default type that is not explicitly represented. Furthermore, annotations can own attributes that are instances of attribute types. Attributes of annotations can store data values to enrich the graph pattern matches, which are marked by annotations, with additional knowledge.

5.3. Notation

The following sections describe the notation of base graph schemata and view graph schemata as well as base graphs and view graphs.

5.3.1. Type Graphs

This thesis adapts UML class diagrams as concrete syntax to describe artifact types, annotation types, relation types, role types, and attribute types. Artifact types and annotation types map to UML classes. The type hierarchy of artifact types and annotation types maps to UML generalizations. Abstract artifact types and annotation types map to abstract UML classes. Relation types and role types map to UML associations. Attribute types map to attributes of UML classes and UML associations.

The notation employs the «artifact type» and «annotation type» stereotypes to distinguish artifact types of base graph schemata and annotation types of view graph schemata. Furthermore, the notation employs the «relation type» and «role type» stereotypes to distinguish relation types of base graph schemata and role types of view graph schemata.

Base Graph Schema

According to the running example, Figure 5.3 depicts an excerpt of the base graph schema. The UML class diagram shows the abstract Classifier and Member artifact types. The Classifier and Member artifact types consist of a name attribute type to describe that instances of the Classifier and Member artifact types have a name. The UML class diagram depicts the Class and Interface artifact types as specializations of the Classifier artifact type. Furthermore, the UML class diagram depicts the Field and Method artifact types as specializations of the Member artifact type. The Classifier artifact type owns the members relation type that targets at the Member artifact type to describe that classifiers own members. For the sake of simplicity, Figure 5.3 omits the remaining artifact and relation types of the running example.

View Graph Schema

According to the running example, Figure 5.3 depicts an excerpt of the view graph schema. The UML class diagram depicts the abstract DesignPattern and DesignAntiPattern annotation types. The Composite annotation type is a specialization of the DesignPattern annotation type. The ExtractInterface annotation type is a specialization of the DesignAntiPattern annotation type. Furthermore, the UML class diagram depicts annotation types for low-level kinds of graph patterns such as the abstract Hierarchy and Association annotation types.

The Hierarchy annotation type owns the Super and Sub role types that target at the Classifier artifact type to describe that instances of the Hierarchy annotation type mark instances of the Classifier artifact type as superordinate and subordinate classifier. Additionally, the Hierarchy annotation type owns a levels attribute type to describe that instances of the Hierarchy annotation type store the number of hierarchy levels between the superordinate and subordinate classifier. According to the running example, the Generalization annotation type

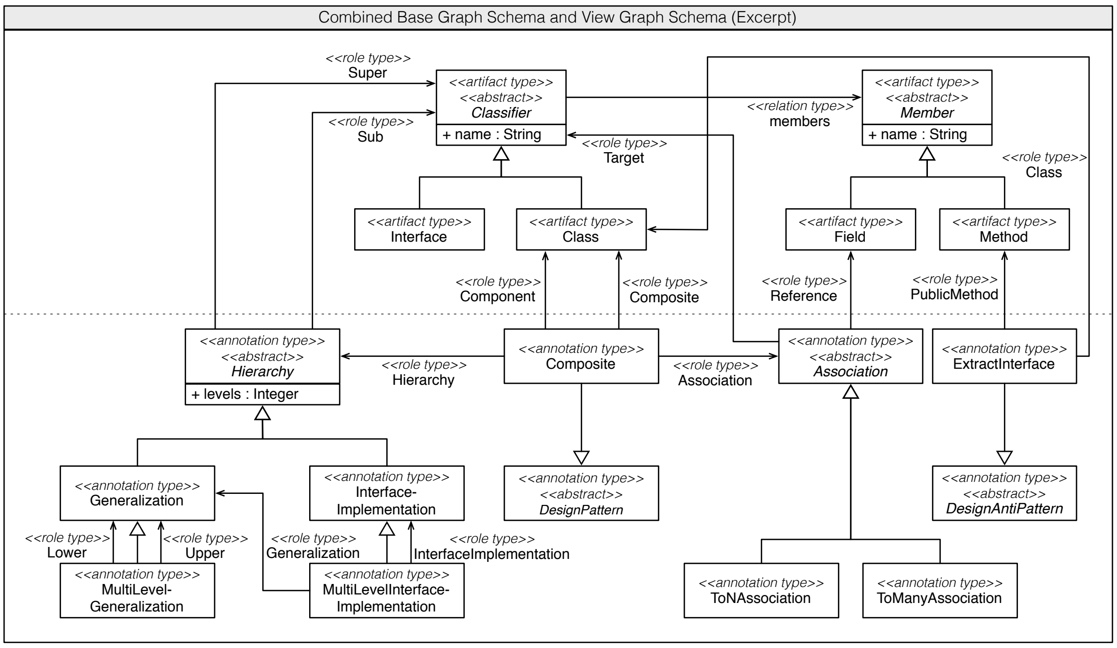

Combined Base Graph Schema and View Graph Schema (Excerpt)
<<role type>> Super
<<role type>> Sub
<<artifact type>> <<abstract>> Classifier + name : String
<<role type>> Target
<<relation type>> members
<<artifact type>> <<abstract>> Member + name : String
<<role type>> Class
<<artifact type>> Interface
<<artifact type>> Class
<<artifact type>> Field
<<artifact type>> Method
<<role type>> Component
<<role type>> Composite
<<role type>> Reference
<<role type>> PublicMethod
<<annotation type>> <<abstract>> Hierarchy + levels : Integer
<<role type>> Hierarchy
<<annotation type>> Composite
<<role type>> Association
<<annotation type>> <<abstract>> Association
<<annotation type>> ExtractInterface
<<annotation type>> Generalization
<<annotation type>> Interface-Implementation
<<annotation type>> <<abstract>> DesignPattern
<<annotation type>> <<abstract>> DesignAntiPattern
<<role type>> Lower
<<role type>> Upper
<<role type>> Generalization
<<role type>> InterfaceImplementation
<<annotation type>> MultiLevel-Generalization
<<annotation type>> MultiLevelInterface-Implementation
<<annotation type>> ToNAssociation
<<annotation type>> ToManyAssociation

Figure 5.3.: Excerpt of the base graph schema (top) and the view graph schema (bottom) as UML class diagram according to the running example

and InterfaceImplementation annotation type are specializations of the Hierarchy annotation type. Both annotation types inherit the role types and the attribute types from the Hierarchy annotation type. Thus, instances of the Generalization annotation type mark matches of the Generalization pattern (cf. Figure 3.5(a)) and mark the super- and subclass in the matches. Instances of the InterfaceImplementation annotation type mark matches of the Interface Implementation pattern (cf. Figure 3.5(b)) and mark the interface and the class that implements the interface in the matches. The MultiLevelGeneralization annotation type is a specialization of the Generalization annotation type. Instances of the MultiLevelGeneralization annotation type mark matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)). Additionally, the MultiLevelGeneralization annotation type owns the Lower and Upper role types. The Lower and Upper role types describe that instances of the MultiLevelGeneralization annotation type mark instances of the Generalization annotation type as lower and upper generalization of a multi-level generalization. Note that the lower and upper generalizations can be multi-level generalizations as well, because the MultiLevelGeneralization annotation type is a specialization of the Generalization annotation type. The MultiLevelInterfaceImplementation annotation type is a specialization of the InterfaceImplementation annotation type and, additionally, owns the Generalization and InterfaceImplementation role types. These role types describe that instances of the MultiLevelInterfaceImplementation annotation type reference instances of the Generalization and InterfaceImplementation annotation types to mark the (multi-level) generalization and interface implementation that constitute a multi-level interface implementation.

The Association annotation type has two specializations called ToManyAssociation and ToNAssociation annotation types. Instances of the ToManyAssociation annotation type mark matches of the ToMany Association pattern (cf. Figure 3.6(c)). Instances of the ToNAssociation annotation type mark matches of the ToN Association pattern (cf. Figure 3.6(b)). The ToManyAssociation and ToNAssociation annotation types inherit the Reference and Target role types from the Association annotation type. These role types describe that instances of the Association annotation type reference instances of the Field and Classifier artifact types to mark fields as associations with certain target types.

The Composite annotation type owns the Hierarchy, Association, Composite, and Component role types. The Hierarchy and Association role types describe that instances of the Composite annotation type mark instances of the Hierarchy and Association annotation types to describe which matches of the Hierarchy and Association patterns constitute a match of the Composite pattern. The Composite and Component role types describe that instances of the Composite annotation type reference instances of the Classifier artifact type to mark which classifiers act as composite and component in the detected Composite software design pattern, respectively.

The ExtractInterface annotation type owns the PublicMethod and Class role types, which describe that instances of this annotation type reference instances of the Method and Class artifact types to mark for which method and class an interface should be extracted.

5.3.2. Instance Graphs

This thesis adapts UML object diagrams as concrete syntax to describe base graphs and view graphs. Artifacts and annotations map to UML objects. Relations and roles map to UML links. Attributes of artifacts and annotations map to attributes of UML objects. Attributes of relations map to attributes of UML links.

This thesis depicts graph nodes and edges of base graphs and view graphs differently. UML objects with solid border depict artifacts of base graphs. UML links with solid lines between UML objects depict relations of base graphs. UML objects with dashed rounded border depict annotations of view graphs. UML links with dashed lines depict roles of view graphs. UML links with dotted lines depict scopes.

Base Graph

According to the running example, Figure 5.4 depicts an ASG as base graph. The base graph shows that a child class extends a parent class, which is located in a certain namespace. Furthermore, the child class owns an one-dimensional field that stores instances of the parent class. Moreover, the parent class owns a method with a public visibility.

View Graph

According to the running example, Figure 5.4 depicts annotations that represent matches of the Generalization, ToN Association, Composite, and Extract Interface pattern.

The generalization annotation with Generalization type owns roles with Super and Sub role types, which mark that the parent class acts as superclass and the child class acts as subclass in a generalization. Furthermore, this annotation owns two scopes, which mark that the namespace1 and reference1 artifacts are part of the match as well. Additionally, this annotation owns a levels attribute, which describes that one hierarchy level between the super- and subclass exists.

The association annotation with ToNAssociation type owns roles with Reference and Target role types, which mark that the children field acts as reference and the parent class acts as target of the reference in a bounded association. Furthermore, this annotation owns three scopes, which mark the dimension, namespace2, and reference2 artifacts as part of the match.

The composite annotation with Composite type owns roles with Hierarchy, Association, Composite, and Component role types. The role with Hierarchy role type targets at the generalization annotation to mark the match of the Generalization pattern that is reused to find the match of the Composite pattern. The role with Association role type targets at the

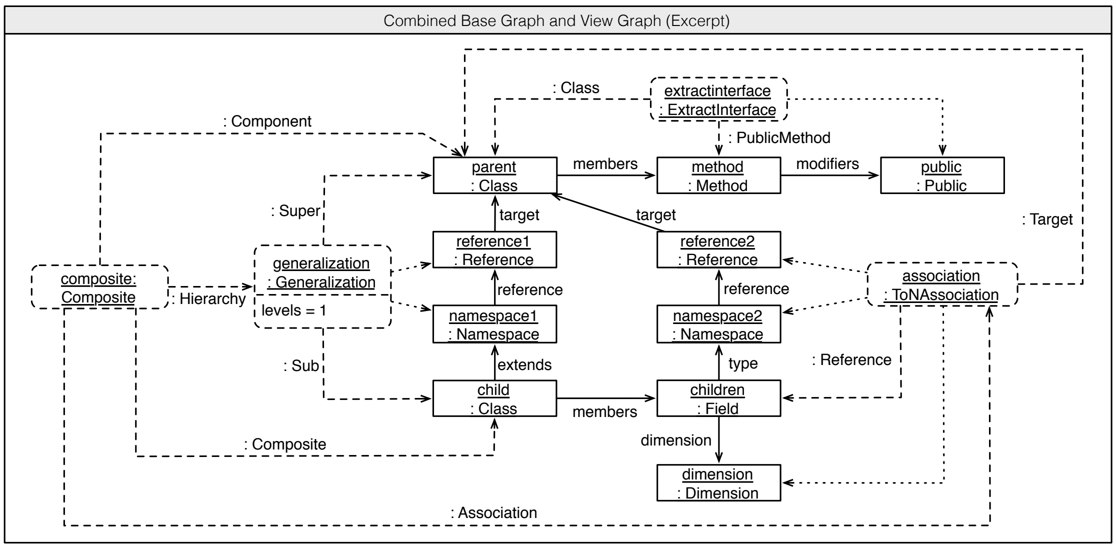


Figure 5.4.: Excerpt of the base graph (solid elements) and the view graph (dashed and dotted elements) as UML object diagram according to the running example

association annotation to mark the match of the ToN Association pattern that is reused to find the match of the Composite pattern. The role with Composite role type marks the child class as composite of the Composite design pattern. The role with Component role type marks the parent class as component of the Composite design pattern. Note that the running example omits the Leaf class of the Composite design pattern (cf. Section 3.1) for the sake of simplicity, because this class is an optional part of the Composite design pattern.

The extractinterface annotation with ExtractInterface type owns roles with Class role type and PublicMethod role type. The role with Class role type targets at the parent class to mark that for this class an interface should be extracted. The role with PublicMethod role type targets at the method artifact to mark that for this method an interface should be extracted. Furthermore, this annotation owns one scope that marks the public artifact, because this artifact belongs to the match as well.

5.4. Discussion

This chapter describes the notion of base graphs and view graphs that must adhere to base graph and view graph schemata. Base graph schemata are attributed type graphs, which describe the kinds of entities and relationships between these entities that are stored by base graphs. View graph schemata are attributed type graphs that describe the kinds of graph pattern matches that are represented by graph nodes of view graphs and the kinds of graph edges that mark which graph nodes satisfy the graph pattern. These kinds of graph edges denote the roles of graph nodes in matches.

Table 5.1 summarizes which kinds of type graphs and instance graphs satisfy the elicited requirements of Section 4.2. The requirement R1a - Nativeness describes that graphs must be stored natively by using a graph data model. The requirement R1b - Memory-Efficiency describes that matches must be stored efficiently, i. e., without copying graph nodes and edges from base graphs to view graphs. The requirement R1c - Match-Properties describes that stored graph pattern matches must enable developers to store and maintain additional

Table 5.1.: Mapping requirements to type graphs and instance graphs

		Requirements		
		R1a - Nativeness	R1b - Memory-Efficiency	R1c - Match-Properties
Type Graphs	Base Graph Schema	●	○	○
	View Graph Schema	●	●	●
Instance Graphs	Base Graph	●	○	○
	View Graph	●	●	●

covered: ●; partially covered: ◐; not covered: ○

attribute values to enrich views with additional domain knowledge that is derived from the graph nodes and edges of the matches.

In this thesis, the base graph and view graph schemata as well as base graphs and view graphs are natively represented as attributed type graphs and typed attributed graphs, respectively. Thus, the base graph and view graph schemata as well as the base graphs and view graphs satisfy the requirement R1a - Nativeness.

The view graph schemata describe which kinds of graph pattern matches are marked by graph nodes and edges of view graphs. View graphs store markings for graph pattern matches. Therefore, view graphs do not store copies of graph nodes and edges of base graphs and view graphs. Consequently, these graph nodes and edges do not have to be kept synchronized with the original graph nodes and edges. Thus, the view graph schemata and the view graphs satisfy the requirement R1b - Memory-Efficiency.

The view graph schemata enable developers to define attribute types of annotation types. These attribute types describe kinds of attributes that can be stored by annotations to enrich matches with additional domain knowledge. Thus, the view graph schemata and view graphs satisfy the requirement R1c - Match-Properties.

6. View Definition Language

This chapter describes a modeling language that enables developers to model generalized discrimination networks (cf. Section 1.3.1). These networks describe which kinds of view graphs are initially created and, afterwards, maintained. Furthermore, these networks describe which view graphs build on the content of other view graphs.

This chapter starts with a description of the modeling methodology that developers employ to model generalized discrimination networks. Section 6.1 gives an overview of the modeling methodology. Then, the subsequent sections describe the modeling activities of the modeling methodology in detail. Section 6.2 describes the modeling language for view modules that represent network nodes of the discrimination networks. Section 6.3 describes the modeling language for view module dependencies that represent network edges between network nodes of the discrimination networks. Section 6.4 discusses the expressiveness of the modeling language. Finally, Section 6.5 discusses the satisfaction of the elicited requirements.

6.1. Modeling Methodology

The developers employ the following modeling methodology to define view graphs. First, the developers model base graph schemata, which describe the kinds of graph nodes and edges that are stored by base graphs. Then, the developers model view graph schemata, which describe the kinds of graph pattern matches that are maintained by view graphs. Afterwards, the developers define the view modules that are responsible to initially find and continuously maintain these kinds of matches. Then, the developers define the dependencies between these modules to describe the reuse of matches by other modules. Finally, the developers embed graph queries into these modules to search for matches and mark these matches accordingly. The following sections describe the required steps during each modeling activity.

Modeling the Base Graph Schemata

The developers think about which kinds of graph nodes and edges are stored in base graphs. For that purpose, developers define artifact types and relation types by means of a base graph schema as described in Section 5.1.

Modeling the View Graph Schemata

The developers think about which kinds of graph pattern matches are stored and maintained by view graphs. For that purpose, developers create annotation types and role types by means of a view graph schema as described in Section 5.2. Developers create an annotation type for each kind of graph pattern match that is stored and maintained by view graphs. Thereby, the developers think about the decomposition of graph patterns into partial patterns to enable the reuse of (partial) patterns, when defining view graphs. Afterwards, the developers think about which kinds of graph nodes of graph pattern matches must be effectively accessible, when retrieving matches for a certain kind of (partial) graph pattern from the view graphs. For each kind of graph node that must be effectively accessible, the developers create a role type that is owned by the annotation type, which describes the kind of the graph pattern match.

These role types either target at artifact types of base graph schemata, when artifacts with a certain artifact type participate in a match, or annotation types of view graph schemata, when matches of certain graph patterns are reused to find other graph pattern matches.

Modeling the View Modules

The developers model view modules that search for matches of graph patterns and instantiate annotation types and role types, when they find graph pattern matches. The developers describe an interface for each module. The developers describe which kinds of artifacts and annotations are required by the module and which kinds of annotations are created and maintained by the module. For that purpose, the developers employ the artifact types and annotation types of base graph schemata and view graph schemata.

Modeling the View Module Dependencies

After developers defined view modules, they describe the reuse of graph pattern matches by other view modules. For that purpose, developers model dependencies between these modules. These dependencies describe the flow of annotations between modules. By creating these dependencies, the developers compose a directed view module dependency graph. The module dependency graph can be cyclic. This dependency graph enables the developers to describe conjunctions, disjunctions, negations, and recursive definitions of view graphs.

Embedding Graph Queries

The developers embed graph queries into each view module in a graph query language of their choice to implement the search for graph pattern matches and the marking of these matches. When the graph queries find matches, they create annotations and roles in the view graphs to mark these matches.

6.2. View Module

The developers define view modules, which encapsulate graph queries, to enable the framework to handle these graph queries uniformly. When creating a module, the developers define an interface for the encapsulated graph query. This interface describes the kinds of artifacts and annotations that are required by the graph query during the graph pattern matching. Furthermore, this interface describes the kind of annotations that are created in the view graph, when the graph query finds a match.

Figure 6.1 depicts the metamodel for view modules. The metamodel depicts parts of the metamodel that are already introduced in gray color. The metamodel describes that view modules have a name and a description. View modules own artifact connectors and annotation connectors. These connectors have a name and a type. Artifact connectors receive artifacts with a certain artifact type, which are processed by the encapsulated query. Annotation connectors receive or provide annotations with a certain annotation type. Input connectors are annotation connectors, which describe the kind of annotations that are processed by the encapsulated query. Output connectors are annotation connectors, which describe the kind of annotations that are created by the encapsulated query. Moreover, artifact connectors and input connectors can be negative to describe that the encapsulated query checks for the absence of artifacts and annotations with certain types.

Figure 6.1 depicts view modules in concrete syntax. This thesis depicts modules as rounded rectangles that are labeled with the module name. Connectors are attached to modules. This thesis depicts artifact connectors as rectangles that contain a filled black square and are

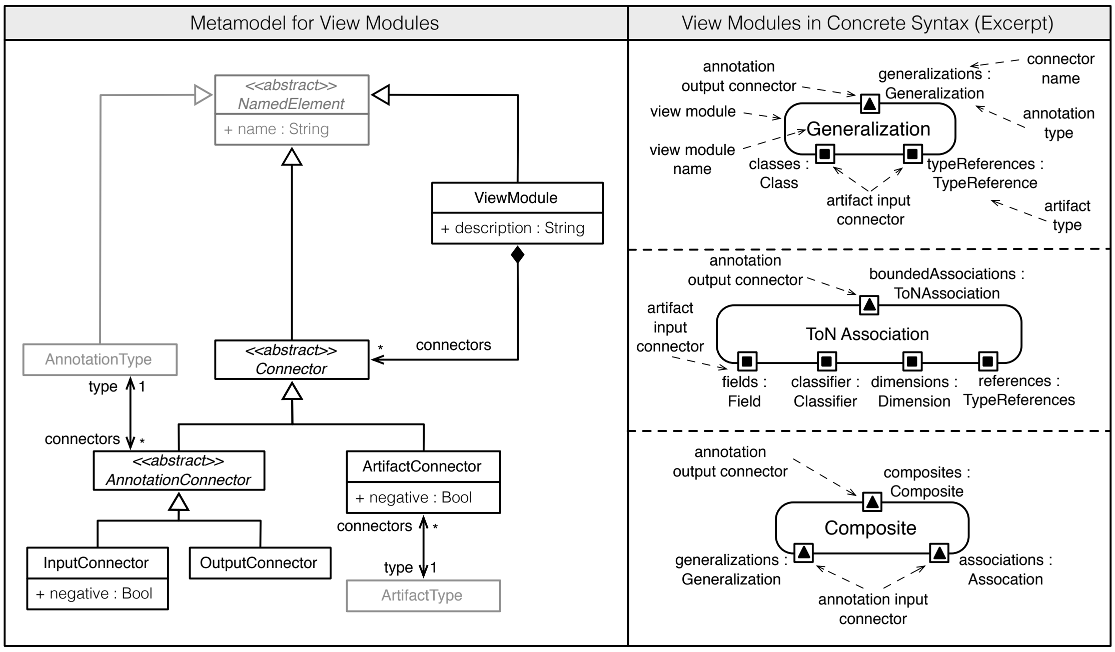


Figure 6.1.: Metamodel for view modules (left) and concrete syntax of view modules (right)

labeled with the name and artifact type separated by a colon. Negative artifact connectors have a black background and consist of a filled white rectangle. This thesis depicts annotation connectors as rectangles that contain a filled black triangle and are labeled with the name and annotation type separated by a colon. Input connectors consist of a triangle that points to the module. Output connectors consist of a triangle that points away from the module. Negative input connectors have a black background and consist of a filled white triangle.

Figure 6.1 depicts the view modules Generalization, ToN Association, and Composite. According to the Generalization pattern (cf. Figure 3.5(a)), the Generalization module owns two artifact connectors that receive artifacts with Class and TypeReference artifact type. The TypeReference artifact type is the common supertype of the Namespace and Reference artifact type. The Generalization module owns an output connector, which provides annotations with Generalization annotation type that mark matches of the Generalization pattern.

According to the ToN Association pattern (cf. Figure 3.6(b)), the ToN Association module owns four artifact connectors and one output connector. The artifact connectors describe that the module receives artifacts with Field, Classifier, Dimension, and TypeReference artifact type. The output connector provides annotations with ToNAssociation annotation type, which mark matches of the ToN Association pattern.

According to the Composite pattern (cf. Figure 3.8(a)), the Composite module owns two input connectors and one output connector. The input connector receives annotations with Generalization and Association annotation type, respectively. Thus, the Composite module reuses matches of the Generalization and ToN Association pattern. The output connector provides annotations with Composite type, which mark matches of the Composite pattern.

Figure 6.2 shows the Extract Interface module. According to the Extract Interface pattern (cf. Figure 3.9), the Extract Interface module owns three artifact connectors, one negative input connector, and one output connector. The artifact connectors receive artifacts with Class, Method, and Public artifact type. The negative input connector receives annotations

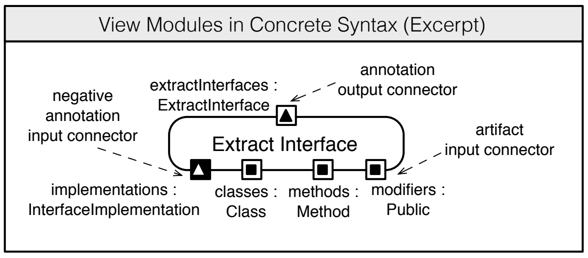

View Modules in Concrete Syntax (Excerpt)
negative annotation input connector
extractInterfaces : ExtractInterface
annotation output connector
Extract Interface
artifact input connector
implementations : InterfaceImplementation
classes : Class
methods : Method
modifiers : Public

Figure 6.2.: Extract Interface module in concrete syntax

with InterfaceImplementation annotation type. The negative input connector describes that the encapsulated graph query checks for the absence of annotations that mark matches of the Interface Implementation pattern (cf. Figure 3.5(b)). The output connector provides annotations with Extract Interface type that mark matches of the Extract Interface pattern.

6.3. View Dependency Graph

When view modules build on the graph pattern matches that are maintained by other modules, they need to exchange these matches by means of annotations. The framework enables developers to model this reuse of annotations by means of dependencies between modules. The resulting discrimination network is called view dependency graph and shows how view graphs are derived from other view graphs. This dependency graph enables developers to model conjunctions, disjunctions, negations, and recursive definitions of view graphs.

Figure 6.3 shows an extension of the metamodel for view modules. The metamodel depicts parts that are already introduced in gray color. The metamodel describes that view dependency graphs are specializations of view modules that contain other view modules and dependencies between these view modules. Therefore, a view dependency graph is also a view module that consists of connectors. The metamodel distinguishes module dependencies and graph dependencies. Module dependencies are directed links that connect the output connectors of view modules with the input connectors of view modules within a view dependency graph. Graph dependencies are directed links that connect input connectors of view dependency graphs with input connectors of view modules as well as output connectors of view modules with the output connectors of view dependency graphs.

According to the running example, Figure 6.3 shows a view module dependency graph in concrete syntax that depicts dependencies as directed solid lines. The arrow heads denote the dependent view modules. The dependency graph consists of the Generalization, ToN Association, and Composite modules. The dependency between the Generalization and Composite module describes that the Composite module receives graph pattern matches in terms of annotations that are created by the Generalization module. The dependency between the ToN Association and the Composite module describes that the Composite module receives graph pattern matches in terms of annotations that are created by the ToN Association module.

Figure 6.4 shows the Hierarchy view dependency graph. The dependency graph consists of the Generalization, Multi-Level Generalization, Interface Implementation, and Multi-Level Interface Implementation modules. The Generalization module provides annotations that mark matches of the Generalization pattern (cf. Figure 3.5(a)). The Interface Implementation module provides annotations that mark matches of the Interface Implementation pattern (cf. Figure 3.5(b)).

The Multi-Level Generalization module receives annotations from the Generalization module

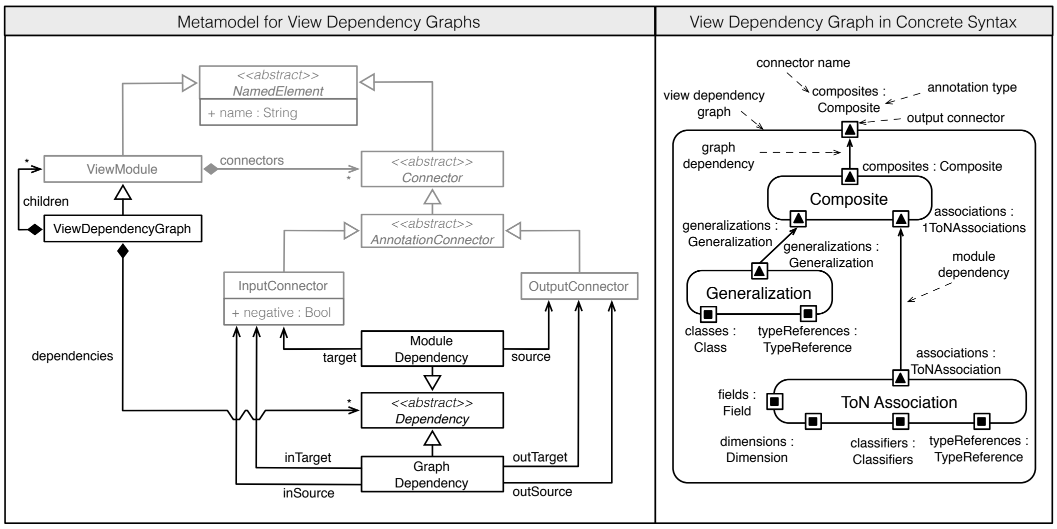


Figure 6.3.: Metamodel (left) and concrete syntax (right) of view dependency graphs

to search for matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)). Furthermore, the Multi-Level Generalization module receives annotations from itself to find multi-level generalizations that base on other multi-level generalizations. For that purpose, a module dependency connects the output connector of the Multi-Level Generalization module with the input connector of the Multi-Level Generalization module.

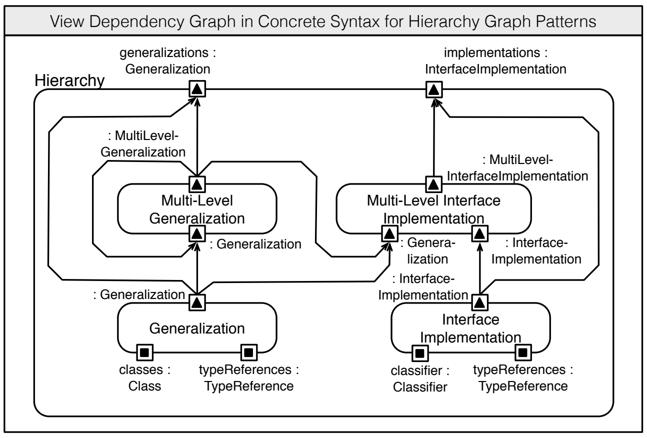

View Dependency Graph in Concrete Syntax for Hierarchy Graph Patterns
generalizations : Generalization
implementations : InterfaceImplementation
Hierarchy
: MultiLevel-Generalization
: MultiLevel-InterfaceImplementation
Multi-Level Generalization
Multi-Level Interface Implementation
: Generalization
: Genera-lization
: Interface-Implementation
: Generalization
: Interface-Implementation
Generalization
Interface Implementation
classes : Class
typeReferences : TypeReference
classifier : Classifier
typeReferences : TypeReference

Figure 6.4.: View modules for Hierarchy patterns in concrete syntax

The Multi-Level Interface Implementation module receives annotations with Generalization annotation type from the Generalization module and Multi-Level Generalization module as well as annotations with InterfaceImplementation annotation type from the Interface Implementation module to find matches of the Multi-Level Interface Implementation pattern (cf. Figure 3.5(d)).

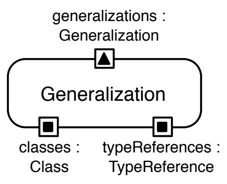
Moreover, the Generalization and Multi-Level Generalization modules provide annotations that mark matches of the Generalization patterns to the generalizations output connector of the dependency graph. The Interface Implementation and Multi-Level Interface Implementation modules provide annotations that mark matches of the Interface Implementation patterns to the implementations output connector of the dependency graph.

6.4. Expressiveness of the View Definition Language

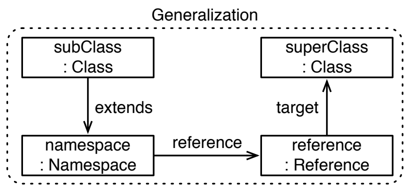
This section describes how developers employ view modules and module dependencies to model conjunctions, disjunctions, negations, and recursive definitions of graph conditions (cf. Definition 14). Section 6.4.1 describes modules that encapsulate atomic graph conditions. Section 6.4.2 describes modeling variants for modules that encapsulate conjunctions of graph conditions. Section 6.4.3 describes modules that encapsulate disjunctions of graph conditions. Section 6.4.4 describes modules that encapsulate negations of graph conditions. Section 6.4.5 describes cyclic dependencies between modules that encapsulate recursive graph conditions.

6.4.1. Atomic Graph Conditions

View modules encapsulate atomic graph conditions, when they own artifact input connectors, but do *not* own annotation input connectors.



(a) Generalization module



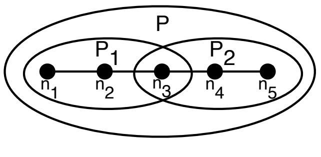
(b) Generalization pattern

Figure 6.5.: Atomic graph condition

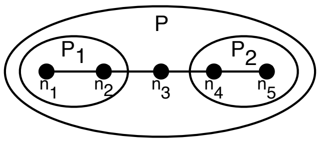
According to the running example, Figure 6.5 shows the Generalization module and the Generalization pattern. The Generalization module owns two artifact input connectors with Class and TypeReference artifact type and does *not* own annotation input connectors. Therefore, the Generalization module encapsulates the Generalization pattern as atomic graph condition. Note, the TypeReference type is the common supertype of the Namespace and Reference type.

6.4.2. Conjunctions

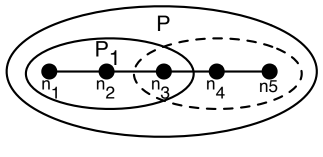
View modules encapsulate conjunctions of graph conditions, when they own annotation input connectors and, optionally, artifact connectors. These connectors enable developers to model conjunctions of graph conditions in three different ways. View modules encapsulate conjunctions, when graph patterns either a) overlap, b) are disjoint, or c) are extended.



(a) Overlapping patterns



(b) Disjoint patterns



(c) Extended pattern

Figure 6.6.: Conjunction of graph patterns

Figure 6.6 gives an overview of these variants. Figure 6.6(a) describes that two patterns, which overlap in one or more pattern nodes, implement a conjunction, because the graph nodes that match the pattern nodes in the intersection of both patterns satisfy both patterns.

According to Figure 6.6(a), the pattern P_1 consists of the pattern nodes n_1, n_2, and n_3. The pattern P_2 consists of the pattern nodes n_3, n_4, and n_5. The intersection of pattern P_1 and P_2 consists of the pattern node n_3. Thus, the graph nodes, which match pattern node n_3 in the overall pattern P, satisfy pattern P_1 and P_2.

Figure 6.6(b) describes that two disjoint patterns implement a conjunction, when the overall pattern employs additional pattern nodes and edges that connect both disjoint patterns in a manner that the overall pattern is a connected graph.

According to Figure 6.6(b), the pattern P_1 consists of the pattern nodes n_1 and n_2. The pattern P_2 consists of the pattern nodes n_4 and n_5. The intersection of pattern P_1 and P_2 is empty. The pattern node n_3, the pattern edge between pattern node n_2 and n_3, and the pattern edge between pattern node n_4 and n_3 connect the pattern P_1 and P_2. Therefore, matches of the overall pattern P satisfy the embedded patterns P_1 and P_2.

Figure 6.6(c) depicts that a pattern, which is extended by additional pattern nodes and edges, implements a conjunction, when the pattern and the additional pattern nodes and edges constitute a connected graph, because the additional pattern nodes and edges constitute a second *implicit* pattern that overlaps with the *extended* pattern.

According to Figure 6.6(c), the pattern P_1 consists of the pattern nodes n_1, n_2, and n_3. The pattern nodes n_4 and n_5 are directly or indirectly connected to pattern node n_3 of pattern P_1. Therefore, the pattern nodes n_3, n_4, and n_5 constitute an implicit pattern that overlaps with pattern P_1. Thus, matches of the pattern P satisfy the pattern P_1 and the implicit pattern.

The following sections describe each case in detail and show modeling variants for each case.

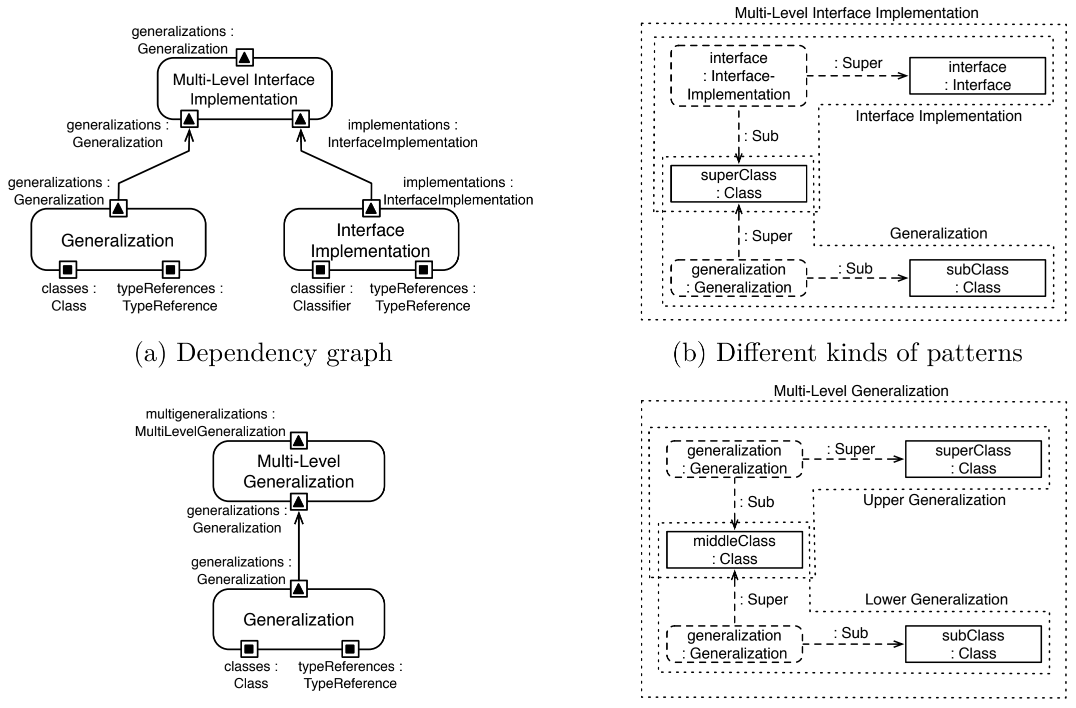
Overlapping Graph Patterns

This thesis distinguishes two kinds of overlapping patterns. The first kind deals with overlapping patterns of *different* kinds. View modules that encapsulate conjunctions with overlapping patterns of different kinds own two or more annotation input connectors, which receive annotations with different annotation types, and do not own artifact input connectors.

The second kind deals with overlapping patterns of the *same* kind. View modules that encapsulate conjunctions with overlapping patterns of the same kind own one annotation input connector, which receives annotations with a certain annotation type, and do *not* own artifact input connectors. The following examples describe both modeling variants.

According to the running example, Figure 6.7(a) shows the Multi-Level Interface Implementation module, which encapsulates a conjunction that deals with *different* kinds of patterns. This module owns two annotation input connectors with Generalization and InterfaceImplementation annotation type, respectively. Therefore, the module describes that it encapsulates a conjunction of the Generalization pattern (cf. Figure 3.5(a)) and Interface Implementation pattern (cf. Figure 3.5(b)). Figure 6.7(b) depicts the Multi-Level Interface Implementation pattern (cf. Figure 3.5(d)) and describes that both patterns overlap in the superclass pattern node, which describes that the generalization implements an interface.

According to the running example, Figure 6.7(c) shows the Multi-Level Generalization module, which encapsulates a conjunction that deals with the *same* kind of pattern. The Multi-Level Generalization module owns one annotation input connector with Generalization annotation type. Therefore, the module can encapsulate a conjunction that consists of two or more Generalization patterns (cf. Figure 3.5(a)). Figure 6.7(d) depicts the Multi-Level Generalization pattern (cf. Figure 3.5(c)) that employs two Generalization patterns. Both Generalization patterns overlap in the middleclass pattern node, which describes that the superclass of the lower generalization and the subclass of the upper generalization are the same.


generalizations : Generalization
Multi-Level Interface Implementation
generalizations : Generalization
implementations : InterfaceImplementation
generalizations : Generalization
implementations : InterfaceImplementation
Generalization
Interface Implementation
classes : Class
typeReferences : TypeReference
classifier : Classifier
typeReferences : TypeReference
Multi-Level Interface Implementation
interface : Interface-Implementation
: Super
interface : Interface
Interface Implementation
: Sub
superClass : Class
: Super
Generalization
generalization : Generalization
: Sub
subClass : Class
multigeneralizations : MultiLevelGeneralization
Multi-Level Generalization
generalizations : Generalization
generalizations : Generalization
Generalization
classes : Class
typeReferences : TypeReference
Multi-Level Generalization
generalization : Generalization
: Super
superClass : Class
: Sub
Upper Generalization
middleClass : Class
: Super
Lower Generalization
generalization : Generalization
: Sub
subClass : Class

(a) Dependency graph

(b) Different kinds of patterns

(c) Dependency graph

(d) Same kinds of patterns

Figure 6.7.: Conjunction with overlapping graph patterns

Disjoint Graph Patterns

Graph patterns are disjoint, when they have no pattern nodes in common. View modules implement conjunctions with disjoint patterns, when they own one or more annotation input connectors and *may* own one or more artifact input connector. This thesis distinguishes two modeling variants. The first modeling variant employs only additional pattern edges to connect disjoint patterns. The second modeling variant employs additional pattern nodes and edges to connect disjoint patterns. The following examples describe both variants.

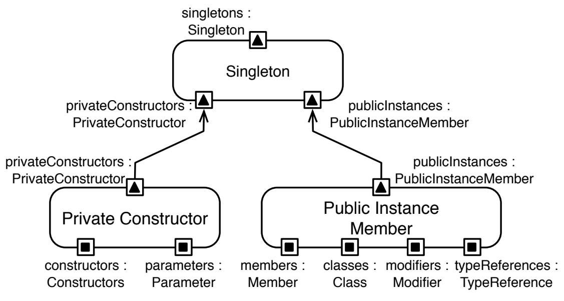
According to the running example, Figure 6.8(a) shows a Singleton module that searches for employed Singleton design patterns[1]. The Singleton module owns two annotation input connectors with PrivateConstructor and PublicInstanceMember annotation type, respectively. Therefore, the module encapsulates a conjunction of the PrivateConstructor pattern[2] and PublicInstanceMember pattern[3]. Appendix F describes these patterns. Figure 6.8(b) depicts that both patterns are disjoint. Thus, the overall pattern employs an additional members pattern edge that connects the PrivateConstructor and PublicInstanceMember patterns in a way that the constructor is a member of the class that contains the public instance member.

According to the running example, Figure 6.8(c) shows an alternative Singleton module. The Singleton module owns two annotation input connectors with PrivateConstructor and PublicClassMember annotation type, respectively. Furthermore, the Singleton module owns two artifact input connectors with Class artifact type and TypeReference artifact type. There-

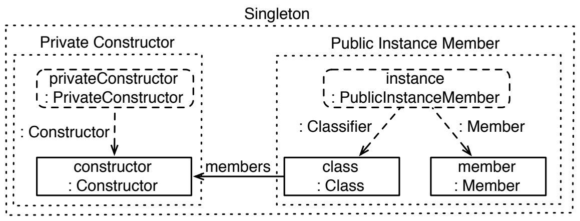
[1]The Singleton design pattern enables callers to receive always the same instance of a class [35, p. 127].

[2]The PrivateConstructor pattern matches constructors with private visibility.

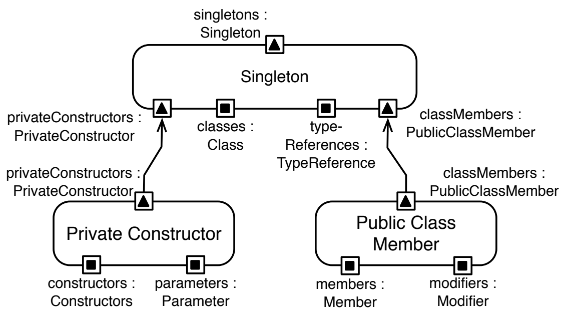
[3]The PublicInstanceMember pattern matches static public fields and methods that store or return instances of the containing class.


singletons : Singleton
Singleton
privateConstructors : PrivateConstructor
publicInstances : PublicInstanceMember
privateConstructors : PrivateConstructor
publicInstances : PublicInstanceMember
Private Constructor
Public Instance Member
constructors : Constructors
parameters : Parameter
members : Member
classes : Class
modifiers : Modifier
typeReferences : TypeReference

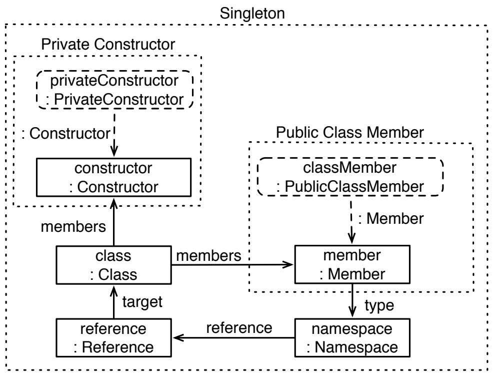
(a) Dependency graph



(b) Additional pattern edges


singletons : Singleton
Singleton
privateConstructors : PrivateConstructor
classes : Class
type-References : TypeReference
classMembers : PublicClassMember
privateConstructors : PrivateConstructor
classMembers : PublicClassMember
Private Constructor
Public Class Member
constructors : Constructors
parameters : Parameter
members : Member
modifiers : Modifier

(c) Dependency graph



(d) Additional pattern nodes and edges

Figure 6.8.: Conjunction with disjoint graph patterns

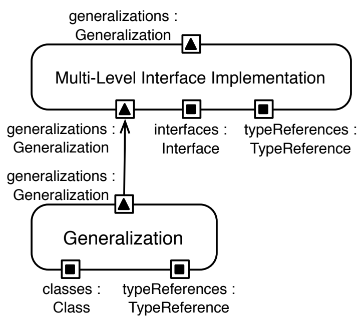
fore, the module describes that it encapsulates conjunctions of the PrivateConstructor and PublicClassMember pattern[4] and, furthermore, requires artifacts with Class and TypeReference artifact type. Figure 6.8(d) depicts that both patterns are disjoint and the overall pattern employs artifacts with Class and TypeReference artifact type (supertype of the Namespace and Reference artifact type) to connect both patterns. The additional pattern nodes connect both patterns in a way that the constructor and the member are members of the same class. Furthermore, the type of the member must be the class that owns the member.

Extended Graph Patterns

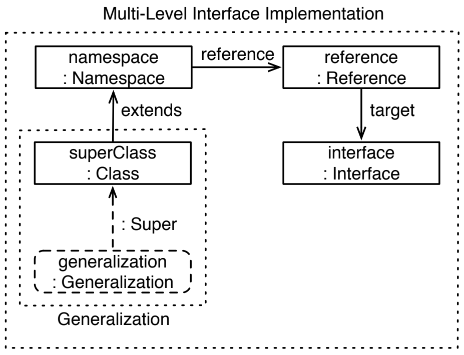
View modules encapsulate extended graph patterns, when they own at least one annotation input connector and at least one artifact input connector.

According to the running example, Figure 6.9(a) depicts an alternative Multi-Level Interface Implementation module that encapsulates the Multi-Level Interface Implementation pattern (cf. Figure 3.5(d)). The Multi-Level Interface Implementation module owns one annotation input connector with Generalization annotation type and two artifact input connectors with Interface and TypeReference artifact type (supertype of the Namespace and Reference artifact type). Therefore, the module describes that it encapsulates a conjunction by extending the Generalization pattern (cf. Figure 3.5(a)) with pattern nodes of Interface and TypeReference artifact type. Figure 6.9(b) depicts the pattern nodes with Interface, Namespace, and Reference artifact type as implicit pattern, which extends the Generalization pattern by a chain of pattern nodes. These nodes describe that the superclass of the generalization implements an interface.

[4]The PublicClassMember pattern matches static public fields or methods.


generalizations : Generalization
Multi-Level Interface Implementation
generalizations : Generalization
interfaces : Interface
typeReferences : TypeReference
generalizations : Generalization
Generalization
classes : Class
typeReferences : TypeReference

(a) Dependency graph



(b) Extended pattern

Figure 6.9.: Conjunction with extended graph pattern

6.4.3. Disjunctions

View modules encapsulate disjunctions of graph conditions, when they own at least one annotation input connector that has multiple incoming module dependencies. Then, the encapsulated pattern exploits the polymorphism of the received annotations to implement disjunctions. Thus, the annotation type of the annotation input connector can have annotation subtypes that are instantiated by other view modules.

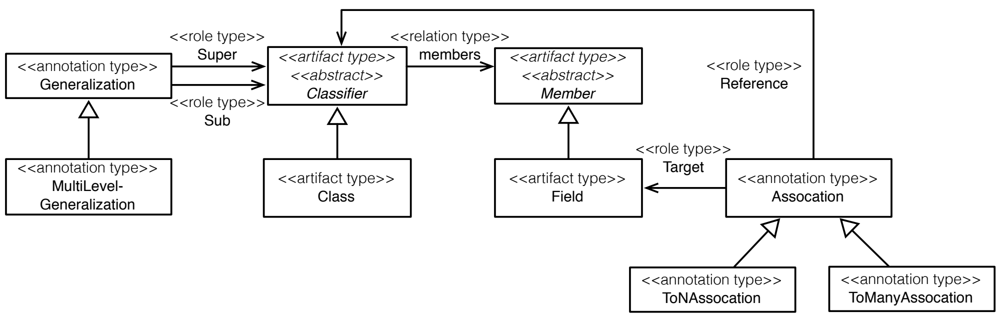
According to the running example, Figure 6.10(b) depicts the Composite module. The Composite module owns two annotation input connectors with Generalization annotation type and Association annotation type, respectively. Both annotation input connectors describe that the Composite module encapsulates two disjunctions.

First, the Composite module encapsulates a disjunction of Generalization patterns, because the annotation input connector with Generalization annotation type has two incoming module dependencies with the Generalization module and Multi-Level Generalization module as sources. According to the running example (cf. Figure 6.10(a)), the Multi-Level Generalization annotation type is a subtype of the Generalization annotation type. Therefore, the Generalization annotation type and the Multi-Level Generalization annotation type have the Sub and Super role type in common. Thus, the Composite module encapsulates a disjunction of Generalization patterns, because the generalization pattern node of the Composite pattern can match annotations that mark matches of the Generalization or Multi-Level Generalization pattern.

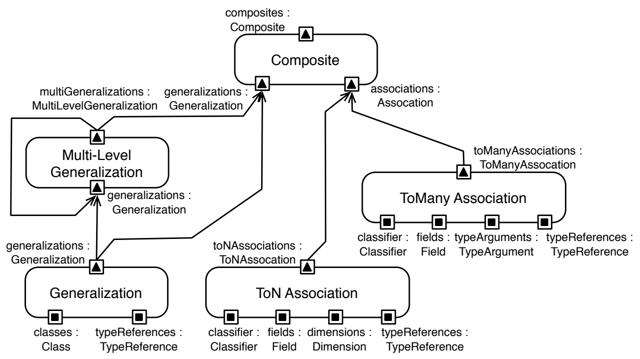
Second, the Composite module implements a disjunction of Association patterns, because the annotation input connector with Association annotation type has two incoming module dependencies with the ToNAssociation module and ToManyAssociation module as sources. According to the running example (cf. Figure 6.10(a)), the ToNAssocation and ToManyAssociation annotation types are subtypes of the abstract Association annotation type. Thus, the Composite module encapsulates a disjunction of Association patterns, because the association pattern node of the Composite pattern can match annotations that mark matches of the ToN Association or ToMany Association patterns.

6.4.4. Negation

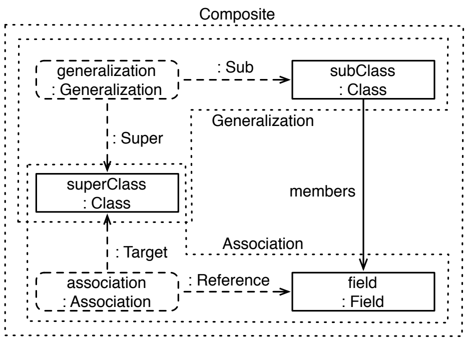
Negations express that certain graph nodes and edges must not exist. According to Definition 8, this thesis distinguishes *simple* and *complex* negation. Figure 6.11 depicts examples for simple and complex negations. The pattern nodes and edges that must not exist are crossed out.



(a) View graph schema



(b) Dependency graph

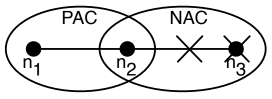


(c) Graph pattern

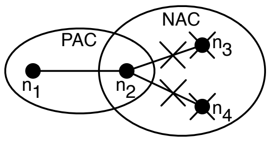
Figure 6.10.: Disjunction

A graph (sub-)pattern is a simple negation, when all pattern nodes of the negated graph (sub-)pattern are directly connected to patterns nodes that are not negated and no additional pattern edges connect the pattern nodes of the negated graph (sub-)pattern. Figure 6.11(a) and Figure 6.11(b) depict simple negations, because the negated pattern nodes n_3 and n_4 are directly connected to the non-negated pattern node n_2 and no graph edges exist between the pattern nodes n_3 and n_4.

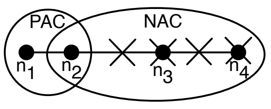
A graph (sub-)pattern is a complex negation, when at least one pattern node of the negated graph (sub-)pattern is *not* directly connected to pattern nodes that are not negated or pattern edges exist that connect the negated pattern nodes. Figure 6.11(c) depicts a complex negation, because the negated pattern node n_4 is *not* directly connected to the non-negated pattern node n_2. Figure 6.11(d) depicts a complex negation, because a pattern edge connects the negated pattern nodes n_4 and n_5.



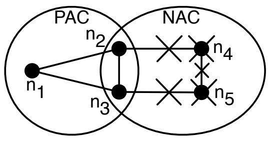
(a) Simple negation



(b) Simple negation



(c) Complex negation



(d) Complex negation

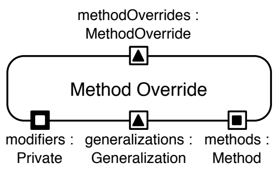
Figure 6.11.: Kinds of negative graph conditions

In this thesis, simple and complex negations are mapped differently to modules. The following sections describe how this thesis maps simple and complex negations to modules.

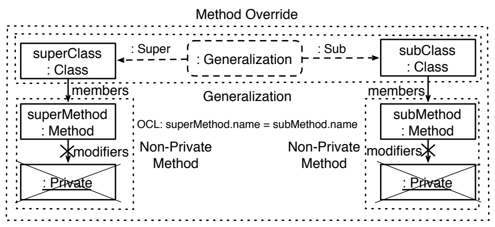
Simple Negation

In this thesis, simple negation is mapped to modules that consist of negated *artifact* input connectors. A negated artifact connector describes that the module checks for the non-existence of artifacts that have an artifact type as specified by the negated connector.

According to the running example, Figure 6.12(a) depicts the Method Override module that encapsulates the Method Override pattern (cf. Figure 3.7). The Method Override module consists of a negative artifact connector with Private artifact type, an artifact connector with Method artifact type, and an annotation input connector with Generalization annotation type. The negative artifact connector with Private artifact type describes that the Method Override module checks for the non-existence of artifacts with Private artifact type. The artifact connector with Method artifact type describes that the Method Override module checks for the existence of artifacts with Method artifact type. The annotation input connector with Generalization annotation type describes that the Method Override module checks for the existence of annotations with Generalization annotation type.



(a) Dependency graph



(b) Graph pattern

Figure 6.12.: Simple Negation

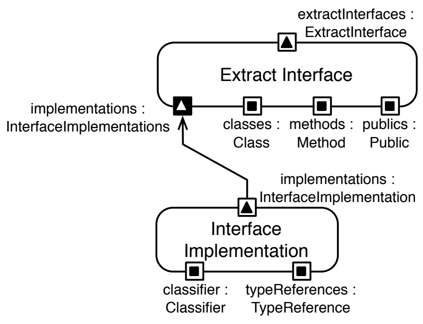
Figure 6.12(b) depicts the Method Override pattern (cf. Figure 3.7). The Method Override pattern consists of a Generalization pattern (cf. Figure 3.5(a)) and two patterns that match methods *without* private modifiers. The crossed out pattern nodes and edges denote the simple negation and describe that the methods must not consist of private modifiers.

Complex Negation

In this thesis, complex negation is mapped to modules that consist of negated *annotation* input connectors. The negated annotation input connector describes that the module checks for the non-existence of annotations that have an annotation type as specified by the negated annotation input connector. Therefore, the mapping of complex negation requires two modules. The first module searches for the matches of a pattern that must *not* exist to satisfy the pattern of the second module.

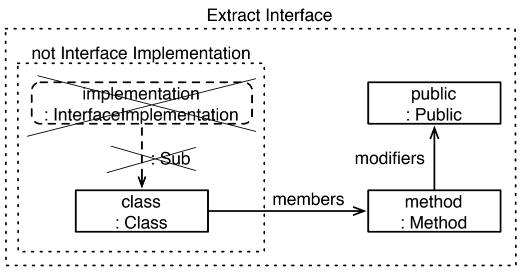
According to the running example, Figure 6.13(a) shows the Extract Interface module that encapsulates the Extract Interface pattern (cf. Figure 3.9). The Extract Interface pattern implements a complex negation of the Interface Implementation pattern (cf. Figure 3.5(b)), because the pattern nodes with Reference and Interface artifact type are not directly connected to non-negated pattern nodes of the Extract Interface pattern.

The Extract Interface module owns a negative annotation input connector with InterfaceImplementation annotation type to describe that the Extract Interface module checks for the non-existence of matches for the Interface Implementation pattern. The InterfaceImplementation module encapsulates the Interface Implementation pattern to create annotations that mark matches of the Interface Implementation pattern. These annotations are forwarded to the Extract Interface module that checks whether Class artifacts do not satisfy the Interface



extractInterfaces : ExtractInterface
Extract Interface
implementations : InterfaceImplementations
classes : Class
methods : Method
publics : Public
implementations : InterfaceImplementation
Interface Implementation
classifier : Classifier
typeReferences : TypeReference

(a) View module dependency graph



(b) Graph pattern

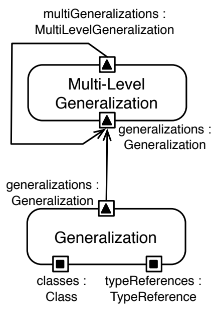
Figure 6.13.: Complex Negation

Implementation pattern. The pattern in Figure 6.13(b) crosses out the pattern node with InterfaceImplementation annotation type to describe that the Class artifact must not participate in a match of the Interface Implementation pattern.

6.4.5. Recursion

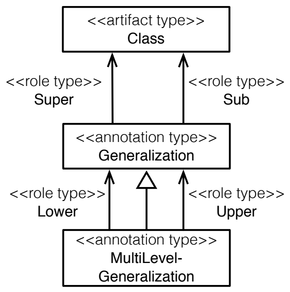
In this thesis, recursive graph conditions (cf. Definition 15) are mapped to at least two modules. One or more modules describe the recursion start. One or more modules describe the recursion step. The view modules that describe the recursion step consist of cyclic module dependencies to take the result of a previous recursion step as input for the next recursion step. Accordingly, the view graph schema consists of annotation types that represent annotations, which mark matches for recursion starts and recursion steps. The annotations that represent recursion starts are polymorphic to annotations that represent recursion steps, i. e., annotations for recursion starts are special kinds of annotations for recursions steps. Then, the view module implementation exploits the polymorphism of the received annotations during the pattern matching, similar to the disjunctions of graph conditions (cf. Section 6.4.3).

Note that also multiple modules can describe the recursion starts and recursion steps. Furthermore, recursion cycles can consist of other recursion cycles.

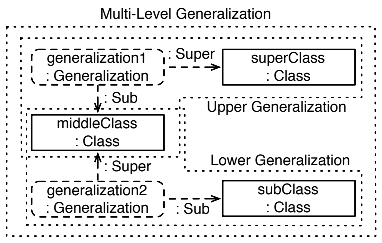


multiGeneralizations : MultiLevelGeneralization
Multi-Level Generalization
generalizations : Generalization
generalizations : Generalization
Generalization
classes : Class
typeReferences : TypeReference

(a) View module dependency graph



(b) View graph schema



(c) Graph pattern

Figure 6.14.: Recursion

According to the running example, Figure 6.14(a) depicts the Generalization module and Multi-Level Generalization module. The Generalization module describes the recursion start and encapsulates the Generalization pattern (cf. Figure 3.5(a)) to create annotations that mark matches of the Generalization pattern. The Multi-Level Generalization module describes the recursion step and encapsulates the Multi-Level Generalization pattern (cf. Figure 3.5(c)) to create annotations that mark matches of the Multi-Level Generalization pattern. The Multi-Level Generalization module consists of a module dependency, which connects its annotation output connector with its annotation input connector, because matches of the Multi-Level Generalization pattern can lead to additional matches of the Multi-Level Generalization pattern as described in Section 3.3.2.

Figure 6.14(b) depicts an excerpt of the view graph schema for Generalization patterns. The Generalization annotation type represents annotations that mark matches for the recursion start. The Multi-Level Generalization annotation type is a subtype of the Generalization annotation type. The Multi-Level Generalization annotation type represents annotations that mark matches for recursion steps. The Multi-Level Generalization annotation type inherits the Super and Sub role types of the Generalization annotation type. Roles with Super and Sub role types mark the outermost super- and subordinate classes of the (multi-level) generalizations. Furthermore, the Multi-Level Generalization annotation type owns the Lower and Upper role type. Roles with Lower and Upper role type mark the annotations that describe the matches of the lower and upper (multi-level) generalizations, which constitute the multi-level generalization. Thus, the Lower and Upper role type mark annotations that were created by a previous recursion step.

Figure 6.14(c) depicts the Multi-Level Generalization pattern that is employed by the Multi-Level Generalization module. The Multi-Level Generalization pattern describes that two matches for Generalization patterns and / or Multi-Level Generalization patterns must exist that have a class in common, which is the outermost superclass in the lower (multi-level) generalization and the outermost subclass in the upper (multi-level) generalization. Since the generalization1 and generalization2 pattern nodes can match annotations with Generalization and Multi-Level Generalization type, also annotations with Multi-Level Generalization annotation type can satisfy the pattern in Figure 6.14(c).

6.5. Discussion

This chapter describes a modeling language, which enables developers to model generalized discrimination networks for view graph maintenance. The network consists of view modules and module dependencies. View modules encapsulate graph patterns. The connectors of modules provide all information that are required by the framework for view graph maintenance such as which kinds of artifacts and annotations are matched by the encapsulated graph patterns and which kinds of annotations are created by the modules to mark found graph pattern matches. The module dependencies describe the reuse of graph pattern matches. View modules and module dependencies constitute directed cyclic graphs that enables developers to model conjunctions, disjunctions, and negations of graph conditions. Furthermore, cycles in the dependency graphs enable developers to model recursive graph conditions.

Table 6.1 summarizes which concepts of the proposed modeling language satisfy the elicited requirements of Section 4.2.2. In summary, the requirement R2a - Encapsulation describes that the modeling language must enable to encapsulate graph queries that are specified by developers. The requirement R2b - Effectiveness describes that markings for matches must keep track of the roles of graph nodes in graph pattern matches. The requirement R2c - Reusability describes that view graphs must be able to build on the content of other view graphs. The requirement R2d - Nesting describes that the modeling language must support

Table 6.1.: Mapping the requirements to the modeling language

		Requirements						
		R2a - Encapsulation	R2b - Effectiveness	R2c - Reusability	R2d - Nesting	R2e - Recursion	R3a - Languages	R3b - Transformations
Modeling Language	Base Graph Schema Definition	◐	◐	○	○	○	○	○
	View Graph Schema Definition	◐	◐	◐	◐	◐	○	○
	View Module	●	○	◐	●	●	●	●
	View Module Dependency	○	○	●	●	●	○	○
	View Module Implementation	○	●	◐	◐	◐	●	●

covered: ●; partially covered: ◐; not covered: ○

the expressiveness of nested graph conditions to enable developers to express disjunctions, conjunctions, and negations of graph conditions. The requirement R2e - Recursion describes that the modeling language must enable developers to model recursive graph conditions. The requirement R3a - Languages describes that the framework must support different kinds of pattern matching approaches to be applicable to a multitude of graph query languages. The requirement R3b - Transformations describes that the framework should support different kinds of graph transformations as native implementation of graph queries.

The modeling language enables developers to model base graph schemata and view graph schemata that describe artifact types and annotation types. The developers employ these artifact and annotation types to describe the interfaces of view modules. The modules encapsulate graph queries behind theses interfaces. Therefore, the base graph and view graph schemata contribute to the satisfaction of the requirement R2a - Encapsulation and the view modules satisfies the requirement R2a - Encapsulation. Note that the base graphs and view graphs must be typed to be able to define such interfaces. Base graphs must be typed weak, e. g., by means of labels, or strong, e. g., by means of types with inheritance. Weak typing can be mapped to strong typing. View graphs must be typed strong, i. e., by means of types with type inheritance, to be able to map disjunction and recursion to view modules, because the mapping of disjunctions and recursion exploits the polymorphism of graph nodes of view graphs. A weak typing of view graphs is not sufficient.

The base graph and view graph schemata describe which kinds of artifacts and annotations can act in certain roles within graph pattern matches. These kinds of roles are defined by the developers depending on the kinds of artifacts and annotations that must be effectively retrievable, when post-processing the matches. Therefore, the modeling of base graph and view graph schemata enables developers to effectively mark graph nodes of matches for easy retrieval. Thus, the base graph and view graph schemata contribute to the satisfaction of the requirement R2b - Effectiveness. The module implementation enables developers to instantiate annotation types and role types, when the implementation finds graph pattern matches. Thus, the module implementation satisfies the requirement R2b - Effectiveness.

The interfaces of modules enable developers to describe which kinds of annotations are processed by modules and which kinds of annotations are created and maintained by modules. Therefore, the view graph schema definition and the modules contribute to the satisfaction of the requirement R2c - Reusability. The view module dependencies enable developers to describe which view modules reuse matches that are maintained by other modules. Therefore, these dependencies satisfy the requirement R2c - Reusability. The module implementation processes the received annotations and, thus, contributes to the requirement R2c - Reusability.

View modules and their dependencies enable to express atomic graph conditions, conjunctions, disjunctions, and negations. Atomic graph conditions map to modules without annotation input connectors. Conjunctions map to modules with annotation input connectors. Disjunctions map to modules with input connectors that receive annotations from two or more predecessor modules. Negations map to modules with negative input connectors. Thus, the modules satisfy requirement R2d - Nesting.

View modules that implement conjunctions, disjunctions, and (complex) negations reuse the matches that are provided by other modules. Module dependencies enable developers to describe the reuse of matches. Thus, the module dependencies satisfy the requirement R2d - Nesting. The mapping of disjunctions requires that the module receives annotations with annotation types that have a common annotation supertype as specified by the input connector of the module, because the module implementation exploits the polymorphism of these annotations to implement the disjunction. Therefore, the view graph schema definition contributes to the satisfaction of the requirement R2d - Nesting. The actual conjunction, disjunction, and negation is implemented within the module itself. Therefore, the module implementation contributes to the satisfaction of the requirement R2d - Nesting.

Cycles in module dependency graphs describe recursive graph conditions. In general, at least one module defines the recursion start and at least one module defines the recursion step. A module dependency between the module that defines the recursion start and the module that defines the recursion step as well as the last and first module in the recursion cycle describes the recursive evaluation of found matches. Therefore, the modules and module dependencies satisfy the requirement R2e - Recursion. Furthermore, the view graph schema must describe annotation types that represent annotations that mark matches of the recursion starts and the recursion steps. Therefore, the view graph schema definition contributes to the satisfaction of the requirement R2e - Recursion. Furthermore, the actual recursion is implemented within the view modules that exploit the polymorphism of annotations that mark the recursion starts and steps. Thus, the module implementations contribute to the satisfaction of the requirement R2e - Recursion.

View modules encapsulate the implementation of graph queries from the framework. Therefore, modules also hide the employed graph pattern matching approach as well as the employed graph transformation languages. Thus, the modules and module implementations satisfies the requirements R3a - Languages and R3b - Transformations.

7. View Graph Transformation

This chapter describes the implementation of view modules by means of graph transformation rules (cf. Definition 10) that transform view graphs to maintain graph pattern matches.

Section 7.1 describes the execution modes of view modules. Section 7.2 describes the implementation of view modules for conjunctions, disjunctions, negations, and recursions.

7.1. View Module Execution Modes

View modules are responsible to maintain the view graphs that they created. For that purpose, modules must be able to create missing annotations, delete obsolete annotations, and update suspicious annotations. View modules provide execution modes for each of these maintenance actions. Figure 7.1 depicts the metamodel for the implementations of modules. The metamodel describes that modules own implementations for creating, deleting, and updating annotations, roles, and scopes in view graphs.

As rule of thumb, the module that initially created an annotation is responsible for its maintenance. For that purpose, view modules keep track of the annotations that they created. Moreover, the annotations keep track of the module by which they were created.

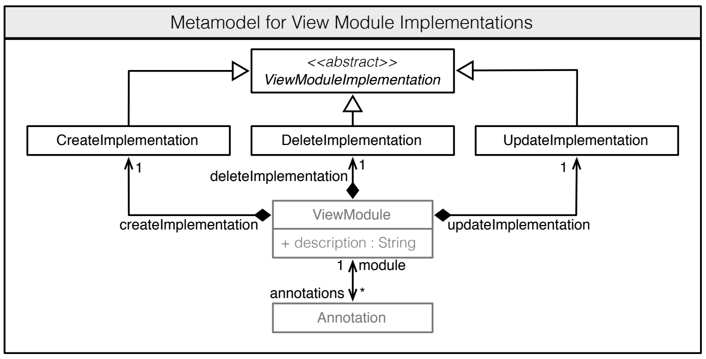
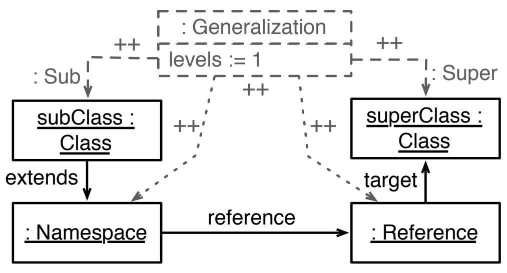


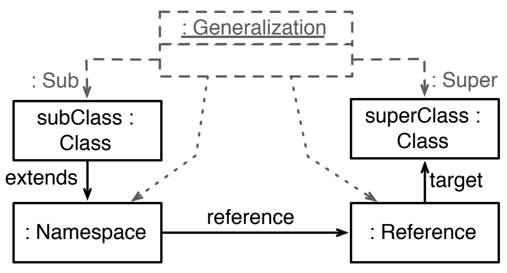
Figure 7.1.: Metamodel for view module implementations

According to the running example, Figure 7.2(a) depicts the CreateImplementation of the Generalization module. This implementation shows a graph transformation rule that employs the shorthand notation for graph transformation rules (cf. Definition 10). The transformation rule employs the Generalization pattern as left-hand side of the transformation rule. The right-hand side of the transformation rule creates an annotation with Generalization type for each match of the Generalization pattern. This annotation references all graph nodes that belong to the match by means of roles as well as scopes and marks the super- and subclass in the match. Furthermore, the transformation rule adds a levels attribute to this annotation to describe that the annotation marks a generalization with one hierarchy level.

This thesis employs the following terminology to distinguish the pattern nodes and edges of graph transformation rules. Solid rectangles are *artifact pattern nodes* and match artifacts of base graphs. Dashed rectangles are *annotation pattern nodes* and match annotations of view graphs. Solid lines are *relation pattern edges* and match relations of base graphs. Dashed lines are *role pattern edges* and match roles of view graphs. Dotted lines are *scope pattern edges* and match scopes of view graphs. Furthermore, pattern nodes can be bound or unbound. This thesis depicts *bound* pattern nodes as pattern nodes with underlined labels. Bound pattern nodes match a predefined set of graph nodes and are the input parameters of graph transformation rules. This thesis depicts *unbound* pattern nodes as pattern nodes that have no underlined label. Unbound pattern nodes can match graph nodes that are not limited to a certain set of predefined graph nodes. Appendix B describes a metamodel for this graph transformation language.



(a) Create implementation



(b) Delete and Update implementation

Figure 7.2.: Graph transformation rules of view module execution modes

The DeleteImplementation and UpdateImplementation can be derived from the CreateImplementation. For the sake of clarity, Figure 7.2(b) shows a general graph transformation rule for both implementations. For the DeleteImplementation and UpdateImplementation, the annotation pattern node, which matches the annotations that have to be maintained, are bound in the graph transformation rule. The remaining pattern nodes are unbound.

The DeleteImplementation deletes an annotation with Generalization annotation type including its roles and scopes, when the annotation consists of at least one role or scope that does not mark a graph node anymore, e. g., when the graph node is deleted.

The UpdateImplementation either preserves an annotation with Generalization annotation type and updates its attribute values, when the annotation still marks graph nodes that satisfy the Generalization pattern, or flags the annotation with Generalization annotation type for deletion, when the marked graph nodes do not satisfy the Generalization pattern anymore.

The following sections describe how and when view modules create, delete, and update annotations to maintain the matches of the encapsulated graph patterns that are stored by view graphs. The next sections demonstrate the execution modes of the view modules by using the example of the Generalization pattern. The Generalization pattern is an example and can be replaced by other graph patterns as well. Note that these patterns can also contain annotation pattern nodes that match annotations of view graphs.

7.1.1. Creation of Annotations

Graph transformation rules for Create implementations of view modules search for graph pattern matches and mark all artifacts and annotations that belong to these matches. All pattern nodes of the encapsulated graph pattern belong to the left-hand side of the graph transformation rule. Therefore, these pattern nodes are bound in the graph transformation rule and match the graph nodes that are received by the type compatible input connectors of the view module. The right-hand side of the graph transformation rule preserves all artifacts and annotations that satisfy the graph pattern and marks these artifacts and annotations as match of the graph pattern. For that purpose, the right-hand side of the graph transformation rule creates an annotation with an annotation type as defined by the output connector of the view module in the view graph. For this annotation, the right-hand side of the graph transformation rule creates roles and scopes, which reference the artifacts and annotations that belong to the match. The right-hand side of the graph transformation rule creates a role with a certain role type for each artifact and annotation of the match, when it has a certain role in the match. The right-hand side of the graph transformation rule creates a scope for each artifact and annotation of the match, when it does *not* have a certain role in the match.

Algorithm 7.1 Create implementation of view modules

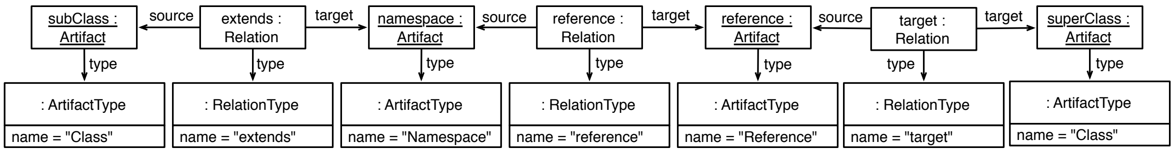
```
Input: Graph nodes of base graphs and view graphs
Output: Created annotations
 1: procedure MODULE_CREATE(nodes)
 2:     annotations := ∅
 3:     for each match of the graph pattern for received graph nodes do          //cf. Fig. 7.3(a)
 4:         if match is not already marked by annotation then                    //cf. Fig. 7.3(b)
 5:             annotation := CREATE_ANNOTATION(match)
 6:             annotations := annotations ∪ {annotation}
 7:     return annotations
```

Algorithm 7.1 shows the Module_Create procedure that describes the Create implementation of view modules with the help of pseudo code. The Module_Create procedure makes use of the patterns and graph transformation rules that are depicted by Figure 7.3. The Module_Create procedure receives graph nodes that define the search space of the view module. The framework passes this search space in terms of artifacts and annotations to the view module.

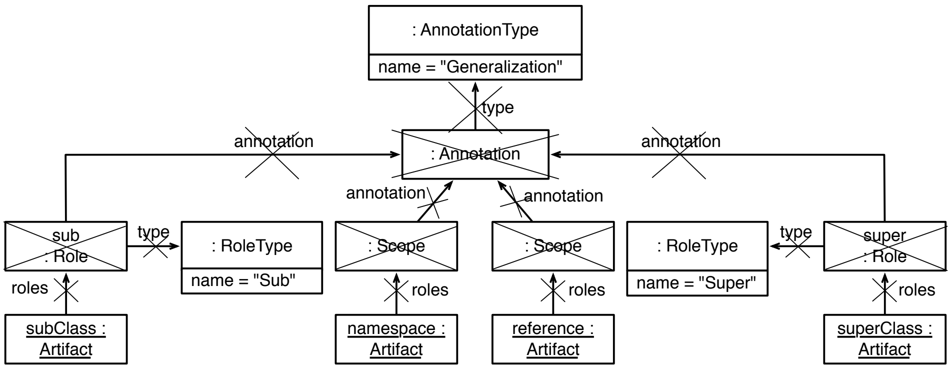
Algorithm 7.2 Creation of annotation in view graphs

```
Input: Graph pattern match that has to be marked
Output: Annotation that marks the graph pattern match
 1: procedure CREATE_ANNOTATION(match)
 2:     annotation := create annotation for match                                //cf. Fig. 7.3(c)
 3:     for each attribute assignment of view graph transformation rule do
 4:         create attribute                                                     //cf. Fig. 7.3(d)
 5:         evaluate expression of attribute assignment                          //cf. Fig. 7.3(d)
 6:     for each role pattern edge of view graph transformation rule do
 7:         if role pattern edge has CREATE modifier then
 8:             create role to mark graph node of match                          //cf. Fig. 7.3(e)
 9:     for each scope pattern edge of view graph transformation rule do
10:         if scope pattern edge has CREATE modifier then
11:             create scope to mark graph node of match                         //cf. Fig. 7.3(f)
12:     return annotation
```

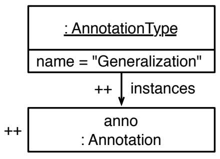
First, the Module_Create procedure initializes an empty set of annotations. Then, the Module_Create procedure searches for matches of the encapsulated graph pattern using the received graph nodes as search space. For each found match, the Module_Create procedure checks whether the graph nodes of the match are *not* already marked by an annotation with an annotation type as defined by the output connector of the view module. If the found match is *not* already marked by an annotation with such an annotation type, the Module_Create procedure creates an annotation as instance of this annotation type. For that purpose, the algorithm calls the Create_Annotation procedure and passes the found match to the procedure. The Create_Annotation procedure returns the created annotation. Afterwards, the Module_Create procedure adds the created annotation to the set of created annotations.



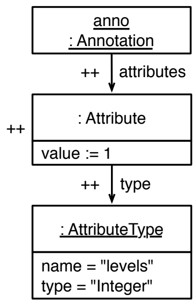
(a) Search graph pattern matches



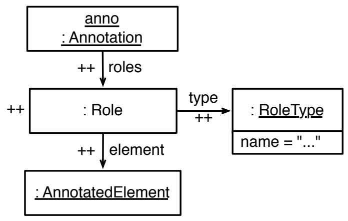
(b) Graph pattern match is not already marked by annotation



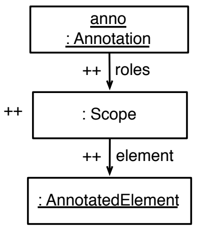
(c) Create annotation



(d) Create attribute



(e) Create role



(f) Create scope

Figure 7.3.: Graph patterns and graph transformations of Create implementation

When the Module_Create procedure investigated the complete search space and does not find additional matches anymore, the Module_Create procedure returns all created annotations.

The Module_Create procedure makes use of the Create_Annotation procedure (cf. Algorithm 7.2). First, the Create_Annotation procedure creates an annotation as instance of the annotation type as defined by the output connector of the view module. For each attribute assignment of the transformation rule, the Create_Annotation procedure creates an annotation attribute as instance of a certain attribute type, adds this annotation attribute to the created annotation, evaluates the expression of the attribute assignment, and sets the evaluation result as attribute value. For each role pattern edge of the transformation rule, the Create_Annotation procedure checks whether the role pattern edge has a Create modifier. If the role pattern edge has a Create modifier, the Create_Annotation procedure creates a role as instance of the role type as defined by the role pattern edge, adds the role to the created annotation, and sets the graph node that matches the target pattern node of the role pattern edge as target of the created role. For each scope pattern edge of the view graph transformation rule, the Create_Annotation procedure checks whether the scope pattern edge has a Create modifier. If the scope pattern edge has a Create modifier, the Create_Annotation procedure creates a scope, adds the scope to the created annotation, and sets the graph node that matches the target pattern node of the scope pattern edge as target of the created scope.

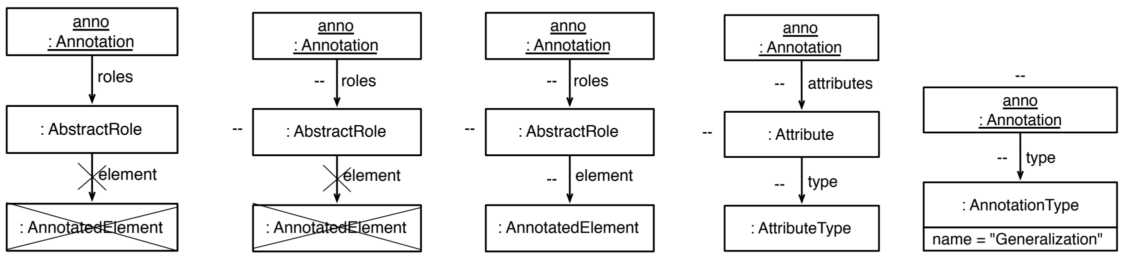
7.1.2. Deletion of Annotations

Graph transformation rules for Delete implementations of view modules delete annotations from view graphs, when these annotations are obsolete, because they do not mark graph pattern matches anymore. The left-hand side of the graph transformation rule consists of a bound annotation pattern node that is the input parameter of the transformation rule. Furthermore, the bound annotation pattern node consists of role pattern edges and scope pattern edges that target at unbound pattern nodes that belong to the left-hand side of the transformation rule as well. These unbound pattern nodes match the artifacts and annotations, which are marked by the annotation that matches the bound annotation pattern node. The right-hand side of the graph transformation rule deletes the annotation that matches the bound annotation pattern node and also deletes all its roles and scopes that mark the graph pattern match, if at least one role or scope does *not* reference an artifact or annotation anymore. The right-hand side of the graph transformation rule preserves all artifacts and annotations that match the unbound pattern nodes.

Algorithm 7.3 Delete implementation of view modules

```
Input: Annotations of view graphs
Output: Graph nodes referenced by deleted annotations
1: procedure MODULE_DELETE(annotations)
2:     markedNodes := ∅
3:     dependentAnnotations := ∅
4:     for each annotation in annotations do
5:         if annotation has dangling role/scope then                    //cf. Fig. 7.4(a)
6:             markedNodes := markedNodes ∪ {graph nodes originally marked by annotation}
7:             dependentAnnotations := dependentAnnotations ∪ annotation.dependents
8:             DELETE_ANNOTATION(annotation)
9:     return markedNodes, dependentAnnotations
```

Algorithm 7.3 describes the Module_Delete procedure. The Module_Delete procedure describes the Delete implementation of view modules with the help of pseudo code. The Module_Delete procedure makes use of the graph patterns and graph transformation rules that are depicted by Figure 7.4. The Module_Delete procedure receives annotations that have to be deleted by the Delete implementation of the view module, if they are obsolete.



(a) Annotation obsolete (b) Remove dangling roles and scopes (c) Remove non-dangling roles and scopes (d) Remove annotation attributes (e) Remove obsolete annotation

Figure 7.4.: Graph patterns and graph transformations of Delete implementation

First, the Module_Delete initializes an empty set of marked graph nodes and dependent annotations. For each annotation in the set of received annotations, the Module_Delete procedure checks whether the annotation owns dangling roles or scopes. If the annotation has dangling roles or scopes, the annotation is obsolete and must be deleted. For each obsolete annotation, the Module_Delete procedure collects the artifacts and annotations that were originally marked by the annotation and adds these artifacts and annotations to the set of originally marked graph nodes. Furthermore, the Module_Delete procedure adds all annotations that dependent on the obsolete annotation to the set of dependent annotations. Then, the Module_Delete procedure calls the Delete_Annotation procedure and passes the obsolete annotation to delete this annotation. When the Module_Delete procedure processed the set of received annotations, the Module_Delete procedure returns a) the set of graph nodes that were originally marked by the deleted annotations and b) the set of annotations that depend on deleted annotations.

Algorithm 7.4 Deletion of annotation from view modules

```
Input: Annotation that has to be deleted
 1: procedure DELETE_ANNOTATION(annotation)
 2:     for each dangling role / scope of annotation do
 3:         remove dangling role / scope                          //cf. Fig. 7.4(b)
 4:     for each non-dangling role / scope of annotation do
 5:         remove non-dangling role / scope                      //cf. Fig. 7.4(c)
 6:     for each attribute in annotation.attributes do
 7:         remove attribute from annotation                      //cf. Fig. 7.4(d)
 8:     remove annotation                                         //cf. Fig. 7.4(e)
```

The Module_Delete procedure makes use of the Delete_Annotation procedure (cf. Algorithm 7.4). The Delete_Annotation procedure removes all dangling and non-dangling roles and scopes from the annotation. Then, the Delete_Annotation procedure removes all attributes from the annotation. Afterwards, the procedure deletes the annotation itself. The procedure preserves the graph nodes that are referenced by roles and scopes.

7.1.3. Update of Annotations

Graph transformation rules for Update implementations of view module revise annotations of view graphs, when graph nodes and edges of base graphs or view graphs changed to ensure that annotations still mark graph pattern matches. The left-hand side of the graph transformation rule consists of a bound annotation pattern node that is the input parameter of the graph transformation rule. Furthermore, the bound annotation pattern node consists of role pattern edges and scope pattern edges that target at unbound pattern nodes that belong to the left-hand side of the graph transformation rule as well. The unbound pattern nodes match the artifacts and annotations that are marked by the annotation that matches the bound annotation pattern node. If the marked graph nodes satisfy the encapsulated graph pattern of the view module, the right-hand side of the graph transformation rule preserves the annotation that marks the match and updates the attribute values of the annotation. If the marked graph nodes do *not* satisfy the encapsulated graph pattern anymore, the right-hand side of the graph transformation rule detaches all graph nodes from the roles and scopes of the annotation to make the annotation obsolete.

Algorithm 7.5 describes the Module_Update procedure. The Module_Update procedure describes the Update implementation of view modules with the help of pseudo code. The Module_Update procedure makes use of the graph patterns and graph transformation rules that are depicted by Figure 7.5. The Module_Update procedure receives annotations and

Algorithm 7.5 Update implementation of view modules

```
Input: Annotations of view graphs
Output: Annotations that were set obsolete
 1: procedure MODULE_UPDATE(annotations)
 2:   obsoleteAnnotations := $\emptyset$
 3:   preservedAnnotations := $\emptyset$
 4:   dependentAnnotations := $\emptyset$
 5:   for each annotation in annotations do
 6:     if annotation marks graph nodes that satisfy graph pattern then    //preserve annotation (cf. Fig. 7.5(a))
 7:       PRESERVE_ANNOTATION(annotation)
 8:       preservedAnnotations := preservedAnnotations $\cup$ {annotation}
 9:       dependentAnnotations := dependentAnnotations $\cup$ annotation.dependents
10:     else                                                               //set annotation obsolete
11:       OBSOLETE_ANNOTATION(annotation)
12:       obsoleteAnnotations := obsoleteAnnotations $\cup$ {annotation}
13:   return obsoleteAnnotations, preservedAnnotations, dependentAnnotations
```

checks whether these annotations mark graph nodes that still satisfy the encapsulated graph pattern of the view module. Otherwise, the view module flags these annotations for deletion.

First, the Module_Update procedure initializes an empty set of obsolete, preserved and dependent annotations. For each annotation in the set of received annotations, the Module_Update procedure checks whether the artifacts and annotations, which are referenced by the annotation, still satisfy the encapsulated graph pattern of the view module. If the referenced artifacts and annotations still satisfy this graph pattern, the Module_Update procedure preserves the annotation. For that purpose, the Module_Update procedure calls the Preserve_Annotation procedure and passes the annotation that must be preserved.

Algorithm 7.6 Preservation of annotation in view graph

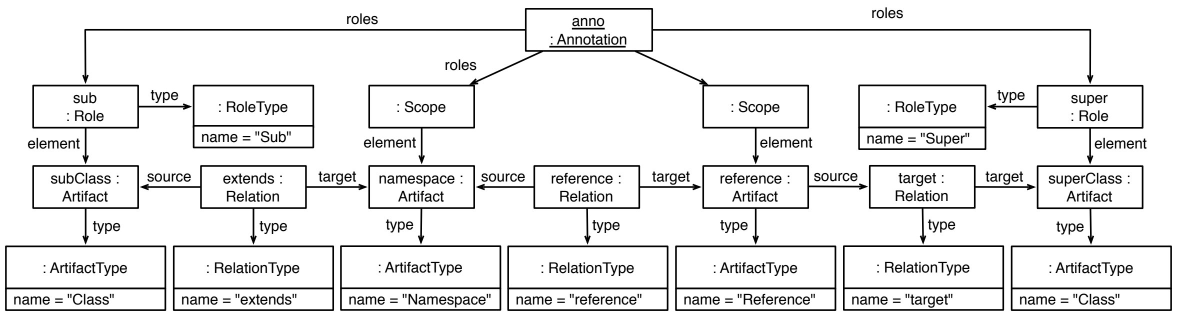
```
Input: Annotation that has to be preserved
1: procedure PRESERVE_ANNOTATION(annotation)
2:   for each attribute in annotation.attributes do
3:     re-evaluate expression of attribute assignments for attribute    //cf. Fig. 7.5(b)
```

The Preserve_Annotation procedure (cf. Algorithm 7.6) updates the attribute value of each annotation attribute by re-evaluating the expression of the corresponding attribute assignment. Then, the Module_Update procedure adds the preserved annotation to the preserved annotations and adds the annotations that dependent on the preserved annotation to the set of dependent annotations. If the referenced artifacts and annotations do *not* satisfy the graph pattern anymore, the Module_Update procedure sets the annotation obsolete. For that purpose, the Module_Update procedure calls the Obsolete_Annotation procedure and passes the annotation that must be set obsolete.

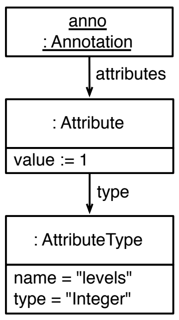
Algorithm 7.7 Set annotations obsolete

```
Input: Annotation that has to be set obsolete
1: procedure OBSOLETE_ANNOTATION(annotation)
2:   for each role / scope in annotation.roles do
3:     set roles and scopes dangling    //cf. Fig. 7.5(c)
```

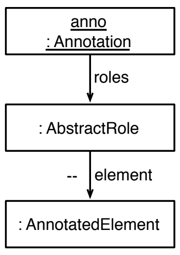
The Obsolete_Annotation procedure (cf. Algorithm 7.7) detaches all graph nodes that are referenced by roles and scopes of the annotation to make the annotation obsolete. Then, the Module_Update procedure adds the obsolete annotation to the set of obsolete annotations. Finally, the Module_Update procedure returns the set of obsolete, preserved, and dependent annotations.



(a) Annotation marks graph pattern match



(b) Preserve annotation and update attributes



(c) Set annotation obsolete

Figure 7.5.: Graph patterns and graph transformations of Update implementation

7.2. Expressiveness of View Modules

This section describes the expressiveness of view modules in conjunction with their encapsulated graph transformation rules. Section 7.2.1 describes graph transformation rules for atomic graph conditions. Section 7.2.2 describes graph transformation rules that implement conjunctions. Section 7.2.3 describes graph transformation rules that implement disjunctions. Section 7.2.4 describes graph transformation rules for simple and complex negations. Section 7.2.5 describes graph transformation rules that implement recursion.

The following section reuse the examples of Section 6.4 and, additionally, describe the employed graph transformation rules. Moreover, Appendix C provides a schematic description of mapping graph conditions to graph transformation rules.

7.2.1. Atomic Graph Condition

Atomic graph conditions are graph conditions that are *not* composed of other graph conditions. According to the running example, Figure 7.6 depicts the Generalization view module as described in Section 6.4.1. The encapsulated graph transformation rule employs an atomic graph condition, because it does not dependent on annotations that are created by other view modules. The Generalization module employs the Generalization pattern as left-hand side of the graph transformation rule. The right-hand side of the graph transformation rule creates a Generalization annotation that marks the super- and subclass of the generalization by creating a Sub role and Super role in the view graph that target at the super- and subclass.

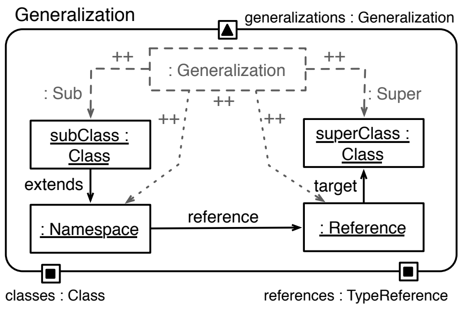


Figure 7.6.: Generalization view module with atomic graph condition

7.2.2. Conjunction

View modules implement conjunctions, when either a) graph patterns overlap in at least one pattern node, b) the graph patterns are disjoint and additional pattern nodes and edges connect these graph patterns, or c) a graph pattern is extended by additional pattern nodes and edges. The following sections describe each variant.

Overlapping Graph Patterns

Graph transformation rules implement conjunctions, when two or more graph patterns overlap in at least one pattern node. According to the running example, Figure 7.7 depicts the Multi-Level Interface Implementation module as described in Section 6.4.2. The module provides annotations for matches of the Multi-Level Interface Implementation pattern (cf. Figure 3.5(d)). The Multi-Level Interface Implementation view module implements a conjunction of the Generalization pattern (cf. Figure 3.5(a)) and Interface Implementation pattern (cf. Figure 3.5(b)). For that purpose, the view module receives annotations that mark matches of the Generalization and Interface Implementation pattern. When the superclass of the generalization implements an interface, the generalization and the interface implementation constitute a multi-level interface implementation. As result, the graph transformation rule creates an annotation with MultiLevelInterfaceImplementation annotation type. This annotation marks the annotations that represent the matches of the Generalization and Interface Implementation pattern. Furthermore, the annotation marks the subclass of the generalization as subordinate and the interface of the interface implementation as superordinate in the hierarchy to describe that the subclass implements the interface.

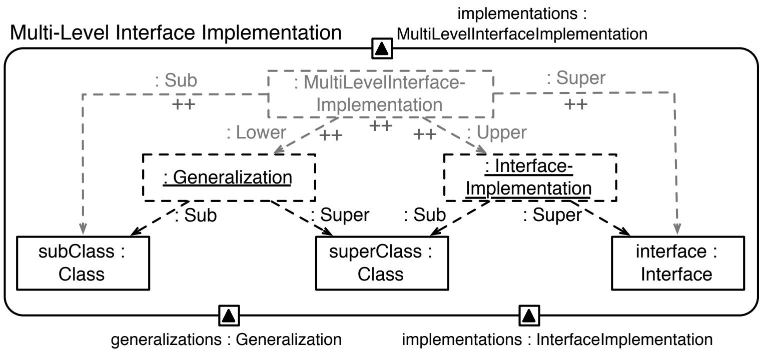

implementations : MultiLevelInterfaceImplementation
Multi-Level Interface Implementation
: Sub
++
: MultiLevelInterface-Implementation
: Super
++
: Lower
++
++
++
: Upper
: Generalization
: Interface-Implementation
: Sub
: Super
: Sub
: Super
subClass : Class
superClass : Class
interface : Interface
generalizations : Generalization
implementations : InterfaceImplementation

Figure 7.7.: Multi-Level Interface Implementation view module with conjunction

Disjoint Graph Patterns

Graph transformation rules implement conjunctions, when two or more disjoint graph patterns are connected by additional pattern nodes and edges. According to the running example, Figure 7.8 depicts the Singleton module as described in Section 6.4.2. The encapsulated transformation rule describes that a class implements a Singleton design pattern, when the class contains a private constructor as well as a static and public member (e. g., a field or method) that stores an instance of the class. The pattern node with Class artifact type connects the PrivateConstructor and PublicInstanceMember pattern, because the public constructor and the public instance member must be members of the same class. When the encapsulated graph transformation rule finds a match, the transformation rule creates an annotation with Singleton annotation type. The created annotation marks the annotations that mark the matches of the PrivateConstructor pattern and PublicInstanceMember pattern. Furthermore, the created annotation marks the class that owns the private constructor and public instance member as class that implements the Singleton design pattern.

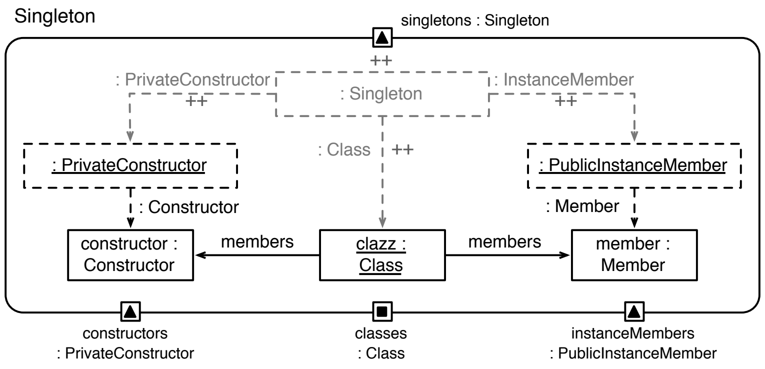


Figure 7.8.: Singleton view module with conjunction

Extended Graph Pattern

Graph transformation rules implement conjunctions, when a graph pattern is extended by additional pattern nodes and edges. According to the running example, Figure 7.9 depicts an alternative Multi-Level Interface Implementation module that extends the Generalization pattern (cf. Figure 3.5(a)) with the Interface Implementation pattern (cf. Figure 3.5(b)) as described in Section 6.4.2 without reusing annotations that mark matches of the Interface Implementation pattern. The encapsulated graph transformation rule describes that the subclass of a match of the Generalization pattern implements an interface, when the superclass of the generalization points via an artifact with Namespace artifact type and an artifact with Reference artifact type to an artifact with Interface artifact type. When the left-hand side of the graph transformation rule finds a match, the right-hand side of the graph transformation rule creates an annotation with Multi-Level Interface Implementation annotation type that marks the subclass of the generalization as subordinate in the hierarchy, the interface as superordinate in the hierarchy, the annotation that marks the reused Generalization annotation, and the artifacts with Namespace and Reference artifact type.

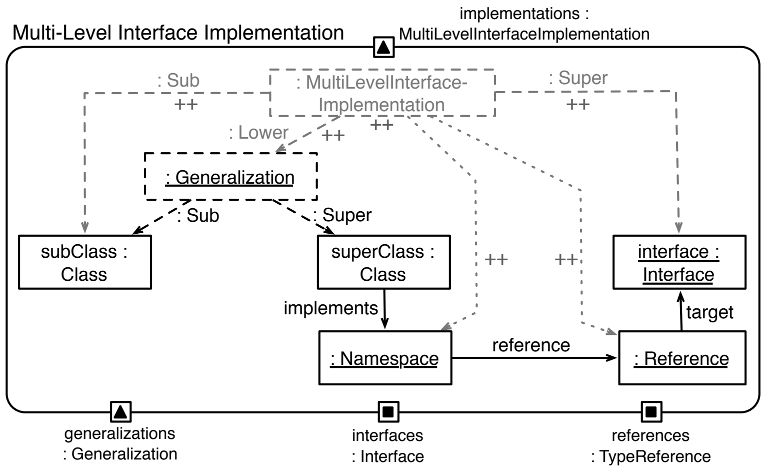


Figure 7.9.: Alternative Multi-Level Interface Implementation view module with conjunction

7.2.3. Disjunction

Graph transformation rules implement disjunctions, when they receive annotations that can match annotation pattern nodes, which employ the annotation super type of the received annotations. According to the running example, Figure 7.10 shows the Composite module, which implements two disjunctions as described in Section 6.4.3. The first disjunction deals with the disjunction of the Hierarchy patterns (cf. Figure 3.5). The second disjunction deals with the disjunction of the Association patterns (cf. Figure 3.6). For that purpose, the Composite module consists of two annotation input connectors that receive annotations with Hierarchy and Association annotation type. That means, these annotation input connectors can receive annotations with annotation subtypes of the Hierarchy and Association annotation type. For example, the annotation input connector with Hierarchy annotation type can receive annotations with Generalization and Multi-Level Generalization annotation type. Then, the annotation pattern node with Hierarchy annotation type can match annotations with Generalization and Multi-Level Generalization type. For example, the annotation input connector with Association annotation type can receive annotations with ToNAssociation and ToManyAssociation annotation type. Then, the annotation pattern node with Association annotation type can match annotations with ToNAssociation and ToManyAssociation type. Thus, the encapsulated graph transformation rule creates annotations with Composite annotation type, when the hierarchy between the subclass and superclass is a generalization or multi-level generalization and the field of the subclass is an association with bounded or unbounded length. Then, the annotation with Composite annotation type references the annotations that lead to the graph pattern match accordingly. Furthermore, this annotation marks the super- and subclass of the generalization as component and composite in the graph pattern match, respectively.

7.2.4. Negation

Negations express that certain graph nodes must not exist, when searching for graph pattern matches. The following sections distinguish simple and complex negations.

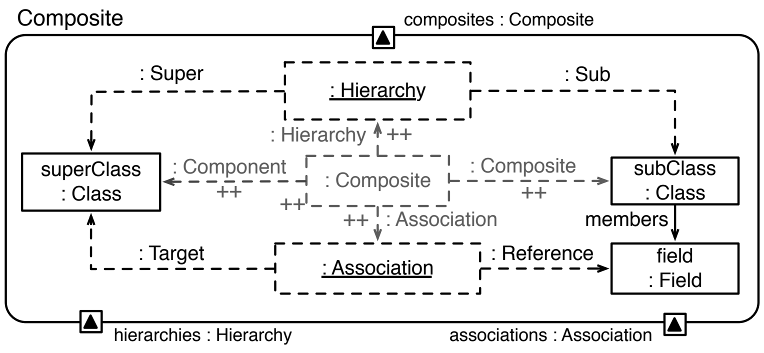


Figure 7.10.: Composite view module with disjunctions

Simple Negation

In this thesis, simple negation (cf. Definition 8) is mapped to negated *artifact* pattern nodes of graph transformation rules. According to the running example, Figure 7.11 shows the MethodOverride module as described in Section 6.4.4. The MethodOverride module implements the Method Override pattern (cf. Figure 3.7). The MethodOverride view module implements two simple negations as denoted by the two crossed out artifact pattern nodes with Private artifact type. These methods must be located in a super- and subclass and must have the same names. The right-hand side of the transformation rule creates annotations with MethodOverride type that mark both methods and the reused generalization annotations.

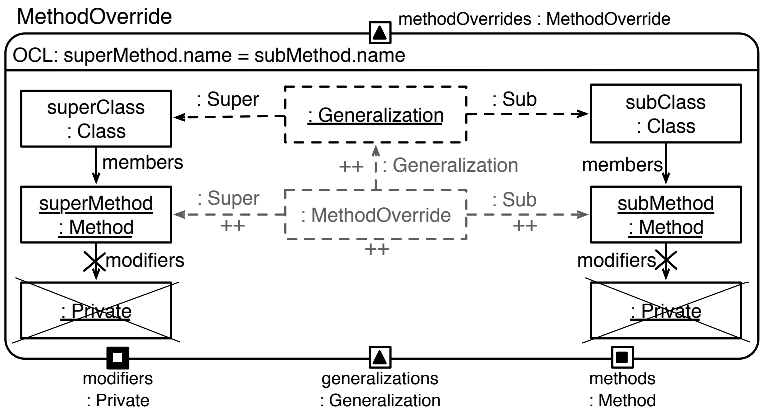


Figure 7.11.: MethodOverride view module with simple negation

Complex Negation

In this thesis, complex negation (cf. Definition 8) is mapped to negated *annotation* pattern nodes in graph transformation rules. Thus, the mapping of complex negation requires two view modules. According to the running example, Figure 7.12 shows the Extract Interface and Interface Implementation module as described in Section 6.4.4, which together enable to detect classes for which an interface should be extracted. The Extract Interface module implements the Extract Interface pattern (cf. Figure 3.9). The Interface Implementation module implements the Interface Implementation pattern (cf. Figure 3.5(b)). The Interface Implementation module creates annotations for classes that implement an interface by marking the interface and the class that implements the interface. The Extract Interface view module receives annotations that mark matches of the Interface Implementation pattern. Furthermore, the graph transformation rule of the Extract Interface module describes that a class must consist of a public method and must *not* implement an interface. Therefore, the graph transformation rule of the Extract Interface module crosses out the annotation pattern node with InterfaceImplementation annotation type to ensure the non-existence of an interface implementation. For each found match, the graph transformation rule creates an annotation with ExtractInterface annotation type that marks the class and method for which an interface should be extracted as well as the public modifier of the method.

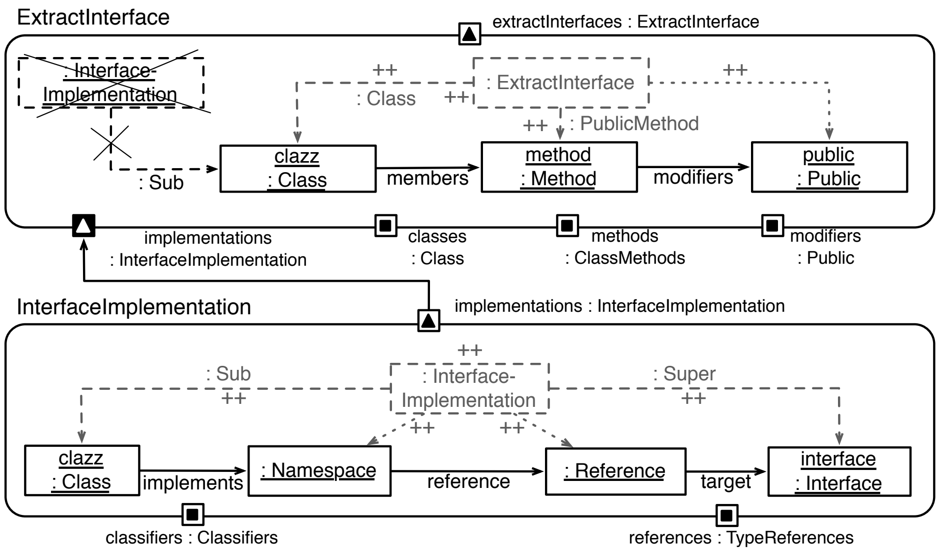


Figure 7.12.: Extract Interface view module with complex negation

7.2.5. Recursion

This thesis refers to cyclic dependencies between view modules as recursion. Recursion is mapped to two or more graph transformation rules. One or more transformation rules describe the recursion start. One or more transformation rules describe the recursion step. The recursion step exploits the polymorphism of annotations, which are created during the previous recursion starts and steps, to implement recursive graph conditions (cf. Definition 15).

Figure 7.13 shows the Generalization and Multi-Level Generalization view modules as described in Section 6.4.5. The Generalization view module at the bottom implements the recursion start. The Generalization view module creates annotations for matches of the Generalization pattern (cf. Figure 3.5(a)). The Multi-Level Generalization view module on top implements the recursion step. The Multi-Level Generalization view module creates annotations for matches of the Multi-Level Generalization pattern (cf. Figure 3.5(c)). According to the running example, the MultiLevelGeneralization annotation type is a subtype of the Generalization annotation type. Therefore, annotations with Generalization as well as MultiLevelGeneralization type can match annotation pattern nodes with Generalization type. For each found match, the transformation rule of the Multi-Level Generalization module marks the reused matches of the Generalization and Multi-Level Generalization pattern as well as the outermost super- and subclass as super- and subordinate in the hierarchy, respectively.

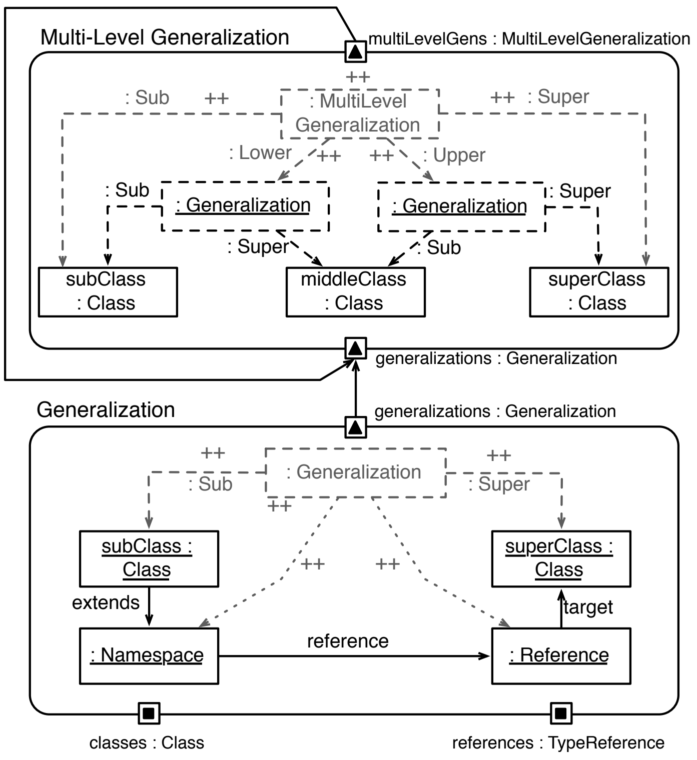


Figure 7.13.: Multi-Level Generalization view module with recursion

7.3. Discussion

This chapter describes the execution modes of view modules for the creation, deletion, and update of annotations. In this thesis, developers employ graph transformation rules to implement these execution modes and transform view graphs to maintain graph pattern matches. The left-hand sides of the graph transformation rules implement graph patterns for which the framework maintains matches. The left-hand sides of the graph transformation rules enable developers to model conjunctions, disjunctions, negations, and recursions in conjunction with view modules and view module dependency graphs as described in Chapter 6. The right-hand sides of the graph transformation rules implement side effects that are applied to view graphs. The right-hand sides of the graph transformation rules create annotations, roles, and scopes in view graphs to mark graph pattern matches. The right-hand sides of the graph transformation rules delete annotations, roles, and scopes from view graphs, when they do not mark graph pattern matches anymore. The right-hand sides of the graph transformation rules update annotations of view graphs, when graph nodes of the marked matches changed and the attribute values of annotations must be revised accordingly, or set annotations obsolete, because they do not mark graph pattern matches anymore.

Table 7.1 summarizes which concepts of the view graph transformations satisfy the elicited requirements of Section 4.2. The requirement R1a - Nativeness describes that developers want to store graph pattern matches natively as graphs. The requirement R1b - Memory-Efficiency describes that developers want to store graph pattern matches efficiently concerning the memory consumption of view graphs by avoiding copies of graph nodes and edges. The requirement R1c - Match-Properties describes that developers want to store additional data

Table 7.1.: Mapping requirements to view graph transformations

		Requirements								
		R1a - Nativeness	R1b - Memory-Efficiency	R1c - Match-Properties	R2b - Effectiveness	R2c - Reusability	R2d - Nesting	R2e - Recursion	R3a - Languages	R3b - Transformations
Graph Transformation	Left-hand side	●	◐	◐	●	●	●	●	●	●
	Right-hand side	●	●	●	●	●	○	◐	○	●
Expressiveness	Conjunction	○	○	○	○	○	●	○	○	○
	Disjunction	○	○	○	○	○	●	○	○	○
	Negation	○	○	○	○	○	●	○	○	○
	Recursion	○	○	○	○	○	○	●	○	○

covered: ●; partially covered: ◐; not covered: ○

values locally at maintained graph pattern matches to enrich graph pattern matches with additional knowledge. The requirement R2b - Effectiveness describes that developers want to store markings of graph pattern matches effectively by keeping track of the roles of graph nodes in these matches. The requirement R2c - Reusability describes that developers want to reuse the content of other graph views. The requirement R2d - Nesting describes that the approach must support nested graph conditions. The requirement R2e - Recursion describes that the approach must support recursion to enable developers to define recursive graph conditions. The requirement R3a - Languages describes that the framework must support different kinds of graph pattern matching approaches. The requirement R3b - Transformations describes that graph transformation rules should be employed as native means to search for graph pattern matches and mark these matches.

Base graphs and view graphs store graphs natively as typed attributed graphs. These graphs can be processed natively using graph transformation rules. Thus, the left-hand sides and right-hand sides of view graph transformations satisfies the requirement R1a - Nativeness.

The left-hand sides of graph transformation rules search for graph pattern matches that are marked as described by the right-hand sides of the graph transformation rules, afterwards. The right-hand sides of graph transformation rules create graph nodes in view graphs that represent matches and create graph edges with certain edge types that target at graph nodes of base graphs and view graphs that satisfy the graph patterns of the left-hand sides. The edge types assign roles to the graph nodes of these matches. Thus, the right-hand sides of the graph transformation rules satisfy requirement R1b - Memory-Efficiency, because they mark graph pattern matches instead of copying graph nodes and edges to view graphs. The left-hand sides of the graph transformation rules contribute to the satisfaction of the requirement R1b - Memory-Efficiency, because they can retrieve graph nodes with certain roles in matches using the edge types of roles.

The right-hand sides of the graph transformation rules enable developers to define the computation of attribute values of annotations. The left-hand sides of the graph transformation rules enable developers to access these attribute values of annotations and process theses values to compute attribute values of other annotations. Thus, the right-hand sides of the graph transformation rules satisfy requirement R1c - Match-Properties, because they compute attribute values of annotations. The left-hand sides of the graph transformation rules contribute to the satisfaction of the requirement R1c - Match-Properties, because they can retrieve attribute values of annotations.

The right-hand sides of the graph transformation rules enable developers to effectively mark graph pattern matches by means of roles. Then, the left-hand sides of the graph transformation rules are able to effectively access graph pattern matches and graph nodes with certain roles in these matches. Thus, the left-hand sides and right-hand sides of view graph transformations satisfy requirement R2b - Effectiveness.

The annotations that are created by the right-hand sides of the graph transformation rules can satisfy the left-hand sides of other graph transformation rules. Thus, the left-hand and right-hand sides of graph transformation rules satisfy the requirement R2c - Reusability.

The left-hand sides of the graph transformation rules enable to describe graph conditions in terms of graph patterns. The graph transformation rules enable to describe conjunctions, disjunctions, and negations of graph conditions. Thus, the left-hand sides of the graph transformation rules satisfy requirement R2d - Nesting.

The left-hand sides of the graph transformation rules enable developers to describe graph patterns that recursively reuse annotations that were created by other view modules in a recursion cycle. Therefore, the right-hand sides of graph transformation rules contribute to requirement R2e - Recursion and the left-hand side of the graph transformation rules satisfy requirement R2e - Recursion.

This thesis does not prescribe which kind of graph pattern matching is employed. Instead, this thesis employs existing graph pattern matching approaches as operationalization of view modules and does *not* provide an own graph pattern matching approach. Since view modules hide the employed graph pattern matching approach, the left-hand sides of graph transformation rules satisfy requirement R3a - Languages.

The graph transformation rules enable developers to natively define kinds of graph pattern matches that must be maintained by the framework. The left-hand sides of the graph transformation rules enable developers to describe graph patterns. The right-hand sides of the graph transformation rules enable developers to describe how found matches are marked for later retrieval. Thus, the left-hand sides and right-hand sides of the graph transformation rules satisfy requirement R3b - Transformations.

8. View Graph Maintenance Algorithms

This chapter describes the maintenance of view graphs, when base graphs change. This maintenance is required, because changes of base graphs may cause satisfied or dissatisfied graph patterns. Then, annotations that are missing in view graphs must be added to view graphs to mark new matches, annotations that do not mark matches anymore must be removed from view graphs, and annotations that still mark matches must be preserved in view graphs.

Section 8.1 gives an overview of the view graph maintenance. Section 8.2 describes how the framework interprets changes of base graphs to support an efficient maintenance of view graphs. Section 8.3 describes the life cycle of annotations during the view graph maintenance. Section 8.4 describes how the framework determines the states of annotations in their life cycle. Section 8.5 describes the maintenance modes of view modules, which maintain annotations that are in certain states of their life cycle. Section 8.6 describes the maintenance phases of the view graph maintenance that execute modules in certain maintenance modes. Section 8.7 describes how the framework computes the search spaces of modules. Section 8.8 describes a naive maintenance strategy for view graphs. Section 8.9 describes a batch maintenance strategy for view graphs. Section 8.10 describes an incremental maintenance strategy for view graphs. Finally, Section 8.11 summarizes the concepts of the view graph maintenance and discusses why these concepts satisfy the elicited requirements.

8.1. Overview

This thesis refers to the maintenance of view graphs due to changes of base graphs as *view graph maintenance*. This thesis proposes different strategies for the view graph maintenance. This thesis distinguishes naive, batch, and incremental maintenance strategies. The naive maintenance strategy deletes all annotations from the view graphs and searches for all graph pattern matches from scratch. The batch maintenance strategy revises *all* annotations of view graphs and preserves annotations that still mark graph pattern matches. The incremental maintenance strategy only revises *portions* of view graphs, which are impacted by base graph changes, and preserves annotations that still mark graph pattern matches.

This thesis subdivides the maintenance algorithms into three maintenance phases, which search for a) annotations that are missing in view graphs, b) annotations that are obsolete and must be removed from view graphs, and c) annotations that are suspicious and must be revised to decide whether they must be preserved in view graphs or must be removed from view graphs. During these maintenance phases, the framework executes the view modules in dedicated maintenance modes that are able to create missing annotations in view graphs, remove obsolete annotations from view graphs, and revise suspicious annotations of view graphs. Depending on the employed maintenance strategy, the framework employs different procedures to compute the search spaces for these view modules. Furthermore, the framework steers the maintenance phases and propagates maintenance information between these phases.

The following sections describe the maintenance algorithms in a bottom-up manner and compose the single parts to naive, batch, and incremental algorithms at the end.

8.2. Changes of Base Graphs

End-users create, delete, and modify graph nodes and edges of base graphs to store domain knowledge. This section describes how the framework interprets these changes of base graphs. This thesis distinguishes the creation, deletion, and modification of graph nodes and edges as well as graph node and graph edge attributes. The following sections describe the semantic of each case. Table 8.1 summarizes the impact of created, deleted, and modified artifacts, relations, and attributes on other artifacts and relations of base graphs. Note that the end-users are not allowed to modify view graphs manually.

Creation

End-users create artifacts in base graphs. The framework interprets these artifacts as added.

End-users create relations between artifacts of base graphs. The framework interprets these relations as added and the source and target artifacts of these relations as modified.

End-users create attributes of artifacts and relations in base graphs. The framework interprets these artifacts and relations as modified. Furthermore, the framework interprets the relations that are connected to these artifacts as well as the source and target artifacts of these relations as modified.

Deletion

End-users remove artifacts of base graphs. The framework interprets these artifacts as deleted.

End-users remove relations between artifacts of base graphs. The framework interprets these relations as deleted and the source and target artifacts of these relations as modified.

End-users remove attributes of artifacts and relations from base graphs. The framework interprets these artifacts and relations as modified. Furthermore, the framework interprets the relations that are connected to these artifacts as well as the source and target artifacts of these relations as modified.

Modification

End-users modify artifacts of base graphs. The framework interprets artifacts as modified, when either a) attributes are added to or removed from artifacts, b) attribute values of these artifacts are changed, or c) incoming or outgoing relations are added, removed, or modified. Then, the framework interprets the relations that are connected to these artifacts as modified.

End-users modify relations between two artifacts in base graphs. The framework interprets relations as modified, when either a) attributes are added to or removed from the relations, b) attribute values of the relations are changed, or c) the source or target artifacts of the relations are changed. Then, the framework interprets the source and target artifacts of the relations as modified. Note that when the source or target artifacts of relations change, the framework interprets the old and new source and target artifacts as modified.

8.3. Life Cycle of Annotations

The state of annotations may change, when base graphs change. This thesis distinguishes missing, obsolete, and suspicious annotations. The following sections describe each kind.

Missing Annotation

A *missing* annotation is an annotation that does not exist in the view graph, although it must exist in the view graph, because a graph pattern match exists that must be marked by an annotation appropriately to represent the match in the view graph.

Table 8.1.: Impact of base graph changes

			Impact of end-user actions											
			Added				Deleted				Modified			
			Artifact	Relation	Artifact (Source)	Artifact (Target)	Artifact	Relation	Artifact (Source)	Artifact (Target)	Artifact	Relation	Artifact (Source)	Artifact (Target)
End-user action	Creation	Artifact	●	○	○	○	○	○	○	○	○	○	○	○
		Relation	○	●	○	○	○	○	○	○	○	○	●	●
		Attribute (Artifact)	○	○	○	○	○	○	○	○	●	●	○	○
		Attribute (Relation)	○	○	○	○	○	○	○	○	○	●	●	●
	Deletion	Artifact	○	○	○	○	●	○	○	○	○	○	○	○
		Relation	○	○	○	○	○	●	○	○	○	○	●	●
		Attribute (Artifact)	○	○	○	○	○	○	○	○	●	●	○	○
		Attribute (Relation)	○	○	○	○	○	○	○	○	○	●	●	●
	Modification	Artifact	○	○	○	○	○	○	○	○	●	○	○	○
		Relation	○	○	○	○	○	○	○	○	○	●	●	●
		Attribute (Artifact)	○	○	○	○	○	○	○	○	●	●	○	○
		Attribute (Relation)	○	○	○	○	○	○	○	○	○	●	●	●

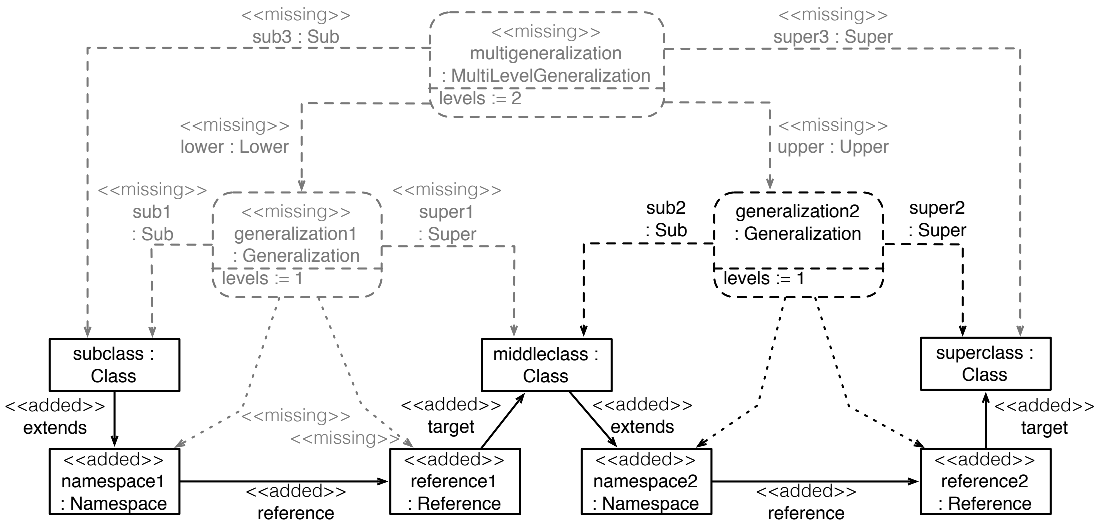
impact: ●; no impact: ○

According to the running example, Figure 8.1 depicts a base graph that consists of the subclass, middleclass, and superclass artifacts. Furthermore, the view graph consists of a generalization2 annotation, which marks the middleclass artifact as subclass and the superclass artifact as superclass of a generalization.

Let's assume an end-user adds the namespace1 and reference1 artifacts as well as the extends, reference, and target relations between the subclass and middleclass artifact to the base graph as depicted by the artifacts and relations with «added» stereotype. Then, the generalization1 annotation is missing as depicted by the annotation, roles and scopes with «missing» stereotype in gray color, because the base graph consists of a match of the Generalization pattern (cf. Figure 3.5(a)) that is currently not represented in the view graph by means of an annotation. Consequently, the multigeneralization annotation is also missing, because the base and view graph consist of a match of the Multi-Level Generalization pattern (cf. Section 3.5(c)) that is currently not represented in the view graph.

Obsolete Annotation

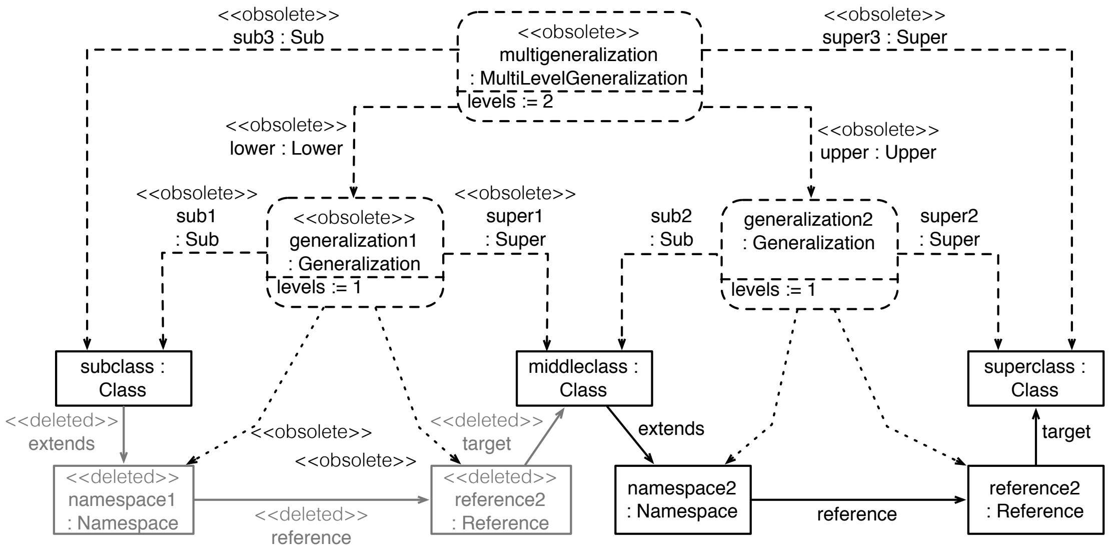
An *obsolete* annotation is an annotation that consists of either a) a dangling role / scope that does not reference a graph node of base graphs or view graphs anymore, or b) a role / scope that references another obsolete annotation.



<<missing>> sub3 : Sub
<<missing>> multigeneralization : MultiLevelGeneralization
levels := 2
<<missing>> super3 : Super
<<missing>> lower : Lower
<<missing>> upper : Upper
<<missing>> sub1 : Sub
<<missing>> generalization1 : Generalization
levels := 1
<<missing>> super1 : Super
sub2 : Sub
generalization2 : Generalization
levels := 1
super2 : Super
subclass : Class
middleclass : Class
superclass : Class
<<added>> extends
<<missing>>
<<missing>>
<<added>> target
<<added>> extends
<<added>> target
<<added>> namespace1 : Namespace
<<added>> reference
<<added>> reference1 : Reference
<<added>> namespace2 : Namespace
<<added>> reference
<<added>> reference2 : Reference

Figure 8.1.: Missing annotations

According to the running example, Figure 8.2 depicts a base graph that consists of the subclass, middleclass, namespace1, and reference1 artifacts, which together satisfy the Generalization pattern (cf. Figure 3.5(a)). Thus, the generalization1 annotation with Generalization annotation type marks these four artifacts. Moreover, Figure 8.2 depicts a generalization2 annotation with Generalization annotation type, which marks a second match of the Generalization pattern. The generalization2 annotation marks the middleclass artifact as subclasss and the superclass artifacts as superclass of a generalization. Furthermore, the generalization1 and generalization2 annotation constitute a match of the Multi-Level Generalization pattern (cf. Figure 3.5(c)) as denoted by the multigeneralization annotation with MultiLevelGeneralization annotation type. The multigeneralization annotation references the subclass and superclass artifacts as well as the generalization1 and generalization2 annotations.



<<obsolete>> sub3 : Sub
<<obsolete>> multigeneralization : MultiLevelGeneralization
levels := 2
<<obsolete>> super3 : Super
<<obsolete>> lower : Lower
<<obsolete>> upper : Upper
<<obsolete>> sub1 : Sub
<<obsolete>> generalization1 : Generalization
levels := 1
<<obsolete>> super1 : Super
sub2 : Sub
generalization2 : Generalization
levels := 1
super2 : Super
subclass : Class
middleclass : Class
superclass : Class
<<deleted>> extends
<<obsolete>>
<<obsolete>>
<<deleted>> target
extends
target
<<deleted>> namespace1 : Namespace
<<deleted>> reference
<<deleted>> reference2 : Reference
namespace2 : Namespace
reference
reference2 : Reference

Figure 8.2.: Obsolete annotations

Let's assume an end-user deletes the namespace1 and reference1 artifacts including the extends, reference, and target relations as depicted by the artifacts and relations with «delete» stereotype in gray color. Then, the generalization1 annotation does not mark a match of the Generalization pattern anymore, because the scopes that previously targeted at the namespace1 and reference2 artifacts are dangling. Therefore, the generalization1 annotation is obsolete as depicted by the annotation, roles, and scopes with «obsolete» stereotype. Consequently, the multigeneralization annotation is obsolete as well, because the lower role targets at the obsolete generalization1 annotation.

Suspicious Annotation

A *suspicious* annotation is an annotation that marks modified artifacts of base graphs or other suspicious annotations of view graphs.

According to the running example, Figure 8.3 depicts a base graph that initially consists of the subclass, middleclass, and superclass artifacts. Figure 8.3 depicts a view graph that consists of the generalization1 and generalization2 annotations. The generalization1 annotation marks the subclass artifact as subclass and the middleclass artifact as superclass of a generalization. The generalization2 annotation marks the middleclass artifact as subclass and the superclass artifact as superclass of a generalization. The view graph also shows a multigeneralization annotation. The multigeneralization annotation marks the subclass artifact as subclass and the superclass artifact as superclass of the multi-level generalization. Furthermore, the multigeneralization annotation marks the lower and upper generalization of the multi-level generalization as denoted by the lower and upper roles.

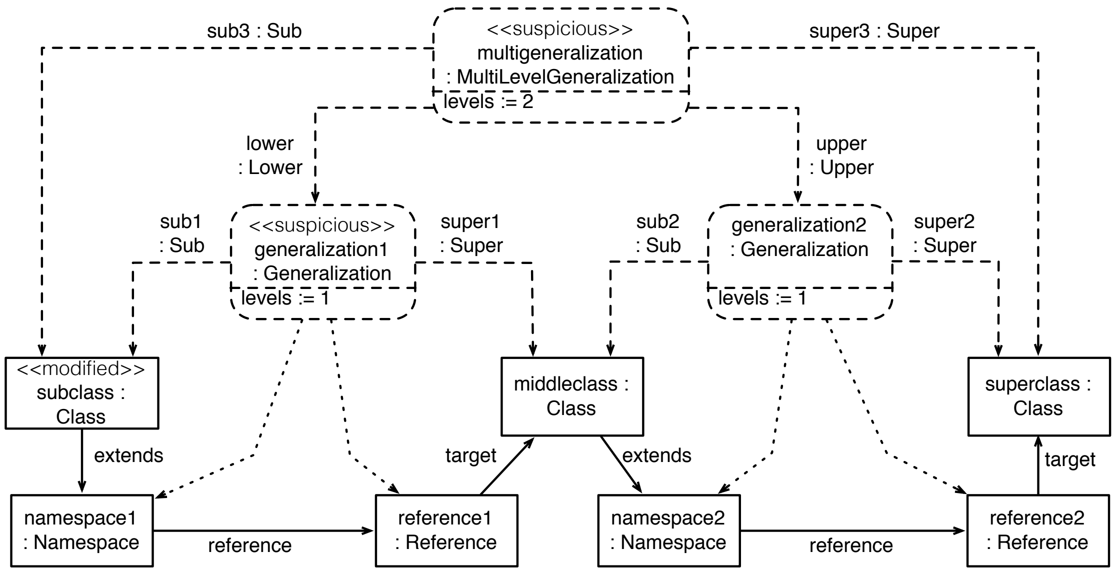


Figure 8.3.: Suspicious annotations

Let's assume an end-user modifies the subclass artifact as denoted by the «modified» stereotype, e. g., by changing its name. Then, the framework interprets the subclass artifact as modified. Consequently, the generalization1 annotation is suspicious as denoted by the «suspicious» stereotype, because it marks the modified subclass artifact. Furthermore, the multigeneralization annotation is suspicious as denoted by the «suspicious» stereotype, because it marks a) the modified subclass artifact and b) the suspicious generalization1 annotation.

8.4. Impact Analysis

This section describes an impact analysis that exploits information about the modification of base graphs to determine the state of annotations in their life cycle. The following sections describe how the framework determines missing, obsolete, and suspicious annotations. Later on, the framework uses these annotations as search space for view modules.

Missing Annotations

Algorithm 8.1 describes the Missing_Annotations procedure. The Missing_Annotations procedure looks up all artifacts that were created and modified in the base graphs since the last view graph maintenance. For that purpose, the Missing_Annotations procedure gets all captured modification events of base graphs as input. First, the algorithm initializes an empty set of artifacts that stores added and modified artifacts, later on. Afterwards, the algorithm iterates over the set of modification events. For each modification event, the algorithm checks whether the event describes the addition or modification of an artifact. If yes, the algorithm adds the artifact to the set of added and modified artifacts. Finally, the algorithm returns all added and modified artifacts as search space for missing annotations.

Note that modified artifacts may satisfy a graph pattern, e. g., due to satisfied attribute constraints. Thus, modified artifacts must belong to the search space.

Algorithm 8.1 Derive the search space for missing annotations

```
Input: Modification events of base graphs
Output: Search space for missing annotations
1: procedure MISSING_ANNOTATIONS(events)
2:     artifacts := ∅
3:     for each event in events do
4:         if event.type = ADDED or event.type = MODIFIED then
5:             artifact := event.artifact
6:             artifacts := artifacts ∪ {artifact}
7:     return artifacts
```

Obsolete Annotations

Algorithm 8.2 describes the Obsolete_Annotations procedure. The Obsolete_Annotations procedure exploits information about the modification of base graphs to look up obsolete annotations. For that purpose, the Obsolete_Annotations procedure gets all captured modification events of base graphs as input. First, the algorithm initializes an empty set of annotations. Afterwards, the algorithm iterates over the set of captured modification events. For each modification event, the algorithm checks whether the event describes the deletion of an artifact. If yes, the algorithm determines the deleted artifact. Then, the algorithm looks up all roles in which the artifact acts. For each role, the algorithm looks up the annotation that owns the role. The algorithm adds this annotation to the set of obsolete annotations. Finally, the algorithm returns all annotations that are obsolete due to the deletion of artifacts.

Suspicious Annotations

Algorithm 8.3 describes the Suspicious_Annotations procedure. The Suspicious_Annotations procedure exploits information about the modification of base graphs to look up suspicious annotations. For that purpose, the Suspicious_Annotations procedure gets all captured modification events of base graphs as input. First, the algorithm initializes an empty set of annotations. Afterwards, the algorithm iterates over the set of captured modification events. For each modification event, the algorithm checks whether the event describes the modification

Algorithm 8.2 Derive obsolete annotations from changes of base graphs

```
Input: Modification events of base graphs
Output: Obsolete annotations
1: procedure OBSOLETE_ANNOTATIONS(events)
2:   annotations := $\emptyset$
3:   for each event in events do
4:     if event.type = DELETED then
5:       artifact := event.artifact
6:       for each role in artifact.roles do
7:         annotation := role.annotation
8:         annotations := annotations $\cup$ {annotation}
9:   return annotations
```

of an artifact. Note that also modified relations result in modified source and target artifacts of relations as described in Section 8.2. If yes, the algorithm determines the modified artifact. Then, the algorithm looks up all roles in which the artifact acts. For each role, the algorithm looks up the annotation that owns the role. The algorithm adds this annotation to the set of suspicious annotations. Finally, the algorithm returns all annotations that are suspicious due to modifications of artifacts.

Algorithm 8.3 Derive suspicious annotations from changes of base graphs

```
Input: Modification events of base graphs
Output: Suspicious annotations
1: procedure SUSPICIOUS_ANNOTATIONS(events)
2:   annotations := $\emptyset$
3:   for each event in events do
4:     if event.type = MODIFIED then
5:       artifact := event.artifact
6:       for each role in artifact.roles do
7:         annotation := role.annotation
8:         annotations := annotations $\cup$ {annotation}
9:   return annotations
```

8.5. Maintenance Modes

According to Section 7.1, view modules consist of three execution modes. These modes process annotations that are in certain states of their life cycle. These modes a) create missing annotations for found graph pattern matches, b) delete obsolete annotations, when the marked graph nodes do not satisfy a graph pattern anymore, and c) update annotations to check whether they still mark matches of satisfied graph patterns. During the view graph maintenance, the framework executes the view modules in these modes. This thesis refers to these execution modes as *maintenance modes* of view modules. Thus, a view module can be executed in CREATE maintenance mode, DELETE maintenance mode, and UPDATE maintenance mode. The following sections summarize each maintenance mode.

Create Maintenance Mode

In CREATE maintenance mode, the framework passes search spaces to view modules. It is the responsibility of the framework to provide artifacts and annotations to view modules that may satisfy the graph patterns that are encapsulated by the modules. These search spaces consist of artifacts and annotations with artifact types and annotation types according to the artifact connectors and annotation input connectors of the modules. According to Section 7.1.1, the CreateImplementations of the modules use the provided artifacts and annotations to search for matches of the encapsulated patterns. For each found match, the CREATE maintenance mode creates one annotation including roles and scopes to mark this match. Furthermore, this maintenance mode collects and returns the created annotations.

Delete Maintenance Mode

In DELETE maintenance mode, the framework passes annotations to view modules that must be deleted by the modules, when they are obsolete. According to Section 7.1.2, the DeleteImplementations of the view modules check whether the provided annotations are obsolete. If yes, the DELETE maintenance mode deletes these annotations including their roles and scopes. If no, the DELETE maintenance mode preserves these annotations. Furthermore, this maintenance mode collects and returns all annotations that depend on the deleted annotations, because then these dependent annotations are obsolete as well.

Update Maintenance Mode

In UPDATE maintenance mode, the framework passes suspicious annotations to view modules that must be revised by the modules. According to Section 7.1.3, the UpdateImplementations of view modules check whether annotations still mark matches of the patterns that are encapsulated by the view module. If the annotations still mark graph nodes that satisfy the patterns, the UPDATE maintenance mode preserves the annotations. If the annotations mark graph nodes that do *not* satisfy the encapsulated patterns anymore, the UPDATE maintenance mode sets the annotations obsolete. Furthermore, this maintenance mode collects and returns all annotations that depend on updated annotations, because then these dependent annotations are suspicious as well.

8.6. Maintenance Phases

The framework employs maintenance phases to structure the overall view graph maintenance. This thesis distinguishes the CREATE, DELETE, and UPDATE maintenance phases. These maintenance phases execute view modules in the corresponding CREATE, DELETE, and UPDATE maintenance modes. The following sections describe each maintenance phase.

Create Maintenance Phase

The CREATE maintenance phase searches for missing annotations in view graphs. For that purpose, the CREATE maintenance phase computes search spaces that must be considered by view modules, when they search for new matches. It depends on the actual maintenance strategy, how the CREATE maintenance phase computes the search space (cf. Section 8.1). Afterwards, the CREATE maintenance phase executes the modules in CREATE maintenance mode and passes the computed search space to the modules. It is the responsibility of the module to search for new matches. It is the responsibility of the module to create annotations, roles, and scopes to mark these matches that are currently not represented by annotations.

Algorithm 8.4 describes the CREATE maintenance phase as Create procedure. Input of the Create procedure is a set of graph nodes that may result in new graph pattern matches. This set contains artifacts of base graphs and annotations of view graphs. Output of the procedure are created annotations that mark found graph pattern matches.

The algorithm traverses the module dependency graph taking into account recursion cycles. When the module dependency graph is acyclic, i. e., the module dependency graph does not consist of recursion cycles, a topological sorting of modules is sufficient to execute the module dependency graph in correct order. When the module dependency graph is cyclic, i. e., the

Algorithm 8.4 Algorithm of CREATE maintenance phase

```
Input: Set of artifacts and annotations that may result in new graph pattern matches
Output: Created annotations
 1: procedure CREATE(nodes)
 2:     newAnnotations := ∅
 3:     createdAnnotations := ∅
 4:     while hasNextModule(newAnnotations) do                              //handles recursion
 5:         module := nextModule(newAnnotations)                            //handles recursion
 6:         candidates := searchScope(nodes,module)                         //cf. Section 8.7
 7:         newAnnotations := module.create(candidates)                     //cf. Algorithm 7.1
 8:         createdAnnotations := createdAnnotations ∪ newAnnotations
 9:     return createdAnnotations
```

module dependency graph consists of recursion cycles, the algorithm executes modules of recursion cycles until the recursion cycle stabilized. In the CREATE maintenance phase, a recursion cycle stabilized, when no module of the recursion cycle created new annotations. The hasNextModule and nextModule procedures implement this behavior. The hasNextModule procedure determines whether additional view modules exist in the view module dependency graph that have to be executed. The nextModule procedure returns the view module that has to be executed next taking into account recursion cycles.

For each module, the algorithm computes a search space in terms of candidate nodes such as artifacts of base graphs and annotations of view graphs using the received nodes. It depends on the maintenance strategy, how the search space is computed (cf. Section 8.1). Then, the algorithm passes the candidate nodes to the module that searches for new graph pattern matches in this search space. Section 7.1.1 describes the steps of the module in CREATE maintenance mode. If the module finds matches, the module returns new annotations that mark these matches. The algorithm adds these new annotations to the set of created annotations. Finally, the algorithm returns all created annotations.

Delete Maintenance Phase

The DELETE maintenance phase deletes obsolete annotations from view graphs. For that purpose, the DELETE maintenance phase executes view modules in DELETE maintenance mode and passes annotations to these modules that be may obsolete. It is the responsibility of the module to check whether annotations are really obsolete and, if yes, to delete these obsolete annotations including their roles and scopes.

Algorithm 8.5 Algorithm of DELETE maintenance phase

```
Input: Annotations that may be obsolete
Output: Artifacts and annotations that were marked by deleted annotations
 1: procedure DELETE(annotations)
 2:     changed := ∅
 3:     for annotation in annotations do
 4:         module := annotation.module
 5:         markedNodes, dependentAnnotations := module.delete(annotation)     //cf. Algorithm 7.3
 6:         changed := changed ∪ {markedNodes}
 7:         changed := changed ∪ DELETE(dependentAnnotations)
 8:     return changed
```

Algorithm 8.5 describes the DELETE maintenance phase as Delete procedure. Input of the Delete procedure is a set of annotations that may be obsolete. Output of the Delete procedure is a set of artifacts and annotations that were marked by deleted annotations.

For each annotation, the algorithm looks up the view module that is responsible to maintain the annotation (cf. Section 7.1), passes the annotation to this module, and executes this module in DELETE maintenance mode. The module checks whether the annotation is obsolete. Section 7.1.2 describes the steps of the module in DELETE maintenance mode. If the annotation is obsolete, the module deletes the annotation including its roles and scopes. The module returns a) all artifacts and annotations that were marked by deleted annotations and b) all annotations that dependent on the deleted annotation. Then, the algorithm adds all artifacts and annotations that were marked by the deleted annotation to the set of changed artifacts and annotations. Afterwards, the algorithm passes the returned dependent annotations to the Delete procedure and executes the Delete procedure recursively, because these annotations are obsolete as well. Thus, the Delete maintenance phase exploits the dependencies between annotations, instead of the module dependencies between view modules, to traverse annotations that are created by different view modules. The call of the Delete procedure returns all artifacts and annotations that were marked by deleted annotations. The algorithm adds these artifacts and annotations to the set of changed artifacts and annotations. Finally, the algorithm returns all artifacts and annotations that were marked by deleted annotations.

Update Maintenance Phase

The UPDATE maintenance phase revises suspicious annotations of view graphs. For that purpose, the UPDATE maintenance phase executes view modules in UPDATE maintenance mode and passes suspicious annotations to these modules. It is the responsibility of the module to check whether a suspicious annotation still marks graph nodes, which satisfy the graph pattern that is encapsulated by the responsible module. It is the responsibility of the module to preserve annotations, which still mark graph nodes that satisfy the pattern. When the module preserves annotations, it is the responsibility of the module to update the values of their attributes by re-evaluating the expressions that compute the attribute values. It is the responsibility of the module to set annotations obsolete, when the graph nodes, which are marked by annotations, do not satisfy the encapsulated graph pattern anymore.

Algorithm 8.6 Algorithm of UPDATE maintenance phase

```
Input: Annotations that may be suspicious
Output: Annotations that are obsolete or preserved
 1: procedure UPDATE(annotations)
 2:   obsoletes := $\emptyset$
 3:   preserved := $\emptyset$
 4:   for annotation in annotations do
 5:     module := annotation.module
 6:     obsoleteAnnotations, preservedAnnotations, dependentAnnotations := module.update(annotation)    //cf. Algorithm 7.5
 7:     obsoletes := obsoletes $\cup$ obsoleteAnnotations
 8:     preserved := preserved $\cup$ preservedAnnotations
 9:     dependentObsoletes, dependentPreserved := UPDATE(dependentAnnotations)
10:     obsoletes := obsoletes $\cup$ dependentObsoletes
11:     preserved := preserved $\cup$ dependentPreserved
12:   return obsoletes, preserved
```

Algorithm 8.6 describes the UPDATE maintenance phase as Update procedure. Input of the Update procedure is a set of annotations that may be suspicious. Output of the Update procedure are two sets of annotations that become obsolete or are preserved during the UPDATE maintenance phase, respectively.

First, the procedure initializes two empty sets of obsolete and preserved annotations. For each annotation, the algorithm looks up the module that is responsible to maintain the annotation. Then, the algorithm passes the annotation to the responsible module and executes the module in UPDATE maintenance mode. The module checks whether the annotation still marks graph nodes that satisfy the graph pattern that is encapsulated by the module. Section 7.1.3 describes the steps of the module in UPDATE maintenance mode. The module returns a) annotations that are set obsolete by the module, b) annotations that are preserved by the module, and c) annotations that dependent on preserved annotations. Note that the UPDATE maintenance mode may changed attribute values of preserved annotations. Afterwards, the algorithm adds the returned obsolete annotations to the set of obsolete annotations and the preserved annotations to the set of preserved annotations. Then, the algorithm passes the returned dependent annotations to the Update procedure and executes the Update procedure recursively, because the modules that are responsible to maintain the dependent annotations may implement conditions over the attribute values of the preserved annotations, which may be dissatisfied now. Thus, the Update maintenance phase exploits the dependencies between annotations, instead of the module dependencies between view module, to traverse annotations that are created by different modules. The call of the Update procedure returns all annotations that are set obsolete and are preserved by the Update procedure. The algorithm adds these dependent obsolete and preserved annotations to the set of obsolete and preserved annotations, respectively. Finally, the algorithm returns the annotations that become obsolete and are preserved during the UPDATE maintenance phase.

8.7. Candidate Sets

The maintenance algorithms execute view modules for certain search spaces. When searching for graph pattern matches, these search spaces describe which artifacts and annotations are considered by view modules in CREATE maintenance mode. Depending on the maintenance strategy, the maintenance algorithms employ different kinds of search space computations. The following sections describe the computations of the search spaces for the batch and incremental view graph maintenance. Especially, the search space computation of the incremental view graph maintenance aims for a reduction of the search space size. Afterwards, this section discusses the differences between both kinds of computations.

Batch Maintenance

Algorithm 8.7 describes the Batch_Search_Scope procedure. When the framework employs a batch maintenance strategy, the Batch_Search_Scope procedure computes the search spaces for modules that are executed during the Create maintenance phase. For that purpose, the Batch_Search_Scope procedure gets the view module as input for which the algorithm computes the search space. First, the algorithm initializes an empty search scope. The algorithm distinguishes two parts. The first part of the algorithm looks up artifacts in base graphs, which have an artifact type that is processed by the view module . The second part of the algorithm retrieves annotations that are maintained by modules that are connected via module dependencies to annotation input connectors of the view module.

The first part of the algorithm iterates over the connectors of the view module. For each connector, the algorithm checks whether the connector is an artifact input connector. If yes, the algorithm looks up its artifact type. Afterwards, the algorithm determines all subtypes of the artifact type in terms of the inheritance clan (cf. Definition 4). For each type in the inheritance clan, the algorithm looks up all its instances and adds these instances to the search scope. Note that the framework maintains a typed-based index for artifact types to look up all instances of an artifact type efficiently (cf. Section 5.1).

The second part of the algorithm iterates over the input connectors of the view module. For

Algorithm 8.7 Candidate set computation of batch maintenance

Input: The module for which the search space is computed
Output: Search space for module in batch maintenance

```
 1: procedure BATCH_SEARCH_SCOPE(module)
 2:     scope := ∅
 3:     for each connector in module.connectors do          //lookup all instances of artifact types relevant for module
 4:         if connector is artifact input connector then
 5:             type := connector.type
 6:             for each subtype in clan(type) do             //cf. Definition 4
 7:                 instances := subtype.instances
 8:                 scope := scope ∪ instances
 9:     for each connector of module.connectors do          //retrieve annotations created by dependency modules
10:         if connector is annotation input connector then
11:             for each dependency in connector.sources do
12:                 source_module := dependency.source
13:                 scope := scope ∪ source_module.annotations  //cf. Section 7.1
14:     return scope
```

each connector, the algorithm checks whether it is an annotation input connector. If yes, the incoming module dependencies are used to look up modules that provide annotations to the view module. The algorithm determines all annotations that are created by these source view modules and adds these annotations to the search scope. Note that each module knows all its created annotations (cf. Section 7.1). Finally, the algorithm returns the search scope.

Incremental Maintenance

Algorithm 8.8 describes the Incremental_Search_Scope procedure. When the framework employs an incremental maintenance strategy, the Incremental_Search_Scope procedure computes the search space for modules that are executed during the Create maintenance phase. For that purpose, the Incremental_Search_Scope procedure gets a set of nodes as input. This set contains artifacts and annotations that underwent modifications in base graphs and view graphs such as the creation and modification of artifacts as well as the modification of annotation attributes. Furthermore, the Incremental_Search_Scope procedure gets the view module as input for which the algorithm computes the search space. Note that this thesis assumes that the encapsulated graph patterns of view modules are connected graphs.

First, the algorithm initializes an empty search scope. The algorithm distinguishes three parts. The first part of the algorithm looks up artifacts and annotations in the set of received nodes that have a type that is relevant to the view module. The second part of the algorithm looks up annotations that were created by predecessor modules during the *current* maintenance. The third part of the algorithm employs a reachability test that collects all artifacts and annotations that are directly or indirectly reachable via relations or roles and have an artifact or annotation type as specified by the input connectors of the view module. This reachability test is required to make the search space complete, because the framework does not make any assumptions on how the view modules process the passed search space. For example, it is up to the view module implementation to decide on at which graph nodes the graph pattern matching starts. Therefore, all reachable graph nodes must be provided at the corresponding input connectors of the view modules.

The first part of the algorithm iterates over the set of nodes. For each node, the algorithm iterates over all connectors of the view module. For each connector, the algorithm checks whether the connector is an artifact input connector or annotation input connector. If yes, the algorithm determines the type of the node. If the inheritance clan (cf. Definition 4) of the connector type contains this type, the node is relevant to the view module. Therefore, the

Algorithm 8.8 Candidate set computation of incremental maintenance

```
Input: Set of artifacts and annotations that may result in new matches and the module
Output: Search scope for module in incremental maintenance
 1: procedure INCREMENTAL_SEARCH_SCOPE(nodes, module)
 2:   scope := $\emptyset$
 3:   for each node in nodes do            //collect artifacts and annotations that are relevant for module
 4:     for each connector in module.connectors do
 5:       if connector is artifact or annotation input connector then
 6:         type := node.type
 7:         if type $\in$ clan(connector.type) then                    //cf. Definition 4
 8:           scope := scope $\cup$ {node}
 9:   for each connector in module.connectors do        //retrieve annotations created by dependency modules
10:     if connector is annotation input connector then
11:       for each dependency in connector.sources do
12:         source_module := dependency.source
13:         annotations := createdInCurrentMaintenancePhase(source_module.annotations)
14:         scope := scope $\cup$ annotations
15:   extendScope := scope
16:   repeat                                                         //reachability test
17:     added := $\emptyset$
18:     for each element in extendScope do
19:       for each node connected to element via relation or role do    //node is either artifact or annotation
20:         for each connector in module.connectors do
21:           if connector is artifact or annotation input connector then   //only for input connectors
22:             if node.type $\in$ clan(connector.type) then                //cf. Definition 4
23:               if scope $\cap$ {node} = $\emptyset$ then
24:                 added := added $\cup$ {node}
25:     scope := scope $\cup$ added
26:     extendScope := added
27:   until added = $\emptyset$
28:   return scope
```

algorithm adds the node to the search scope. Finally, the search scope contains artifacts and annotations that are relevant to the view module and may result in new matches.

The second part of the algorithm iterates over all connectors of the view module. For each connector, the algorithm checks whether the connector is an annotation input connector. If yes, the algorithm looks up the connected source view module by traversing incoming module dependencies in backward direction. Then, the algorithm looks up all annotations that were created by the source view module during the *current* maintenance. The algorithm adds these created annotations to the search scope.

The third part of the algorithm employs a temporary variable called extendScope, which stores all artifacts and annotations that extend the current search scope. First, the algorithm initializes the extendScope variable with the initial search scope. Then, the algorithm enters a loop that is executed as long as the algorithm adds additional artifacts and annotations to the search scope. Within the loop, the algorithm initializes a temporary variable called added with an empty set. For each element in the extendScope set, the algorithm looks up all artifacts or annotations that are directly connected to the element via a relation or role. For each directly connected node, the algorithm iterates over the connectors of the view module to check for each connector whether the connector is an artifact input connector or annotation input connector. If the connector is an artifact or annotation input connector, the algorithm checks whether the node type is contained by the inheritance clan (cf. Section 4) of the input connector type. If yes, the node is relevant to the module and the algorithm checks whether the node is already contained by the search scope. If the search space does not contain the node already, the algorithm stores the node in the set of added nodes. When the algorithm processed all elements of the extendScope set, the algorithm adds the reachable nodes that are stored by the added variable to the search scope. Then, the algorithm replaces the content of the extendScope variable with the content of the added variable. If the set that is stored by the added variable is not empty, the algorithm executes an additional loop cycle to collect relevant reachable artifacts and annotations that are reachable from the artifacts and annotations that were recently added to the search scope. If the set that is stored by the added variable is empty, the algorithm found no additional reachable artifacts and annotations and the algorithm exits the loop. Finally, the algorithm returns the computed search scope.

Discussion

The previous sections describe the computation of search spaces during the Create maintenance phase. This thesis distinguishes a batch and an incremental search space computation.

The batch search space computation looks up all artifacts, which have an artifact type that is relevant to the module for which the search space is computed. The computation adds these artifacts to the search space. Furthermore, the computation adds all annotations, which are maintained by predecessor modules, to the search space. The computation does not exploit information about changes of base graphs.

The incremental search space computation receives artifacts and annotations that underwent modifications and, thus, may satisfy encapsulated graph patterns of view modules. In contrast to the batch search scope computation, the incremental search space computation uses these received artifacts and annotations to remove all artifacts and annotations from the search space that are not reachable from the received artifacts and annotations. These unreachable artifacts and annotations never result in graph pattern matches, because view modules must encapsulate graph patterns that are connected graphs. Furthermore, only added, deleted, and modified artifacts and annotations may result in new graph pattern matches. Artifacts and annotations that are not directly or indirectly connected to these artifacts and annotations cannot result in new graph pattern matches and can be excluded from the search space to decrease the size of the input for the graph pattern matching.

8.8. Naive View Maintenance

This section describes a naive view graph maintenance. Section 8.8.1 describes the algorithm. Section 8.8.2 discusses the handling of suspicious, obsolete, and missing annotations.

8.8.1. Algorithm

Algorithm 8.9 describes the Naive_Maintenance procedure that implements a naive maintenance algorithm. This naive algorithm is a kind of batch maintenance algorithm. This thesis refers to the maintenance algorithm as naive algorithm, because the algorithm deletes all annotations and creates all annotations from scratch. Thus, an annotation that marks the same match before and after the view graph maintenance has different identities before and after the maintenance. Therefore, this naive maintenance algorithm does *not* perform a real view graph maintenance. This thesis refers to a real view graph maintenance, when an annotation keeps its identity, when the match, which is marked by an annotation, does not change.

The Naive_Maintenance procedure gets all artifacts of base graphs and all annotations of view graphs as input. First, the algorithm deletes all annotations. Then, the algorithm passes all artifacts of base graphs to the Create phase that employs the search space computation for the batch maintenance (cf. Section 8.7). Section 8.6 describes the Create phase.

Algorithm 8.9 Naive maintenance algorithm

```
Input: All artifacts and annotations of base graphs and view graphs
Output: Consistent view graphs
1: procedure NAIVE_MAINTENANCE(artifacts, annotations)
2:     clear(annotations)
3:     CREATE(artifacts)                                    //cf. Algorithm 8.4
```

8.8.2. Discussion

First, the Naive_Maintenance procedure deletes all annotations. This deletion implies that the procedure considers all annotations as suspicious and obsolete. Thus, the procedure detects all dissatisfied graph conditions and supports the maintenance of suspicious and obsolete annotations. Second, the procedure searches by means of the Create maintenance phase for all matches from scratch. Thus, the procedure detects all satisfied graph conditions and supports the maintenance of missing annotations. The naive view graph maintenance supports all kinds of PACs and NACs, because the procedure searches for all graph pattern matches from scratch by traversing the view module dependency graph in topological order.

The Naive_Maintenance procedure supports recursion, because the procedure employs the Create maintenance phase, which executes modules in topological order taking into account recursion cycles. The Create phase executes the modules of recursion cycles as long as they create annotations. Afterwards, the Create phase passes the annotations that are created by the recursion cycle to modules that dependent on the modules of the recursion cycle. However, the procedure does not support recursion cycles, when they have to revise suspicious or have to delete obsolete annotations during the execution of the recursion cycle, because the procedure does not employ an Update and Delete maintenance phase that together would enable to detect annotations that must be removed, when the recursion cycle is executed.

The naive algorithm terminates for view module dependency graphs *without* recursion cycles, because the number of artifacts in base graphs has an upper bound during view graph maintenance and each match is marked at most once by an annotation (cf. Section 7.1.1).

The naive algorithm terminates for view module dependency graphs *with* recursion cycles, when additionally, the termination conditions of the recursion steps are correct in a sense that the number of possible matches has an upper bound. The developers of the view modules implement these termination conditions and the framework is *not* able to control these conditions, because the view modules encapsulate these conditions.

8.9. Batch View Maintenance

This section describes a batch algorithm for view graph maintenance that overcomes the limitations of the naive maintenance algorithm. First and foremost, the batch maintenance algorithm performs a real view graph maintenance, because the algorithm preserves all annotations that still mark graph pattern matches. Thus, the algorithm preserves the identity of annotations, when the matches that are marked by these annotations do not change. Section 8.9.1 describes the batch maintenance algorithm. Section 8.9.2 discusses the handling of suspicious, obsolete, and missing annotations.

8.9.1. Algorithm

Algorithm 8.10 describes the Batch_Maintenance procedure that implements a batch maintenance algorithm with real view graph maintenance. The Batch_Maintenance procedure gets all artifacts of base graphs and all annotations of view graphs as input. The Batch_Maintenance procedure applies the maintenance phases in the following order.

Algorithm 8.10 Batch maintenance algorithm

```
Input: All artifacts and annotations of base graphs and view graphs
Output: Consistent view graphs
1: procedure BATCH_MAINTENANCE(artifacts, annotations)
2:   created := ∅
3:   repeat
4:     UPDATE(topological_sort(annotations))          //cf. Algorithm 8.6
5:     DELETE(topological_sort(annotations))          //cf. Algorithm 8.5
6:     created := CREATE(artifacts)                   //cf. Algorithm 8.4
7:     annotations := annotations ∪ created
8:   until created = ∅
```

First, the procedure applies the Update maintenance phase to a) check for all annotations whether they still mark graph nodes that satisfy the encapsulated graph patterns of view modules and b) revise attribute values of these annotations, if these annotations still mark matches. For that purpose, the procedure passes all annotations of view graphs in topological sorted order to the Update maintenance phase. Section 8.6 describes the steps of the Update maintenance phase. Note that the dependencies between the annotations are used for the topological sorting, instead of the dependencies between view modules.

Second, the procedure applies the Delete maintenance phase to a) check for all annotations whether they are obsolete and b) delete these obsolete annotations. For that purpose, the procedure passes all annotations of view graphs in topological sorted order to the Delete maintenance phase. Section 8.6 describes the steps of the Delete maintenance phase. Note that again the dependencies between the annotations are used for the topological sorting.

Third, the procedure applies the Create maintenance phase to a) search for graph pattern matches that are currently not marked by an annotation and b) create annotations to mark these matches. Section 8.6 describes the steps of the Create maintenance phase. The Create maintenance phase employs the batch search space computation (cf. Section 8.7) and, thus, considers all annotations, which are provided by predecessor modules, as search space. Therefore, also annotations with attributes, which change during the Update phase, belong to this search space and the annotations, which are deleted during the Delete phase, do not belong to this search space.

If the Create maintenance phase returns created annotations, the procedure applies an additional iteration of the Update, Delete, and Create maintenance phases. This additional iteration is required, because created annotations may dissatisfy complex NACs that require the non-existence of an annotation (cf. Definition 8). Thus, an additional execution of the Update maintenance phase is required that may result in new obsolete annotations. Consequently, also an additional execution of the Delete maintenance phase is required that may delete annotations. Deleted annotations may satisfy complex NACs that require the non-existence of an annotation (cf. Definition 8). Thus, an additional Create maintenance phase is required to check whether complex NACs are satisfied. The procedure terminates, when the Create maintenance phase creates no new annotations. Otherwise, the procedure executes an additional iteration of the Update, Delete, and Create maintenance phases.

8.9.2. Discussion

The Batch_Maintenance procedure processes all artifacts of base graphs and all annotations of view graphs during view graph maintenance and preserves annotations that still mark matches of the patterns that are encapsulated by view modules.

The UPDATE maintenance phase detects all dissatisfied PACs and NACs due to *modified* artifacts or annotations (cf. Section 8.2), because the UPDATE maintenance phase checks for all annotations whether they still mark graph nodes that satisfy the encapsulated patterns of view modules. Therefore, the UPDATE maintenance phase revises all existing annotations to a) set all annotations obsolete that do not satisfy these graph patterns anymore and b) update all attribute values of preserved annotations.

The DELETE maintenance phase detects all obsolete annotations due to *deleted* artifacts or annotations and annotations that are set obsolete by the previous UPDATE phase, because the DELETE phase checks for all annotations whether they are obsolete. The DELETE phase deletes obsolete annotations. Otherwise, the DELETE phase preserves the annotations.

The CREATE maintenance phase detects all satisfied PACs and NACs, because the CREATE maintenance phase processes all artifacts of base graphs and all annotations of view graphs, i. e., the CREATE phase considers the complete base graph and view graphs as search space. The CREATE maintenance phase executes the modules in topological order taking into account recursion cycles. For each module, the CREATE maintenance phase passes all artifacts that have an artifact type (incl. subtype) as specified by the artifact connectors of the module to the module. Furthermore, the CREATE maintenance phase passes all annotations, which are maintained by predecessor modules, to the module. Thus, when the CREATE maintenance phase executes a certain module, all annotations that are required by the module are available, because the CREATE maintenance phase executes all predecessor modules beforehand.

The batch view graph maintenance supports recursion cycles in the same way as the naive view graph maintenance (cf. Section 8.8). But, the batch view graph maintenance also supports recursion cycles that revise and delete annotations, because the batch view graph maintenance employs an UPDATE and DELETE maintenance phase.

The batch view graph maintenance terminates under the same conditions as the naive view graph maintenance (cf. Section 8.8.2). Additionally, in the UPDATE and DELETE maintenance phases terminate for view module dependency graphs *with* and *without* recursion cycles, when the previous CREATE maintenance phase terminates, because then the number of annotations in the view graphs has an upper bound and the UPDATE and DELETE phase check each annotation only once and create no new annotations. Note that the UPDATE and DELETE phases exploit the dependencies between annotations to look up annotations that must be revised or deleted (cf. Section 8.6).

8.10. Incremental View Maintenance

This section describes an incremental algorithm for view graph maintenance. In contrast to the naive and batch maintenance algorithm, the incremental maintenance algorithm exploits information about the modifications of base graphs to prune the search space of view modules. Section 8.10.1 describes the incremental maintenance algorithm. Section 8.10.2 discusses the handling of suspicious, obsolete, and missing annotations.

8.10.1. Algorithm

Algorithm 8.11 describes the Incremental_Maintenance procedure that implements an incremental maintenance algorithm. The Incremental_Maintenance procedure gets all modification events of base graphs as input. Analog to the Batch_Maintenance procedure, the Incremental_Maintenance procedure applies the maintenance phases in the following order.

Algorithm 8.11 Incremental maintenance algorithm

```
Input: All modification events of base graphs
Output: Consistent view graphs
 1: procedure INCREMENTAL_MAINTENANCE(events)
 2:     suspicious := ∅
 3:     repeat
 4:         suspicious := suspicious ∪ SUSPICIOUS_ANNOTATIONS(events)      //cf. Algorithm 8.3
 5:         obsoletes, preserved := UPDATE(suspicious)                     //cf. Algorithm 8.6
 6:         obsoletes := obsoletes ∪ OBSOLETE_ANNOTATIONS(events)          //cf. Algorithm 8.2
 7:         modified := DELETE(obsoletes)                                  //cf. Algorithm 8.5
 8:         modified := modified ∪ MISSING_ANNOTATIONS(events)             //cf. Algorithm 8.1
 9:         created := CREATE(modified ∪ preserved)                        //cf. Algorithm 8.4
10:         events := ∅
11:         suspicious := REACHABILITY_SUSPICIOUS(created)                 //cf. Algorithm D.1
12:     until suspicious = ∅
```

First, the procedure derives suspicious annotations from captured modification events of base graphs (cf. Section 8.4). The procedure passes these suspicious annotations to the UPDATE maintenance phase. The UPDATE phase a) set these annotations obsolete, if they mark graph nodes that do not satisfy the graph pattern, which is encapsulated by the responsible module, anymore and b) revises the attribute values of these annotations, if the responsible view module preserves these annotations. Section 8.6 describes the steps of the UPDATE phase. The UPDATE phase returns all annotations that become obsolete and are preserved.

Second, the procedure derives obsolete annotations from captured modification events of base graphs (cf. Section 8.4). The procedure adds the derived obsolete annotations to the set of obsolete annotations that is returned by the previous UPDATE phase. Then, the procedure passes the set of obsolete annotations to the DELETE maintenance phase. The DELETE phase a) checks for these obsolete annotations whether they are really obsolete and b), if yes, deletes these obsolete annotations. Section 8.6 describes the steps of the DELETE phase. The DELETE phase returns all artifacts and annotations that were marked by deleted annotations. The algorithm considers these artifacts and annotations as modified.

Third, the procedure derives added and modified artifacts from captured modification events of base graphs (cf. Section 8.2). The procedure adds the derived added and modified artifacts to the set of modified artifacts and annotations that is returned by the previous DELETE phase. Then, the procedure passes the set of added and modified artifacts and annotations as well as the preserved annotations from the UPDATE phase to the CREATE maintenance phase. The procedure passes the preserved annotations to the CREATE phase, because annotation attributes may change in such a way during the UPDATE phase that attribute constraints are satisfied now. The CREATE phase a) searches for new matches that are not already marked by an annotation of the same type and b) creates an annotation for each new match. Section 8.6 describes the steps of the CREATE phase. The CREATE phase employs the search space computation for the incremental maintenance (cf. Section 8.7). The CREATE maintenance phase returns all annotations that are created during the CREATE phase.

After the first iteration of the UPDATE, DELETE, and CREATE phases, all modification events of base graphs are processed. Therefore, the algorithm clears the set of events. Afterwards, the

Incremental_Maintenance procedure uses the annotations that are created during the previous CREATE phase to derive annotations that may be suspicious, because created annotations may dissatisfy complex NACs that require the non-existence of annotations (cf. Section 6.4.4). For that purpose, the procedure employs a reachability test, which collects all annotations that are directly or indirectly reachable from the previously created annotations, when the successor view modules of the modules, which created these annotations, consist of *negative* annotation input connectors (cf. Appendix D.1). The reachability test only traverses artifacts and annotations, which have a type that is relevant to these successor modules. If suspicious annotations exist, the procedure employs an additional iteration of the UPDATE, DELETE, and CREATE phases. Thus, the procedure passes the set of new suspicious annotations to the UPDATE maintenance phase. The UPDATE phase may result in new obsolete annotations. Consequently, also an additional execution of the DELETE phase is required. Therefore, the procedure passes the set of annotations that become obsolete during the previous UPDATE phase to the DELETE phase. The DELETE maintenance phase may delete annotations. Deleted annotations may satisfy complex NACs. Thus, an additional execution of the CREATE maintenance phase is required to create annotations that are currently missing in view graphs. Therefore, the procedure passes the artifacts and annotations, which were marked by the annotations that are deleted by the previous DELETE phase, to the CREATE phase. Afterwards, the Incremental_Maintenance procedure again derives new suspicious annotations based on annotations that are created by the previous CREATE phase. The procedure terminates, when no new suspicious annotations exist. Otherwise, the procedure executes an additional iteration of the UPDATE, DELETE, and CREATE phases.

8.10.2. Discussion

The incremental algorithm prunes the search space of modules using changes of base graphs. The next sections describe the handling of suspicious, obsolete, and missing annotations.

Suspicious Annotations

The UPDATE maintenance phase searches for dissatisfied PACs and NACs. A PAC may be dissatisfied, when graph nodes and edges are deleted or modified (e. g., changed attribute value). These graph nodes and edges are artifacts and relations of base graphs as well as annotations and roles of view graphs. A simple NAC may be dissatisfied, when artifacts and relations are created or modified. A complex NAC may be dissatisfied, when annotations and roles are created or modified. The following paragraphs discuss these scenarios.

According to Section 8.2, when artifacts are connected to or disconnected from other artifacts, the source and target artifacts of relations between are considered as modified. This can be the case, when either a relation is created / deleted between two existing artifacts or an artifact and relation are created / deleted at the same time. Furthermore, artifacts are modified, when their attribute values change. According to Section 8.4, the impact analysis uses these modified artifacts to determine suspicious annotations. The UPDATE phase processes these suspicious annotations to check whether PACs and NACs are still satisfied (incl. check of attribute constraints). If no, the UPDATE phase sets these annotations obsolete. If yes, the UPDATE phase preserves these annotations and updates their attribute values. Thus, the UPDATE phase supports the detection of dissatisfied PACs due to deleted and modified artifacts and relations. Furthermore, the UPDATE phase supports the detection of dissatisfied simple NACs due to created and modified artifacts and relations.

According to Section 6.4, PACs can require the existence of annotations that mark certain matches. These annotations may be set obsolete during the UPDATE phase, e. g., when attribute changes of annotations dissatisfy PACs. When the UPDATE phase sets an annotation obsolete, the subsequent DELETE phase deletes these annotations as well as all annotations that dependent on this annotation, because then the dependent annotations are obsolete as well. Thus, the UPDATE phase supports the detection of dissatisfied PACs due to deleted or modified annotations.

According to Section 6.4.4, complex NACs must be mapped to view module, which search for graph pattern matches that dissatisfy these complex NACs. Thus, created and modified annotations may dissatisfy complex NACs. For example, a created annotation for a match of the Interface Implementation pattern (cf. Figure 3.5(b)) can dissatisfy a match of the Extract Interface pattern (cf. Figure 3.9). The Algorithm 8.11 employs a reachability test to determine annotations that are suspicious due to created annotations. Then, the UPDATE phase checks whether the complex NACs are still satisfied by the graph nodes that are marked by these suspicious annotations. If no, the UPDATE phase sets the annotations obsolete and the subsequent DELETE phase deletes these annotations. If yes, the UPDATE phase preserves the annotations, updates their attribute values, and checks whether constraints over annotation attributes of dependent modules are dissatisfied. Thus, the UPDATE phase supports the detection of dissatisfied complex NACs due to created and modified annotations.

Obsolete Annotations

The DELETE maintenance phase deletes obsolete annotations. Annotations are obsolete, when they consist of dangling roles or scopes. Annotations become obsolete, when end-users delete artifacts that are marked by annotations. Then, this modification dissatisfies a PAC and the annotation must be removed. Furthermore, annotations become obsolete, when the previous UPDATE phase sets them obsolete due to dissatisfied PACs, e. g., when attribute constraints are not satisfied anymore, or dissatisfied NACs, e. g., when annotations are created by a previous CREATE phase that now dissatisfy NACs. Moreover, the DELETE phase deletes annotations that are obsolete due to other deleted annotations. Thus, the DELETE phase in combination with the UPDATE phase supports dissatisfied PACs and NACs.

Missing Annotations

The CREATE maintenance phase searches for satisfied PACs and NACs. A PAC may be satisfied, when an artifact, relation, or annotation is created or modified. A NAC may be satisfied, when an artifact, relation, or annotation is deleted or modified.

According to Section 8.2, the source and target artifacts of a relation are modified, when the relation is created or modified. A created artifact results in created relations. Created and modified artifacts are input to the reachability test of the CREATE phase. Thus, added and modified artifacts belong to the search space of view modules. Furthermore, annotations that are created by predecessor view modules are input to the reachability test and belong to this search space as well. Moreover, Algorithm 8.11 passes preserved annotations with changed attribute values to the CREATE phase and the reachability test. Thus, also these potentially modified annotations are part of this search space. Consequently, the reachability test starts at all kinds of artifacts and annotations that can result in satisfied PACs now. Thus, the CREATE phase supports the detection of satisfied PACs due to created and modified artifacts, relations, and annotations.

According to Section 8.2, the source and target artifacts of a relation are modified, when the relation is deleted or modified. A deleted artifact results in deleted relations. Modified artifacts are input to the CREATE phase and reachability test. Consequently, the search space covers graph nodes that now may satisfy simple NACs. Thus, the CREATE phase supports the detection of satisfied simple NACs due to deleted and modified artifacts and relations.

According to Section 6.4.4, complex NACs must be mapped to view module that search for graph pattern matches that dissatisfy these complex NACs. According to Algorithm 8.11, artifacts and annotations are considered as modified, when the DELETE phase deletes annotations that mark these artifacts and annotations. These modified artifacts and annotations are input to the CREATE phase and reachability test. Consequently, the search space covers graph nodes that now may satisfy complex NACs. Thus, the CREATE phase supports the detection of satisfied complex NACs due to deleted annotations.

Furthermore, the CREATE phase receives preserved annotations with potentially modified attribute values. The preserved annotations are passed to the reachability test. Thus, the CREATE phase supports the detection of satisfied complex NACs due to modified annotations.

Recursion

The UPDATE and DELETE phase of the incremental maintenance algorithm use the roles and scopes between annotations to revise annotations that dependent on other revised and obsolete annotations, respectively. The revision starts at suspicious and obsolete annotations, respectively. As described for suspicious and obsolete annotations, both maintenance phases detect dissatisfied PACs and NACs.

When the CREATE phase terminates before, the number of annotations that depend on suspicious / obsolete annotations has an upper bound. Thus, the number of annotations that must be revised / deleted, when an annotation is suspicious / obsolete has an upper bound, too. Thus, the UPDATE and DELETE maintenance phases terminate.

As for naive and batch view graph maintenance, the CREATE maintenance phase executes modules of recursion cycles until no module of the recursion cycles created an annotation anymore, because then the outputs of the modules have no impact on the output of the other modules in the recursion cycles anymore. During CREATE maintenance phase, the algorithm passes annotations that are created by modules of the recursion cycle to dependent modules *within* the recursion cycle, first. When the recursion cycle terminates, the algorithm passes the annotations that are created by modules of the recursion cycle to dependent modules that do *not* belong to the recursion cycle. Thus, the recursion cycle detects all satisfied PACs and NACs as described for missing annotations.

Analog to the naive and batch view graph maintenance, the incremental view graph maintenance terminates for view dependency graphs with and without recursion cycles. Analog, the recursion cycles terminate in CREATE phase, when the developers ensure that the termination conditions of the recursion steps result in an upper bound of possible matches.

8.11. Discussion

This chapter describes view graph maintenance algorithms, which enable the framework to maintain matches of graph patterns. The view graph maintenance algorithms base on maintenance phases that execute modules in certain maintenance modes to update existing annotations, delete obsolete annotations, and create annotations that are currently missing in view graphs. Depending on the employed view graph maintenance algorithm, the maintenance algorithm passes different kinds of search spaces to modules. The naive and batch maintenance algorithms consider the complete base graphs and view graphs, when maintaining view graphs. The incremental maintenance algorithm considers local portions of base graphs and view graphs. The framework derives these portions from modification events of base graphs and propagates the impact of these changes through the view module dependency graph.

Table 8.2.: Mapping the requirements to the view graph maintenance algorithms

		Requirements							
		R1b - Memory-Efficiency	R1c - Match-Properties	R2c - Reusability	R2d - Nesting	R2e - Recursion	R4a - Monitoring	R4b - Time-Efficiency	R4c - Propagation
Modifications	Base Graph Changes	○	○	○	○	○	●	◐	◐
	Annotation Life Cycle	○	○	◐	○	○	◐	◐	◐
	Impact Analysis	○	○	◐	○	○	◐	◐	◐
Algorithmic Framework	Maintenance Modes	●	●	◐	◐	◐	○	◐	◐
	Maintenance Phases	●	●	●	●	●	○	◐	●
View Graph Maintenance	Naive Maintenance	●	●	●	●	○	○	○	◐
	Batch Maintenance	●	●	●	●	●	○	○	◐
	Incremental Maintenance	●	●	●	●	●	○	●	●

covered: ●; partially covered: ◐; not covered: ○

Section 4.2 describes the requirements for the maintenance of view graphs. Table 8.2 summarizes which concepts of the proposed maintenance algorithms satisfy the elicited requirements. Requirement R1b - Memory-Efficiency describes that view graphs must store matches memory-efficient without copying graph nodes and edges to view graphs. Requirement R1c - Match-Properties describes that matches for patterns must enable to store additional properties of matches to enrich view graphs with additional knowledge. Requirement R2c - Reusability describes that view graphs must be able to build on the content of other view graphs. Requirement R2d - Nesting describes that module dependency graphs must enable developers to express disjunctions, conjunctions, and negations of graph conditions. This expressiveness must be also supported by the view graph maintenance. Requirement R2e - Recursion describes that module dependency graphs must enable developers to describe recursive graph conditions. These recursive definitions must be also supported by the view graph maintenance. Requirement R4a - Monitoring describes that the framework must track changes of base graphs to prune search spaces of view modules. Requirement R4b - Time-Efficiency describes that the view graph maintenance algorithms must employ pruned search spaces to efficiently maintain view graphs. Requirement R4c - Propagation describes that the framework must propagate changes of view graphs to dependent view graphs.

The maintenance modes of view modules and the maintenance phases of the view graph maintenance algorithms process suspicious, obsolete, and missing annotations to revise view graphs. Especially, the CREATE maintenance phase and CREATE maintenance mode create annotations and, therefore, mark matches of patterns in a memory-efficient manner. Thus, the maintenance modes and maintenance phases satisfies requirement R1b - Memory-Efficiency. Furthermore, all view graph maintenance algorithms employ the CREATE maintenance phase and, thus, also satisfy requirement R1b - Memory-Efficiency.

The maintenance modes and maintenance phases enable to maintain attribute values of annotations. The CREATE maintenance mode and CREATE maintenance phase add attribute values to annotations. The UPDATE maintenance mode and UPDATE maintenance phase update attribute values of annotations. Thus, the maintenance modes and maintenance phases satisfy requirement R1c - Match-Properties. Furthermore, all view graph maintenance algorithms employ the CREATE maintenance mode and CREATE maintenance phase and, thus, also satisfy requirement R1c - Match-Properties.

The concept of annotation life cycles describes that annotations, which dependent on obsolete annotations, are also obsolete. Furthermore, annotation that dependent on suspicious annotations are also suspicious. Thus, the annotation life cycles and the impact analysis contribute to the satisfaction of requirement R2c - Reusability.

The maintenance modes of modules return annotations, which enable the maintenance algorithms to trigger the view modules that are responsible for the maintenance of these annotations. Therefore, the maintenance modes support the reuse of annotations between modules and, thus, contribute to the satisfaction of requirement R2c - Reusability.

The maintenance phases collect the annotations that are returned by modules and propagate annotations between modules. Thus, in combination with the maintenance modes of modules, the maintenance phases satisfy requirement R2c - Reusability. Furthermore, all maintenance algorithms employ these maintenance phases and, thus, satisfy requirement R2c - Reusability.

The modeling approach maps atomic graph conditions and simple NACs to single view modules. Therefore, modules can maintain matches for patterns with atomic graph conditions and simple NACs independently from other modules. Thus, the maintenance modes of modules partially satisfy the requirement R2d - Nesting.

The modeling approach splits up disjunctions, conjunctions, and complex NACs to multiple modules. Therefore, multiple modules must be executed to maintain matches for patterns with disjunctions, conjunctions, and complex NACs. Thus, the maintenance phases in combination with the maintenance modes satisfy requirement R2d - Nesting.

All view graph maintenance algorithms employ the CREATE maintenance phase, which executes modules in topological order taking into account recursion cycles. Thus, all view graph maintenance algorithms satisfy requirement R2d - Nesting.

The modeling approach maps recursive graph conditions to modules that describe the recursion start and the recursion step. The recursion start and the recursion step is handled by modules that are executed in certain maintenance modes depending on the maintenance phase. The maintenance phases handle the execution of the recursion cycle, i. e., decide on whether an additional execution of the modules in the recursion cycle is required or not. Thus, the maintenance modes contribute to the satisfaction of requirement R2e - Recursion and the maintenance phases satisfy requirement R2e - Recursion.

Note that the naive maintenance algorithm does not employ a DELETE maintenance phase and, therefore, does not support recursion cycles that must delete annotations, when complex NACs become dissatisfied due to created annotations. Thus, only the batch and incremental maintenance algorithms satisfy the requirement R2e - Recursion.

The monitoring of base graph changes tracks creations, deletions, and modifications of artifacts and relations in base graphs. Thus, the monitoring of base graph changes satisfies requirement R4a - Monitoring.

The impact analysis processes captured base graph changes to determine suspicious, obsolete, and missing annotations. Therefore, the impact analysis and the annotation life cycles contribute to the satisfaction of requirement R4a - Monitoring.

The monitoring of base graph changes, the annotation life cycles, and the impact analysis enables the framework to determine suspicious, obsolete, and missing annotations. The suspicious, obsolete, and missing annotations limit the search space to local portions of base graphs and view graphs. Thus, the monitoring of base graph changes, the annotation life cycles, and the impact analysis contribute to the satisfaction of requirement R4b - Time-Efficiency.

The maintenance modes of view modules process these local portions. Thus, the maintenance modes contribute to the satisfaction of requirement R4b - Time-Efficiency.

The maintenance phases process the search spaces that are provided by the maintenance algorithms. It depends on the concrete maintenance strategy, how the search space is computed. Thus, the maintenance phases contribute to the satisfaction of the requirement R4b - Time-Efficiency.

The naive and batch maintenance algorithm do not satisfy requirement R4b - Time-Efficiency, because both view graph maintenance algorithms do not take base graph changes into account to prune the search space. The incremental view graph maintenance algorithm uses the captured base graph changes to employ an impact analysis that enables the maintenance procedure to pass pruned search spaces to view modules. Thus, the incremental view graph maintenance algorithm satisfies requirement R4b - Time-Efficiency.

The framework uses changes of base graphs, annotation life cycles, and the impact analysis to determine the search space for modules. The framework uses the impact analysis to determine suspicious, obsolete, and missing annotations that are maintained by modules that dependent on other modules. Thus, the monitoring of base graph changes, the annotation life cycle, and the impact analysis contribute to the satisfaction of requirement R4c - Propagation.

The maintenance modes of modules return suspicious and obsolete annotations as well as artifacts and annotations that are considered as modified, when view modules deleted attached annotations. Depending on the maintenance strategy, the maintenance phases propagate these artifacts and annotations to dependent modules to search for new annotations, delete obsolete annotations, and revise suspicious annotations. Thus, the maintenance modes contribute to the satisfaction of requirement R4c - Propagation and the maintenance phases satisfy requirement R4c - Propagation.

The naive and batch view graph maintenance algorithm propagate all annotations maintained by predecessor modules to successor modules. The naive and batch view graph maintenance algorithm do not limit the propagation to annotations that are created, deleted, and updated during the *current* view graph maintenance. Thus, the naive and batch view graph maintenance algorithm partially satisfy requirement R4c - Propagation, because they always propagate all annotations maintained by predecessor modules to successor view modules.

The incremental view graph maintenance algorithm propagates annotations between view modules that are created, deleted, and updated during the *current* view graph maintenance. Therefore, the incremental view graph maintenance algorithm propagates only annotations that are subject to change in the *current* view graph maintenance. Thus, the incremental view graph maintenance algorithm satisfies requirement R4c - Propagation.

9. Optimized View Graph Maintenance

This chapter describes an optimized view graph maintenance, which aims for an efficient maintenance of annotations that have the same annotation type and mark the same graph nodes by means of roles with the same types, but result from different graph pattern matches. This thesis refers to these annotations as annotation duplicates. This chapter aims for a reduction of this redundancy without losing the capability to enumerate all matches.

Section 9.1 describes the optimization problems that are in the scope of this chapter. Based on the identified problems, Section 9.2 describes the optimization goals of the view graph maintenance algorithms. Section 9.3 refines the elicited requirements based on the derived optimization goals. Section 9.4 describes the optimizations of the maintenance algorithms. Afterwards, Section 9.5 discusses the space and time complexity of the optimized maintenance algorithms. Finally, Section 9.6 discusses the satisfaction of the elicited requirements.

9.1. Problem Analysis

Goal of this thesis is to effectively enumerate all matches of certain graph patterns (cf. G3a - Enumeration of Matches). The framework must maintain these enumerations efficiently (cf. G3c - View Maintenance). For that purpose, this thesis provides a modeling language that enables developers to define the content of view graphs as described in Chapter 6 as well as algorithms that maintain these view graphs as described in Chapter 8.

When developers encapsulate graph patterns by means of view modules, they decide on which graph nodes of the graph pattern matches are marked by means of roles to keep track of their roles in the matches. It depends on the roles that are owned by annotations which information about graph pattern matches can be reused by view modules and end-users. Depending on this design decision, annotations mark at least one graph node or up to all graph nodes of graph pattern matches by means of roles. Thus, view modules and end-users can access these graph nodes and their roles in the graph pattern matches effectively. Annotations mark graph nodes *without* certain roles in graph pattern matches by means of scopes that consist of a default edge type. Therefore, view modules and end-users *cannot* access these graph nodes and their roles in the graph pattern matches effectively.

Based on the concept of roles and scopes, Section 9.1.1 describes the notion of annotation duplicates in view graphs. Afterwards, Section 9.1.2 describes why annotation duplicates result in superfluous graph pattern matchings of dependent view modules.

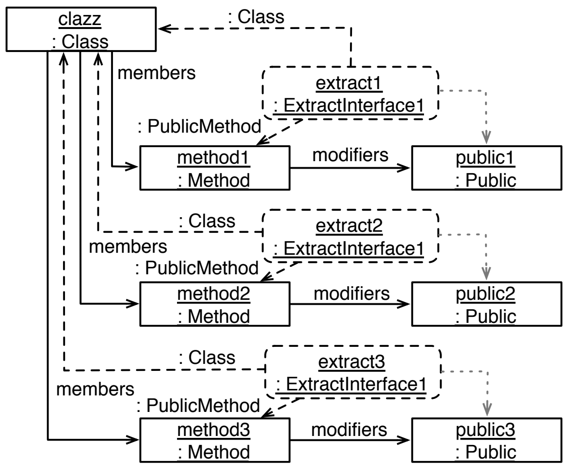
9.1.1. Duplicates of Annotations in View Graphs

View graphs store one annotation per graph pattern match. Only graph nodes that are marked by roles of annotations are visible to dependent view modules and end-users. Graph nodes that are marked by scopes of annotations are *not* visible to dependent view modules and end-users. Therefore, multiple annotations may exist that mark the same graph nodes by means of roles, but mark different graph nodes by means of scopes. Then, these annotations appear to be duplicates, because they mark the same graph nodes by means of roles, when blanking out the scopes of these annotations. This thesis refers to these annotations as *annotation duplicates*. The annotation duplicates result from *different* graph pattern matches. Thus, these duplicates differ in at least one graph node that is marked by a scope. Thus, the NAC in the Create maintenance mode of view modules (cf. Section 7.1.1) does *not* prevent annotation duplicates, because this NAC only ensures that the *same* match is *not* marked more than once.

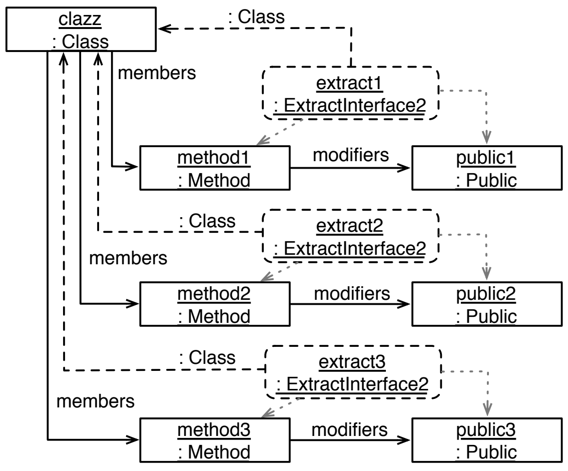
Definition 17 (Annotation Duplicate)
Two annotations are duplicates, when ...

a) they are instances of the same annotation type and
b) they mark the same graph nodes by means of roles that have the same role types and
c) they have the same attribute values.

According to the running example, Figure 9.1 shows two equal base graphs. Each base graph consists of a class with three public methods. The class does *not* implement an interface. Furthermore, Figure 9.1 shows view graphs that differ in the kinds of employed annotations. Note that Figure 9.1 depicts roles as dashed black lines and scopes as dotted gray lines.



(a) Without annotation duplicates



(b) With annotation duplicates

Figure 9.1.: View graphs with annotations for Extract Interface pattern

The view graph in Figure 9.1(a) depicts annotations that are instances of the ExtractInterface1 annotation type. This annotation type describes that its instances mark classes and methods by means of roles for which an interface should be extracted. Therefore, the three annotations in Figure 9.1(a) are *no* annotation duplicates, because they mark the same clazz artifact, but mark different Method artifacts by means of roles. Thus, these annotations are distinguishable by means of the graph nodes that are marked by roles.

In contrast, the view graph in Figure 9.1(b) depicts annotations that are instances of the ExtractInterface2 annotation type. This annotation type describes that its instances mark *only* classes by means of roles for which an interface should be extracted. These annotations do *not* employ *roles* to mark the methods for which an interface should extracted. Thus, these annotations are annotation duplicates, because they mark the same clazz artifact, but do *not* consist of additional roles that mark the different Method artifacts. Consequently, these annotations are *not* distinguishable by means of the graph nodes that are marked by roles.

9.1.2. Superfluous Graph Pattern Matchings

View modules search for graph pattern matches using the search space, which is spanned by annotations that are provided by other modules. When modules receive annotation duplicates, view modules investigate the same search space inefficiently multiple times, because these annotation duplicates mark the same graph nodes with their roles. Modules cannot access the scopes of received annotations and, thus, graph nodes that are marked by scopes do not belong to the search space. When modules receive annotation duplicates, multiple investigations of the same search space are inefficient in space, because this investigation results in similar matches that only differ in the reused annotation duplicates. To overcome these inefficiencies, the search space that is spanned by annotation duplicates must be considered only once to ensure a memory-efficient storing and time-efficient maintenance of view graphs.

For example, Figure 9.1(b) shows the extract1, extract2, and extract3 annotations. These annotations mark the same clazz artifact by means of roles and are annotation duplicates. Dependent view modules can only retrieve this clazz artifact, when they process these annotations. Thus, the clazz artifact defines the search space for the modules that process these annotations. These modules process these annotation duplicates three times, although these annotations provide the same search space. Thus, the annotation duplicates increase the required effort for pattern matching as well as the memory consumption by factor three.

9.2. Optimization Goals

This section describes the goals of the optimization to handle annotation duplicates efficiently.

Space-Efficiency for Annotation Duplicates (OG1)
The proposed optimization must avoid to store redundant parts of annotation duplicates (e. g., their roles) to reduce the memory consumption of view graphs in comparison to view graphs that store each annotation duplicate separately.

Time-Efficiency for Annotation Duplicates (OG2)
The proposed optimization must ensure that the view graph maintenance algorithms investigate the common search space that is spanned by annotation duplicates only once to decrease the execution time of the maintenance algorithms.

Enumeration of all Annotation Duplicates (OG3)
The proposed optimization must ensure that view graphs still store all graph pattern matches that result in annotation duplicates. Thus, the view graphs must keep track of all matches that result in annotation duplicates to ensure that all graph pattern matches can be enumerated.

9.3. Refined Requirements

This section refines the elicited requirements of Section 4.2 based on the identified optimization goals for the view graph maintenance that are described in Section 9.2.

Native Graph Data Model for Graph Views (R1a)
When storing and maintaining annotation duplicates, the framework must preserve the graph structure of view graphs to still benefit from the advantages of employing graphs and graph transformations as native means to store and query graphs.

Memory-Efficient Graph Views (R1b)
The graph views must store graph pattern matches without copying graph nodes to view graphs. Additionally, also matches that result in annotation duplicates must be stored by view graphs, but storing common parts of annotation duplicates must be avoided.

Additional Properties for Stored Graph Pattern Matches (R1c)
View graphs must enable to store additional data values for graph pattern matches, because developers want to enrich views with additional knowledge. These additional data values must be considered, when handling annotation duplicates. For example, two annotations are only duplicates, when all their attributes have equal values.

Encapsulation of Graph Queries (R2a)
View modules encapsulate graph patterns and maintain annotations that mark graph pattern matches. Thus, it must be the responsibility of the view modules to decide on whether a found match or the revision of an annotation results in an annotation duplicate.

Effectiveness of Graph Views (R2b)
View graphs must mark graph pattern matches effectively to preserve the roles of graph nodes in these matches for the reuse of these matches. For that purpose, these roles must be also preserved for matches that result in annotation duplicates.

Reusability of Graph Views (R2c)
The framework must support the reuse of graph pattern matches to enable developers to define graph views on top of other graph views. The same must hold for graph pattern matches that result in annotation duplicates. But, the views must reuse only one representative for a set of annotation duplicates to ensure a memory- and time-efficient handling of these duplicates.

Nested Graph Conditions (R2d)
The framework must support definitions of graph views that have the same expressive power as nested graph conditions (cf. Definition 14). The support of annotation duplicates must preserve this expression power.

Recursion Graph Conditions (R2e)
The framework must support recursive graph conditions (cf. Definition 15). The handling of annotation duplicates must preserve this expressive power.

Monitoring of Graph Changes (R4a)
When end-users change base graphs, these changes may result in new annotation duplicates or obsolete annotation duplicates. Thus, changes of base graphs must be tracked and related to already existing and missing annotation duplicates to maintain annotation duplicates.

Efficiency of Graph View Maintenance (R4b)
The framework must maintain the enumeration of graph pattern matches based on captured base graph changes. These graph changes enable to prune the search spaces during view graph maintenance to local portions of base graphs and view graphs. The handling of annotation duplicates must prune search spaces as well to maintain annotation duplicates efficiently.

Propagation of Graph View Changes (R4c)
The view graphs can dependent on each other. When view graphs change, the content of dependent view graphs must be maintained accordingly. The propagation of changes between view graphs must be also supported for annotation duplicates.

9.4. Maintenance of Annotation Duplicates

This thesis distinguishes two approaches for the maintenance of annotation duplicates. Section 9.4.1 describes an approach that employs a naive duplicate handling, which creates a single annotation for a set of annotation duplicates. Section 9.4.2 describes an approach that employs annotations, which aggregate all matches that result in annotation duplicates.

9.4.1. Naive Duplicate Handling

When view modules employ the naive duplicate handling, view modules only mark graph pattern matches, if they do *not* result in annotation duplicates. The matches that result in annotation duplicates are *not* marked by means of annotations. Thus, this approach does *not* keep track of matches that result in annotation duplicates.

The following sections describe the maintenance modes of the view modules, the maintenance phases of the algorithm, and how the algorithm works for the running example.

Maintenance Modes

The naive duplicate handling adapts the maintenance modes of view modules in such a way that they create and preserve annotations only, if they do *not* result in annotation duplicates.

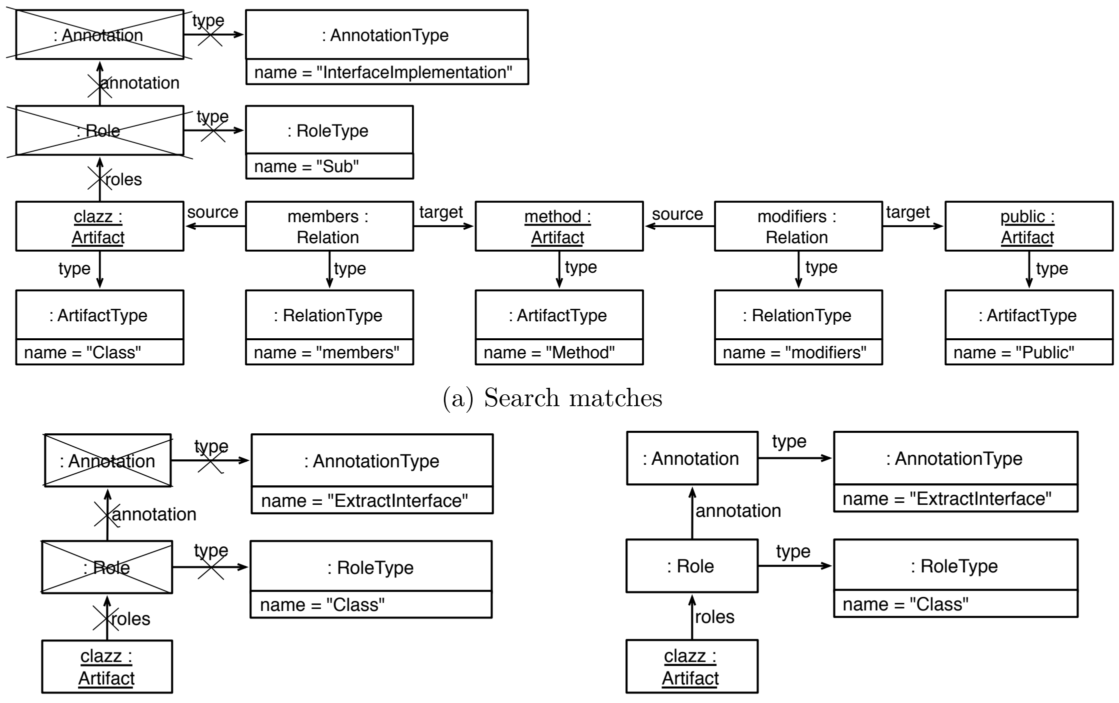
In Create maintenance mode, the view module searches for graph pattern matches and checks for each found match whether it results in an annotation duplicate. If yes, the view module discards the found match and does *not* create an additional annotation.

Algorithm 9.1 Create implementation of view modules with naive duplicate handling

```
Input: Graph nodes of base graphs and view graphs
Output: Created annotations
 1: procedure MODULE_CREATE_DUPLICATE_NAIVE(nodes)
 2:   annotations := ∅
 3:   for each match of the encapsulated pattern for received graph nodes do        //cf. Fig. 9.2(a)
 4:     if match is not already marked by roles then                                //cf. Fig. 9.2(b)
 5:       annotation := CREATE_ANNOTATION(match)                                    //cf. Algorithm 7.1
 6:       annotations := annotations ∪ {annotation}
 7:     else
 8:       existingAnnotations := retrieve annotations that mark match already by means of roles   //cf. Fig. 9.2(c)
 9:       if EQUAL_ATTRIBUTE_VALUES(match, existingAnnotations) then                //cf. Algorithm E.2
10:         continue                                                                //skip creation of annotation
11:       else
12:         annotation := CREATE_ANNOTATION(match)                                  //cf. Algorithm 7.1
13:         annotations := annotations ∪ {annotation}
14:   return annotations
```

Algorithm 9.1 shows the Module_Create_Duplicate_Naive procedure, which describes the naive duplicate handling of view modules that are executed in Create maintenance mode. The procedure makes use of the graph patterns and graph transformations that are depicted by Figure 9.2. Figure 9.2 employs the Extract Interface pattern as example that can be replaced by every other graph pattern as well.

First, the procedure initializes an empty set of annotations. Then, the procedure searches for matches of the pattern that is encapsulated by the view module. For each match, the procedure checks whether the match is *not* already marked by an annotation that marks the same graph nodes by means of roles as the annotation that would be created to mark the found match. Note that this check does *not* consider scopes of annotations. If such an annotation does *not* exist, the procedure creates an annotation to mark the match and adds the annotation to the set of created annotations. If such an annotation exists, the procedure checks whether



(a) Search matches

(b) Already marked match

(c) Retrieve existing annotations

Figure 9.2.: Create implementation for naive duplicate handling

this annotation has the same attribute values as the annotation that would be created to mark the match. For that purpose, the procedure retrieves all existing annotations that mark the graph nodes of the match by means of roles. Note that the procedure does *not* consider scopes to retrieve these annotations. Then, the procedure calls the Equal_Attribute_Values procedure (cf. Algorithm E.2) and passes the found match and the existing annotations. The Equal_Attribute_Values procedure checks whether the set of existing annotations contains an annotation, which has the same attribute values as the annotation that would be created for the found match. If an annotation exists that has the same attribute values, the procedure does *not* create a new annotation, because this annotation would be an annotation duplicate. If no annotation with the same attribute values exists, the procedure creates a new annotation to mark the match and adds the annotation to the set of created annotations. Finally, the procedure returns the created annotations.

In DELETE maintenance mode, the module deletes obsolete annotations including its roles, scopes, and attributes. The DELETE maintenance mode with naive duplicate handling does not differ from the DELETE maintenance mode without duplicate handling. Algorithm 9.2 shows the Module_Delete_Duplicate_Naive procedure, which calls the Module_Delete procedure (cf. Section 7.1.2) to delete annotations, if they are obsolete. Finally, the procedure returns all artifacts and annotations that were marked by deleted annotations.

Algorithm 9.2 Delete implementation of view modules with naive duplicate handling

```
Input: Annotations of view graphs
Output: Annotations that were set obsolete
1: procedure MODULE_DELETE_DUPLICATE_NAIVE(annotations)
2:     return MODULE_DELETE(annotations)                    //cf. Algorithm 7.3
```

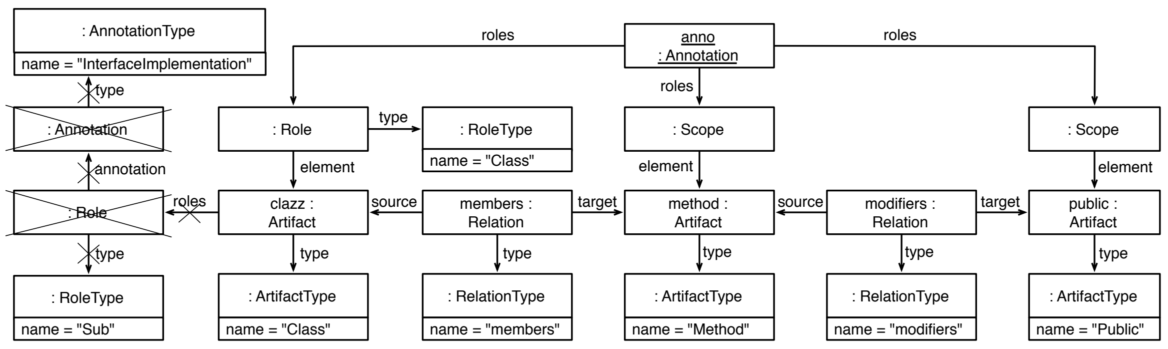
In UPDATE maintenance mode, the view module checks whether annotations still mark matches for the graph pattern that is encapsulated by the module. If the annotation does not mark a graph pattern match anymore, the view module sets the annotation obsolete. Otherwise, the view module preserves the annotation, if the update of the annotation attributes does not result in an annotation duplicate. If the update of the annotation attributes results in an annotation duplicate, the view module sets the annotation obsolete as well.

Algorithm 9.3 Update implementation of view modules with naive duplicate handling

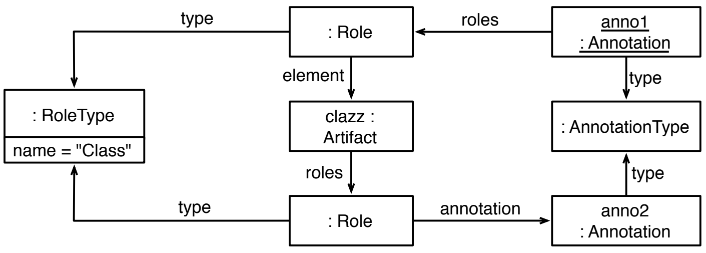
```
Input: Annotations of view graphs
Output: Annotations that were set obsolete
 1: procedure MODULE_UPDATE_DUPLICATE_NAIVE(annotations)
 2:     obsoleteAnnotations := $\emptyset$
 3:     preservedAnnotations := $\emptyset$
 4:     dependentAnnotations := $\emptyset$
 5:     for each annotation in annotations do
 6:         if annotation still marks match of the encapsulated pattern then        //cf. Fig. 9.3(a)
 7:             PRESERVE_ANNOTATION(annotation)                                      //cf. Algorithm 7.6
 8:             existingAnnotations := retrieve annotations that mark match already by means of roles   //cf. Fig. 9.3(b)
 9:             if EQUAL_ATTRIBUTE_VALUES(annotation, existingAnnotations) then     //cf. Algorithm E.3
10:                 OBSOLETE_ANNOTATION(annotation)                                  //cf. Algorithm 7.7
11:                 obsoleteAnnotation := obsoleteAnnotations $\cup$ {annotation}
12:             else
13:                 preservedAnnotations := preservedAnnotations $\cup$ {annotation}
14:                 dependentAnnotations := dependentAnnotations $\cup$ annotation.dependents
15:         else
16:             OBSOLETE_ANNOTATION(annotation)                                      //cf. Algorithm 7.7
17:             obsoleteAnnotation := obsoleteAnnotation $\cup$ {annotation}
18:     return obsoleteAnnotations, preservedAnnotations, dependentAnnotations
```

Algorithm 9.3 describes the Module_Update_Duplicate_Naive procedure, which describes the naive duplicate handling of view modules that are executed in UPDATE maintenance mode. The procedure makes use of the graph patterns and graph transformations that are depicted by Figure 9.3. Figure 9.3 employs the Extract Interface pattern as example that can be replaced by every other graph pattern as well.

The procedure receives a set of annotations. Then, the procedure initializes an empty set of obsolete, preserved, and dependent annotations. For each annotation in the set of received annotations, the procedure checks whether the graph nodes that are marked by the annotation still satisfy the pattern that is encapsulated by the view module. If yes, the Preserve_Annotation procedure updates the attribute values of the annotation by re-evaluating the expressions of the corresponding attribute assignments. Therefore, attribute values of annotations may change in a way that they are equal to the attribute values of annotations with the same type, which mark the same graph nodes as the preserved annotation by means of roles. For that purpose, the Module_Update_Duplicate_Naive procedure looks up all existing annotations that have the same annotation type as the preserved annotation and mark the same graph nodes as the preserved annotation by means of roles. Then, the procedure checks whether one of these existing annotations has the same attribute values as the preserved annotation. For that purpose, the procedure calls the Equal_Attribute_Values procedure (cf. Algorithm E.3), which checks whether an annotation already exists that has the same attribute values as the preserved annotation. If such an annotation already exists, the procedure calls the Obsolete_Annotation procedure to set the preserved annotation obsolete and adds the annotation to the set of obsolete annotations, afterwards. If such an annotation does *not* already exist, the procedure preserves the annotation, adds the annotation to the set of preserved annotations, and adds the annotations that dependent on the preserved annotation to the set of dependent annotations.



(a) Does marked match satisfy pattern?



(b) Retrieve alternative annotations

Figure 9.3.: Update implementation for naive duplicate handling

If the graph nodes that are marked by a received annotation do not satisfy the pattern that is encapsulated by the view module, the procedure sets the annotation obsolete and adds the annotation to the set of obsolete annotations. Finally, the procedure returns the set of obsolete, preserved, and dependent annotations.

Maintenance Phases

This section describes the maintenance phases of the batch and incremental view graph maintenance for the naive duplicate handling. In general, this view graph maintenance employs the same order of maintenance phases as the view graph maintenance without naive duplicate handling. Algorithm 8.10 and Algorithm 8.11 describe the maintenance phases of the batch and incremental maintenance algorithms.

The Update maintenance phase checks whether annotations still mark graph nodes that satisfy the patterns, which are encapsulated by view modules. When view modules detect obsolete annotations in Update maintenance mode, the modules set these annotations obsolete. Otherwise the view modules preserve the annotations, if the update of the annotation attributes does *not* result in annotation duplicates. If the update of the annotation attributes results in annotation duplicates, the view modules set the preserved annotations obsolete as well.

The Delete maintenance phase deletes annotations, when a) annotations are obsolete due to deleted graph nodes or b) the annotations are set obsolete during the Update maintenance phase. When the Delete maintenance phase executes a view module in Delete maintenance mode, the view module deletes obsolete annotation including its roles, scopes, and attributes. Note that the graph nodes that were marked by the deleted annotations are considered as modified and, therefore, are input to the Create maintenance phase.

The Create maintenance phase creates annotations, when a) PACs are satisfied due to the creation or modification of graph nodes or b) NACs are satisfied due to the deletion or modification of graph nodes. When the CREATE maintenance phase executes a view module in CREATE maintenance mode and the view module finds a graph pattern match, the view module checks whether the match results in an annotation duplicate. If such an annotation duplicate already exists, the view module discards the found match and does *not* create an additional annotation. Otherwise, the module creates an annotation to mark the match.

The artifacts and annotations that were marked by the deleted annotation are considered as modified and, thus, are input to the CREATE maintenance phase. In doing so, the view modules find graph pattern matches that were not marked by annotations before, because these matches resulted in annotation duplicates of the deleted annotation, before. Thus, view modules that are executed in CREATE maintenance mode create annotations that were previously considered as annotation duplicates.

If the CREATE maintenance phase creates new annotations, the maintenance algorithms employ an additional sequence of UPDATE, DELETE, and CREATE maintenance phases to determine annotations that become obsolete due to dissatisfied NACs.

Running Example

According to the running example, Figure 9.4(a) depicts a clazz artifact that owns the methods method1, method2, and method3 with the public modifiers public1, public2, and public3. The extract1 annotation marks a match of the Extract Interface pattern (cf. Figure 3.9). The match consists of the artifacts clazz, method1, and public1 artifacts. The artifacts clazz, method2, and public2 constitute also a match of the Extract Interface pattern, but they are not marked by an annotation, because this annotation would be an annotation duplicate that marks the same clazz artifact by means of the role with Class role type as the extract1 annotation and marks the method2 and public2 artifact by means of scopes. The same argument holds for the clazz, method3, and public3 artifacts that are also a match for the Extract Interface pattern.

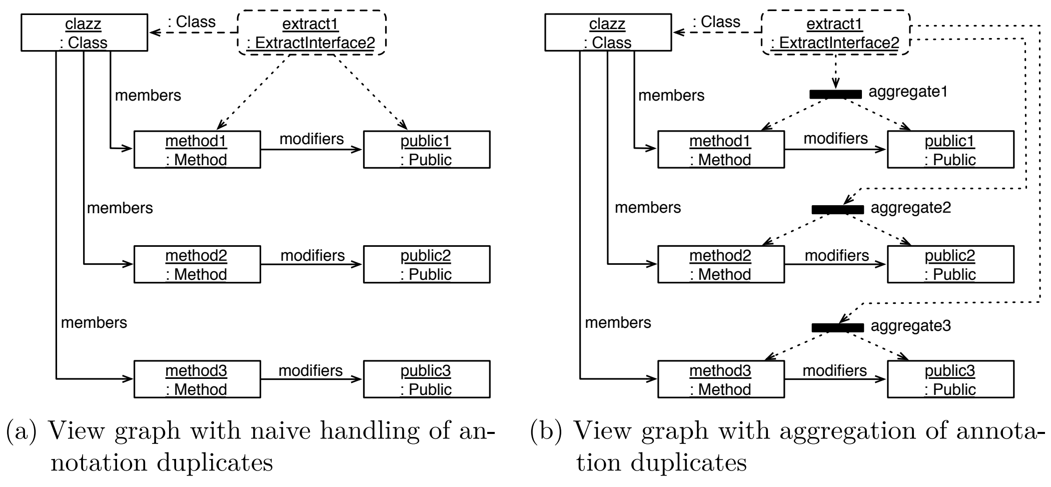
When end-users add an additional method4 artifact with Method artifact type that owns a public4 artifact with Public artifact type to the clazz artifact, the clazz, method4, and public4 artifacts constitute an additional match for the Extract Interface pattern. Before creating the annotation for this match, the view module checks whether the clazz artifact is *not* already marked by an annotation with ExtractInterface2 annotation type by means of a role with Class role type. In this example, the extract1 annotation dissatisfies this conditions. Thus, the view module creates no additional annotation that marks the match, which consists of the clazz, method4, and public4 artifacts, because this annotation is an annotation duplicate.

When end-users remove the method1 and / or public1 artifact, the extract1 annotation is obsolete and the responsible view module deletes the annotation. Then, the framework considers the clazz artifact as modified, because the extract1 annotation was removed from the clazz artifact. Therefore, the clazz artifact is input to the CREATE maintenance phase and the view module searches for additional matches in the context of this clazz artifact. Next, the view module finds either the match that consists of the clazz, method2, and public2 artifacts or the clazz, method3, public3 artifacts. The view module marks only one of both graph pattern matches, because then the second graph pattern match results in an annotation duplicate.

9.4.2. Duplicate Handling with Aggregation

The duplicate handling with aggregation employs special annotations that keep track of all graph pattern matches that result in annotation duplicates. Therefore, these matches can be retrieved easily without additional search, later on.

The following sections extend the concept of annotations by so called aggregations, describe



(a) View graph with naive handling of annotation duplicates (b) View graph with aggregation of annotation duplicates

Figure 9.4.: View graphs with annotations for Extract Interface pattern

the impact analysis that derives missing, obsolete, and suspicious annotations that employ such aggregations, describe the maintenance modes of view modules that create, delete, and update these annotations, describe the maintenance algorithms that make use of these adapted maintenance modes, and demonstrate the adapted concepts by means of the running example.

Adaptation of View Graph Metamodel

The left-hand side of Figure 9.5 shows the adapted view graph metamodel as UML class diagram. Figure 9.5 depicts parts of the metamodel that are already introduced in gray color. The metamodel describes that annotations can additionally consist of special abstract roles called scope aggregations. These scope aggregations own multiple scopes that mark artifacts and annotations that do not have special roles in graph pattern matches.

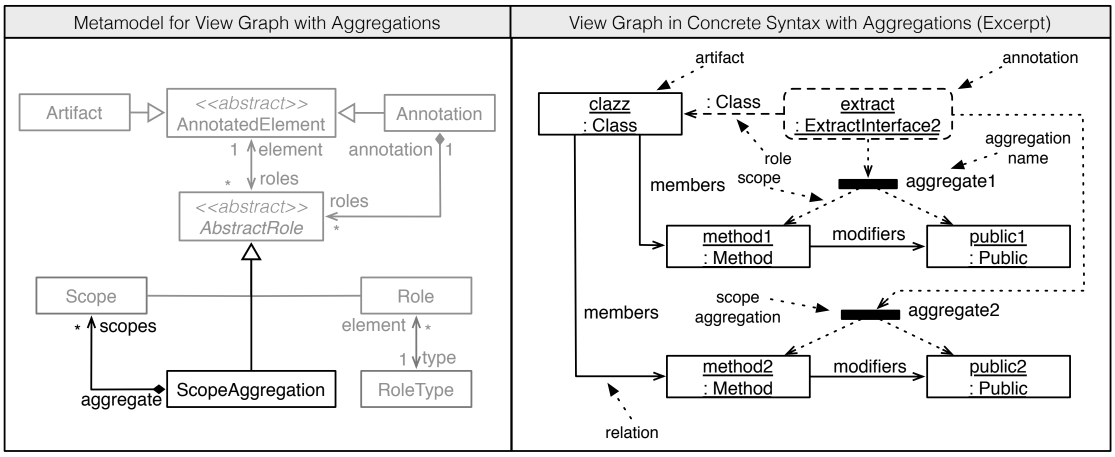


Figure 9.5.: Metamodel for view graphs with aggregation as UML class diagram (left) and view graph in concrete syntax (right) as adapted UML object diagram

Adaptation of View Graph Syntax

The right-hand side of Figure 9.5 shows a view graph that depicts the concrete syntax of scope aggregations. This thesis depicts scope aggregations as black solid bars that own scopes. This thesis depicts scopes as dotted lines. Optionally, scope aggregations can consist of a name.

The right-hand side of Figure 9.5 shows two matches of the Extract Interface pattern. The first match consists of the clazz, method1, and public1 artifacts. The second match consists of the clazz, method2, and public2 artifacts. The extract annotation marks the first match by means of the Class role and the aggregate1 scope aggregation. The aggregate1 aggregation owns scopes that mark the method1 and public1 artifact. The extract annotation marks the second match by means of the Class role and the aggregate2 scope aggregation. The aggregate2 aggregation owns scopes that mark the method2 and public2 artifact.

Impact Analysis

As described in Section 8.4, the framework employs an impact analysis based on captured modification events of base graphs to determine the state of annotations in their life cycle. In general, the impact analysis works in the same way for annotations with scope aggregations.

The impact analysis derives *missing* annotations from added and modified artifacts of base graphs. Furthermore, the impact analysis derives missing annotation duplicates from missing scope aggregations. A scope aggregation is missing, if a scope aggregation does not exist in view graphs although it must exist, because a graph pattern match, which results in an annotation duplicate, exists that is currently not represented in the view graph.

The impact analysis derives *obsolete* annotations from deleted artifacts of base graphs by traversing roles, scopes, and scope aggregations from deleted artifacts to annotations that own the roles and scope aggregations. An annotation duplicate that is marked by a scope aggregation is obsolete, if a) the scope aggregation owns a dangling scope that does not reference an artifact or annotation, or b) the annotation that owns the scope aggregation owns a dangling role that does not reference an artifact or annotation.

The impact analysis derives *suspicious* annotations from modified artifacts of base graphs by traversing roles, scopes, and scope aggregations from modified artifacts to annotations that own the roles and scope aggregations. An annotation duplicate that is marked by a scope aggregation is suspicious, if a) the scope aggregation owns a scope that references a modified artifact or annotation or b) the annotation that owns the scope aggregation owns a role that references a modified artifact or annotation.

Maintenance Modes

This section describes the maintenance modes of view modules that employ scope aggregations to keep track of graph pattern matches that result in annotation duplicates. The following paragraphs describe the adapted CREATE, DELETE, and UPDATE maintenance modes.

In CREATE maintenance mode, view modules search for graph pattern matches and mark these matches by means of annotations. For each match that would result in an annotation duplicate, the view modules create a scope aggregation, which marks the graph nodes without explicit roles in the match, and add this scope aggregation to the already existing annotation.

Algorithm 9.4 shows the Module_Create_Duplicate_Aggregation procedure, which describes the handling of annotation duplicates by means of scope aggregations in CREATE maintenance mode. The procedure makes use of the graph patterns and graph transformations that are depicted by Figure 9.6. Figure 9.6 employs the Extract Interface pattern as example that can be replaced by every other pattern as well.

The procedure receives graph nodes that define the search space in which the view module searches for graph pattern matches. First, the procedure initializes an empty set of created annotations. Then, the procedure searches for matches of the graph pattern that is encapsulated by the view module. For each found match, the procedure checks whether no annotation with

Algorithm 9.4 Create implementation of view modules with scope aggregations

```
Input: Graph nodes of base graphs and view graphs
Output: Created annotations
 1: procedure MODULE__CREATE__DUPLICATE__AGGREGATION(nodes)
 2:   annotations := ∅
 3:   for each match of the encapsulated pattern for the received graph nodes do        //cf. Fig. 9.6(a)
 4:     if match is not already marked by roles then                                      //cf. Fig. 9.6(b)
 5:       annotation := CREATE__ANNOTATION__AGGREGATE(match)                            //cf. Algorithm E.5
 6:       annotations := annotations ∪ {annotation}
 7:     else
 8:       existingAnnotations := retrieve annotations that mark match via roles         //cf. Fig. 9.6(c)
 9:       annotation := RETRIEVE__ANNOTATION__DUPLICATE(match, existingAnnotations)     //cf. Algorithm E.4
10:       if annotation exists then
11:         ADD__SCOPE__AGGREGATION(match, annotation)                                  //cf. Algorithm E.6
12:       else
13:         annotation := CREATE__ANNOTATION__AGGREGATE(match)                          //cf. Algorithm E.5
14:         annotations := annotations ∪ {annotation}
15:   return annotations
```

the same annotation type as the output connector of the view module exists that marks the same graph nodes of the match by means of roles. Note that scopes and scope aggregations are not part of this check.

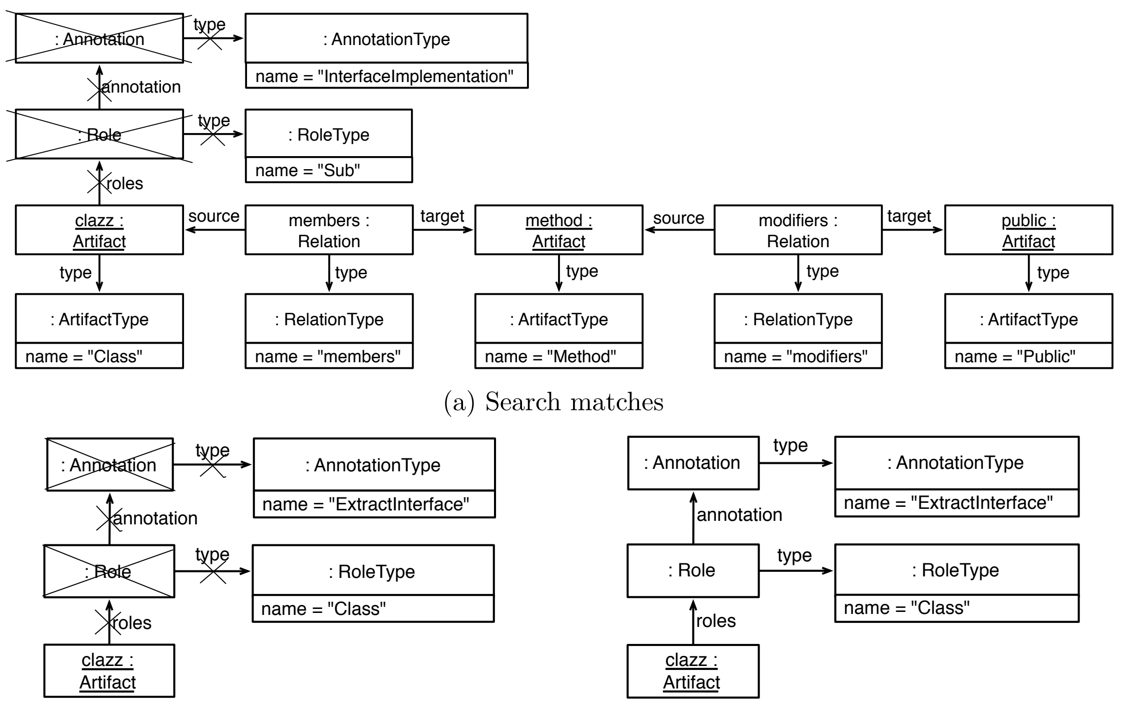
If such an annotation does not already exist, the procedure marks the match. For that purpose, the procedure calls the Create_Annotation_Aggregate procedure (cf. Algorithm E.5) and passes the found pattern match. This procedure creates an annotation with a scope aggregation for the found match. Then, the Module_Create_Duplicate_Aggregation procedure adds the created annotation to the set of created annotations.

If annotations exist that mark the graph nodes of the found match by means of roles already, the procedure retrieves these annotations. Then, the procedure checks whether the set of retrieved existing annotations contains an annotation that has the same attribute values as the annotation that would be created for the found match. If such an annotation already exists, the annotation that would be created is an annotation duplicate. Therefore, the procedure does not create a new annotation. Instead, the procedure adds the graph nodes, which would be marked by the new annotation by means of scopes, to a scope aggregation that is added to the already existing annotation. For that purpose, the procedure calls the Add_Scope_Aggregation procedure (cf. Algorithm E.6) and passes the found match as well as the already existing annotation. If such an annotation does not exist, the procedure creates a new annotation. For that purpose, the procedure calls the Create_Annotation_Aggregate procedure (cf. Algorithm E.5) and passes the found match. Then, the Module_Create_Duplicate_Aggregation procedure adds the created annotation to the set of created annotations. Finally, the procedure returns all created annotations.

In DELETE maintenance mode, view modules remove scope aggregations from annotations or remove annotations, if they do not consist of scope aggregations or their roles are dangling.

Algorithm 9.5 shows the Module_Delete_Duplicate_Aggregation procedure, which describes the handling of annotation duplicates by means of scope aggregations. The procedure makes use of the graph patterns and graph transformations that are depicted by Figure 9.7.

The Module_Delete_Duplicate_Aggregation procedure receives annotations that must be deleted, when they are obsolete. First, the procedure initializes an empty set of marked nodes and an empty set of dependent annotations. For each annotation in the set of received annotations, the procedure checks whether the annotation has at least one scope aggregation, which owns a scope that does not reference an artifact or annotation anymore. If yes, the



(a) Search matches

(b) Already marked match

(c) Retrieve existing annotations

Figure 9.6.: Create implementation for annotations with scope aggregations

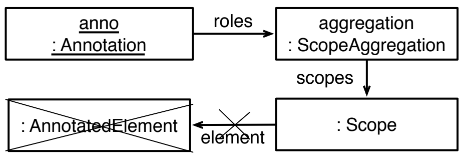
scope aggregation is obsolete. Therefore, the procedure looks up all obsolete scope aggregations of the annotation. For each obsolete scope aggregation, the procedure adds all artifacts and annotations that are marked by the scope aggregation and the roles of the annotation that owns the scope aggregation to the set of marked nodes. Afterwards, the procedure calls the Delete_Scope_Aggregation procedure (cf. Algorithm E.7) and passes the obsolete scope aggregation to delete it. Then, the procedure checks whether the annotation consists of additional scope aggregations. If no, the annotation is obsolete. Therefore, the procedure adds annotations that dependent on the obsolete annotation to the set of dependent annotations. Afterwards, the procedure deletes the annotation by calling the Delete_Annotation procedure and passing the obsolete annotation.

If a received annotation has a dangling role, the procedure adds all artifacts and annotations that are marked by roles and scope aggregations of the annotation to the set of marked nodes. Then, the procedure adds all annotations that dependent on the obsolete annotation to the set of dependent annotations. For each scope aggregation of the annotation, the procedure removes all scopes from the scope aggregation and deletes the scope aggregation itself. Afterwards, the procedure calls the Delete_Annotation procedure and passes the obsolete annotation to delete the annotation. Finally, the procedure returns a) the graph nodes that were previously marked by deleted annotations and b) the annotations that dependent on the deleted annotations.

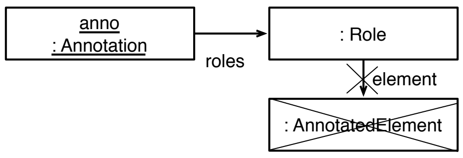
In Update maintenance mode, view modules check whether annotations and their scope aggregations still mark matches of the graph patterns that are encapsulated by the view modules. These modules either preserve annotations, move scope aggregations to other already existing annotations, extract new annotations from scope aggregations, or set scope aggregations obsolete.

Algorithm 9.5 Delete implementation of view modules with scope aggregations

```
Input: Annotations of view graphs
Output: Graph nodes referenced by deleted annotations (duplicates) and dependent annotations
 1: procedure MODULE_DELETE_DUPLICATE_AGGREGATION(annotations)
 2:   markedNodes := ∅
 3:   dependentAnnotations := ∅
 4:   for each annotation in annotations do
 5:     if annotation has dangling scope then                                   //cf. Fig. 9.7(a)
 6:       obsoleteAggregations := retrieve obsolete scope aggregations of annotation   //cf. Fig. 9.7(a)
 7:       for each aggregation in obsoleteAggregations do
 8:         markedNodes := markedNodes ∪ {graph nodes marked by roles and scope aggregations}
 9:         DELETE_SCOPE_AGGREGATION(aggregation)                               //cf. Algorithm E.7
10:       if annotation has no scope aggregations anymore then
11:         dependentAnnotations := dependentAnnotations ∪ annotation.dependents
12:         DELETE_ANNOTATION(annotation)                                      //cf. Algorithm 7.3
13:     if annotation has dangling role then                                    //cf. Fig. 9.7(b)
14:       markedNodes := markedNodes ∪ {graph nodes marked by roles and scope aggregations}
15:       dependentAnnotations := dependentAnnotations ∪ annotation.dependents
16:       for each scopeAggregation of annotation do
17:         DELETE_SCOPE_AGGREGATION(scopeAggregation)                          //cf. Algorithm E.7
18:       DELETE_ANNOTATION(annotation)                                        //cf. Algorithm 7.3
19:   return markedNodes, dependentAnnotations
```



(a) Dangling scope aggregation



(b) Dangling role

Figure 9.7.: Delete implementation for annotations with scope aggregations

Algorithm 9.6 shows the Module_Update_Duplicate_Aggregation procedure, which describes the handling of annotation duplicates by means of scope aggregations in UPDATE maintenance mode. The procedure makes use of the graph patterns and graph transformations depicted by Figure 9.8. Figure 9.8 employs the Extract Interface pattern as example that can be replaced by every other pattern.

The procedure receives annotations that must to be checked whether they still mark matches of the graph pattern that is encapsulated by the view module. First, the procedure initializes empty sets of obsolete, preserved, extracted, and dependent annotations. For each annotation in the set of received annotations, the procedure checks whether all scope aggregations of the annotation satisfy the pattern, which is encapsulated by the module, and whether all scope aggregations result in the equal attribute values. If yes, the procedure updates the attribute values of the annotation, adds the annotation to the set of preserve annotations, adds all dependent annotations to the set of dependent annotations, and continues with the next annotation in the set of received annotations. If no, the procedure checks each scope aggregation of the annotation on its own. The procedure checks for each scope aggregation whether the scopes of the scope aggregation and the roles of the annotation that owns the scope aggregation mark graph nodes that satisfy the pattern that is encapsulated by the view module. If yes, the procedure considers multiple cases.

First, the procedure checks whether the attribute assignments evaluate to the same values as the attribute values that are stored by the annotation that owns the scope aggregation. If yes, the procedure preserves the scope aggregation and adds the annotations that dependent

Algorithm 9.6 Update implementation of view modules with scope aggregations

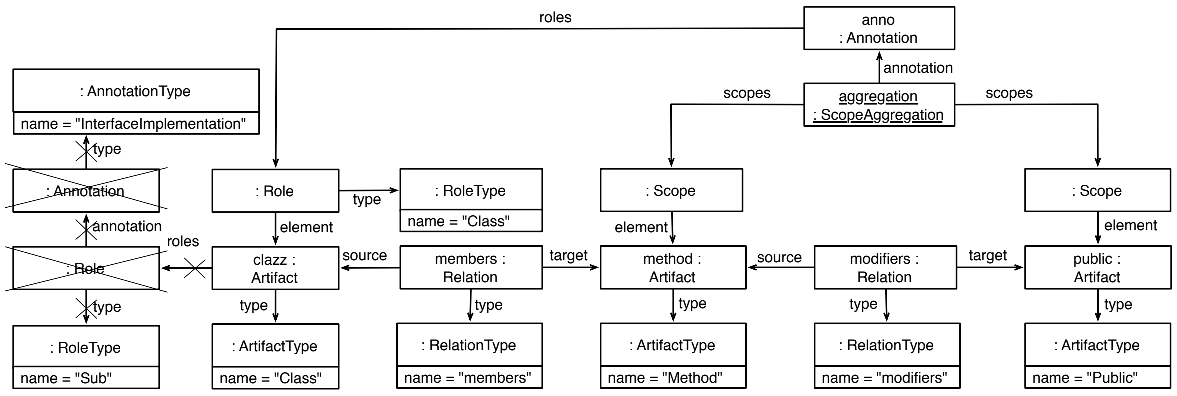
```
Input: Annotations of view graphs
Output: Obsolete annotations, preserved, extracted, and dependent annotations
 1: procedure MODULE__UPDATE__DUPLICATE__AGGREGATION(annotations)
 2:   obsoleteAnnotations := $\emptyset$
 3:   preservedAnnotations := $\emptyset$
 4:   extractedAnnotations := $\emptyset$
 5:   dependentAnnotations := $\emptyset$
 6:   for each annotation in annotations do
 7:     if all scope aggregations satisfy the graph pattern and result in equal attribute values then  //cf. Fig. 9.8(a)
 8:       set new attribute values of annotation
 9:       preservedAnnotations := preservedAnnotations $\cup$ {annotation}
10:       dependentAnnotations := dependentAnnotations $\cup$ annotation.dependents
11:       continue
12:     else                                              //Check each scope aggregation on its own
13:       for each scope aggregation of annotation do
14:         if scope aggregation of annotation satisfies the encapsulated pattern then   //cf. Fig. 9.8(a)
15:           if scope aggregation results in unchanged attributes values then   //Preserve scope aggregation
16:             dependentAnnotations := dependentAnnotations $\cup$ annotation.dependents
17:           else                                        //Attribute values changed
18:             if annotation with equal attribute values exists then   //Extract by move to existing annotation
19:               move scope aggregation to annotation with equal attribute values   //cf. Fig. 9.8(b)
20:             else                                      //Extract by create new annotation
21:               extractedAnnotation := extract annotation   //cf. Fig. 9.8(c)
22:               extractedAnnotations := extractedAnnotations $\cup$ {extractedAnnotation}
23:             if annotation becomes obsolete then
24:               obsoleteAnnotations := obsoleteAnnotations $\cup$ {annotation}
25:         else                                          //Set scope aggregation obsolete
26:           OBSOLETE__SCOPE__AGGREGATION(aggregation)   //cf. Algorithm E.8
27:           obsoleteAnnotations := obsoleteAnnotations $\cup$ {annotation}
28:   return obsoleteAnnotations, preservedAnnotations, extractedAnnotations, dependentAnnotations
```

on the currently revised annotation to the set of dependent annotations. If no, the procedure extracts the scope aggregation by moving the scope aggregation to an existing annotation or creating a new annotation.

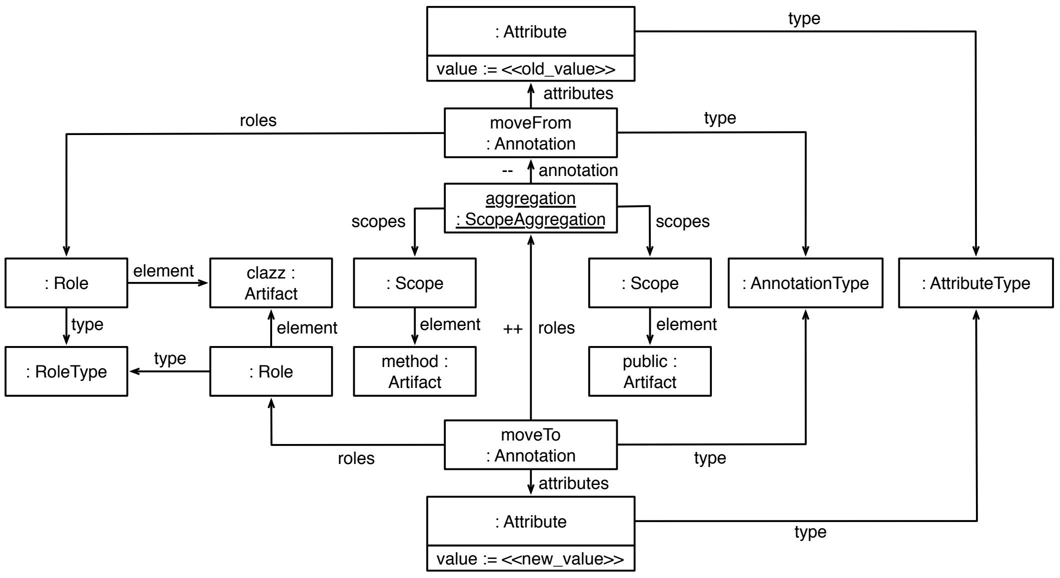
The procedure moves the scope aggregation to an existing annotation, when this annotation marks the same graph nodes by means of roles, has the same attribute values as the attribute values that are computed for the scope aggregation, and has the same annotation type. For that purpose, the procedure removes the scope aggregation from the currently revised annotation and adds the scope aggregation to the other existing annotation.

The procedure creates a new annotation for the scope aggregation, when such an annotation does not exist. For that purpose, the procedure copies all roles of the currently revised annotation, adds the scope aggregation to the extracted annotation by removing the scope aggregation from the current annotation, adds the attributes to the extracted annotation, and sets the attribute values by evaluating the corresponding attribute assignments. Afterwards, the procedure adds the extracted annotation to the set of extracted annotations.

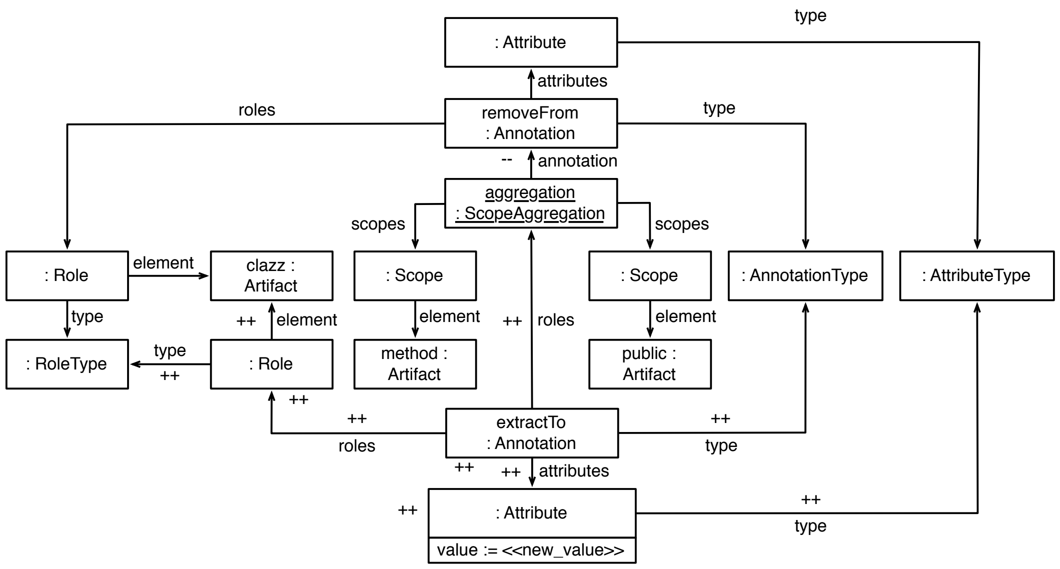
If the graph nodes that are marked by the scope aggregation of the current annotation do *not* satisfy the graph pattern anymore, the procedure sets the scope aggregation obsolete. For that purpose, the procedure calls the Obsolete_Scope_Aggregation procedure (cf. Algorithm E.8), which detaches the graph nodes from all scopes that are owned by the passed scope aggregation. Then, the procedure adds the obsolete annotation to the set of obsolete annotations. Finally, the procedure returns the obsolete, preserved, extracted, and dependent annotations.



(a) Already marked match?



(b) Move scope aggregation to another annotation



(c) Create new annotation for scope aggregation

Figure 9.8.: Update implementation for annotations with scope aggregations

Maintenance Phases

When view modules employ scope aggregations, the batch and incremental view graph maintenance algorithms employ the same order of maintenance phases as for the view graph maintenance without duplicate handling.

Algorithm 9.7 Batch maintenance algorithm with duplicate aggregation

```
Input: All artifacts and annotations of base graphs and view graphs
Output: Consistent view graphs
 1: procedure BATCH_MAINTENANCE_DUPLICATE_AGGREGATION(artifacts, annotations)
 2:     created := ∅
 3:     repeat
 4:         extracted := UPDATE(topological_sort(annotations))
 5:         DELETE(topological_sort(annotations))
 6:         created := extracted ∪ CREATE(artifacts)
 7:     until created = ∅
```

The Algorithm 9.7 describes the Batch_Maintenance_Duplicate_Aggregation procedure. The procedure considers all annotations as suspicious and, therefore, passes all annotations to the Update maintenance phase. In contrast to the original view graph maintenance without duplicate handling, the Update maintenance phase returns annotations that are extracted from annotations due to changed attribute values of annotations. Then, the procedure stores these extracted annotations. Afterwards, the procedure considers all annotations as obsolete and, therefore, passes all annotations to the Delete maintenance phase to delete annotations and scope aggregations that are obsolete. Then, the procedure considers all artifacts of base graphs and annotations, which are maintained by predecessor view modules, as search space for the Create maintenance phase. The Create maintenance phase creates new annotations or adds scope aggregations to annotations for each found graph pattern match. The Create maintenance phase returns all created annotations. Then, the procedure adds the returned annotations to the set of extracted annotations. The procedure repeats the maintenance phases until the Update and Create phases do *not* create or extract annotations anymore, because these annotations can dissatisfy complex NACs. The detection of dissatisfied complex NACs is implemented by the Update phase.

Algorithm 9.8 Incremental maintenance algorithm with duplicate aggregation

```
Input: All modification events of base graphs
Output: Consistent view graphs
 1: procedure INCREMENTAL_MAINTENANCE_DUPLICATE_AGGREGATION(events)
 2:     suspicious := ∅
 3:     repeat
 4:         suspicious:= suspicious ∪ SUSPICIOUS_ANNOTATIONS(events)
 5:         obsoletes, preserved, extracted := UPDATE(suspicious)
 6:         obsoletes := obsoletes ∪ OBSOLETE_ANNOTATIONS(events)
 7:         changed := DELETE(obsoletes)
 8:         changed := changed ∪ MISSING_ANNOTATIONS(events)
 9:         created := CREATE(changed ∪ preserved ∪ extracted)
10:         events := ∅
11:         suspicious := reachabilitySuspicious(extracted ∪ created)
12:     until suspicious = ∅
```

The Algorithm 9.8 describes the Incremental_Maintenance_Duplicate_Aggregation procedure. The procedure derives suspicious, obsolete, and missing annotations from modifications of base graphs. The procedure passes suspicious annotations to the Update phase. The Update phase returns obsolete, preserved, and extracted annotations. Then, the procedure passes obsolete annotations to the Delete phase and stores the graph nodes that were marked by deleted annotations. Afterwards, the procedure derives changed graph nodes from modification events of base graphs. The procedure passes the changed graph nodes as well as the preserved and extracted annotations to the CREATE phase to search for new graph pattern matches. The CREATE phase also processes preserved annotations, because they may have attribute values that now satisfy attribute constraints of view modules. Furthermore, the CREATE phase also processes extracted annotations, because they may satisfy graph patterns of successor view modules. For each found match, the procedure creates a new annotation or scope aggregation to mark this new match. Then, the procedure uses the annotations that are extracted and created during the UPDATE and CREATE phase to derive suspicious annotations and repeats all maintenance phases, if additional suspicious annotations exist.

Running Example
According to the running example, Figure 9.4(b) depicts a clazz artifact that owns the method1, method2, and method3 artifacts with the public1, public2, and public3 artifacts. The extract1 annotation marks three matches for the Extract Interface pattern (cf. Figure 3.9). The extract1 annotation employs the aggregate1 scope aggregation to mark that the method1 and public1 artifacts belong to the first match, the aggregate2 scope aggregation to mark that the method2 and public2 artifacts belong to the second match, and the aggregate3 scope aggregation to mark that the method3 and public3 artifacts belong to the third match.

When the end-users add an additional method4 artifact with Method artifact type that owns a public4 artifact with Public artifact type to the clazz artifact, the clazz artifact and the new method4 and public4 artifacts constitute an additional match for the Extract Interface pattern. Before creating the annotation for this match, the view module checks whether the clazz artifact is already marked by an annotation with ExtractInterface2 annotation type by means of a role with Class role type. In this example, the extract1 annotation satisfies this condition. Thus, the view module adds an aggregate4 scope aggregation, which marks the new method4 and public4 artifacts, to the extract1 annotation.

When the end-users remove the method1 and / or public1 artifact, the graph pattern match that is marked by the aggregate1 scope aggregation is obsolete. Then, the view module removes the aggregate1 scope aggregation from the extract1 annotation. The view module preserves the extract1 annotation, the aggregate2 scope aggregation, and the aggregate3 scope aggregation, because they mark graph nodes that still satisfy the Extract Interface pattern. If also these two scope aggregations become obsolete, the view graph maintenance removes both scope aggregations and the extract1 annotation as well, because all matches disappeared.

9.5. Complexity

This section describes the space and time complexity of the view graph maintenance, when view modules employ a) no duplicate handling, b) naive duplicate handling, and c) duplicate handling with aggregation. Appendix D provides a detailed discussion.

No Duplicate Handling
Chapter 8 describes the original maintenance algorithms. These algorithms create one annotation for each match that results in an annotation duplicate. Thus, these algorithms employ no duplicate handling. Consequently, the number of annotations stored by view graphs depends on the number of matches that result in annotation duplicates. When the number of these matches doubles, then also the number of annotations doubles, which are required to mark these matches.

In the original algorithms, view modules exchange annotation duplicates, which span equal search spaces for dependent modules. Thus, the effort for the graph pattern matching of dependent modules multiplies by the number of exchanged annotation duplicates. Furthermore, these annotation duplicates result in equivalent matches of dependent modules, because these annotation duplicates mark the same graph nodes by means of roles. The found matches only differ in the reused annotation duplicates. Thus, these annotation duplicates also result in additional memory overhead of dependent modules for storing equivalent matches.

Naive Duplicate Handling

Section 9.4.1 describes a naive handling of annotation duplicates. The algorithm creates only one annotation for a set of annotation duplicates. Thus, the number of matches, which result in annotation duplicates, have no impact on the number of annotations that are stored by view graphs. Therefore, modules exchange also only one annotation for a set of annotation duplicates. Therefore, the effort for the pattern matching of dependent modules is independent from the number of matches, which result in annotation duplicates. However, the naive handling of annotation duplicates loses the capability to enumerate all matches that result in annotation duplicates, because the approach does not keep track of these matches.

Duplication Handling with Aggregation

Section 9.4.2 describes a duplicate handling with aggregation of graph pattern matches. This algorithm creates one annotation for a set of annotation duplicates and keeps track of the matches that result in these annotation duplicates. Thus, the memory consumption of a single annotation depends on the number of these matches. When the number of these matches doubles, then also the number of aggregations stored by the annotation doubles. Furthermore, modules exchange only one annotation for a set of annotation duplicates. Thus, the same arguments hold for the effort of the graph pattern matching of dependent modules as for the naive duplicate handling. The effort is independent from the number of aggregated matches.

9.6. Discussion

This chapter uncovers the challenge of annotation duplicates that result in inefficient search spaces for view modules and equivalent annotations, which result in redundantly stored information. For that purpose, this chapter introduces the concept of scope aggregations that enable annotations to aggregate matches that result in annotation duplicates. Then, view modules process only one annotation per set of annotation duplicates and investigate their common search space only once.

Next, Section 9.6.1 describes which concepts satisfy the optimization goals of this chapter. Afterwards, Section 9.6.2 discusses why the extended concepts satisfy the refined requirements.

9.6.1. Optimization Goals

Section 9.2 describes three optimization goals concerning the handling of annotation duplicates. The first goal OG1 - Space-Efficiency aims for the reduction of the memory consumption of view graphs. The second goal OG2 - Time-Efficiency aims for the efficient processing of annotation duplicates by investigating their common search space only once. The third goal OG3 - Enumerate Duplicates aims for the enumeration of all graph pattern matches, also when they result in annotation duplicates. Table 9.1 maps the optimization goals to the concepts of the optimized view maintenance.

Table 9.1.: Mapping the optimization goals to the concepts of the optimized maintenance

		Optimization Goals		
		OG1 - Space-Efficiency	OG2 - Time-Efficiency	OG3 - Enumerate Duplicates
View Graph	Scope Aggregation	●	◐	●
	Impact Analysis	○	◐	○
Algorithmic Framework	Maintenance Modes	●	◐	●
	Maintenance Phases	●	●	●
View Graph Maintenance	No Duplicate Handling	○	○	●
	Naive Duplicate Handling	●	●	○
	Aggregation of Duplicates	●	●	●

covered: ●; partially covered: ◐; not covered: ○

The scope aggregations enable view modules to aggregate graph pattern matches, which result in annotation duplicates, as one annotation to store common roles of annotation duplicates only once. Thus, the scope aggregations satisfy the goal OG1 - Space-Efficiency. Furthermore, the scope aggregations contribute to the goal OG2 - Time-Efficiency, because they enable view modules to process only one representative of the annotation duplicates. Moreover, scope aggregations keep track of all graph pattern matches that result in annotation duplicates and, thus, satisfies the goal OG3 - Enumerate Duplicates.

The impact analysis traverses scope aggregations from artifacts to annotations, which own these scope aggregations, to determine annotations that have to be maintained. Thus, the impact analysis contributes to the satisfaction of the goal OG2 - Time-Efficiency.

The maintenance modes of view modules support annotations with scope aggregations. The CREATE mode checks whether an annotation duplicate would be created, when the view modules find graph pattern matches. The UPDATE mode checks whether annotations own scope aggregations that must be deleted or extracted. The DELETE mode checks whether scope aggregations are obsolete and, if yes, deletes them. Thus, the maintenance modes satisfy the goal OG1 - Space-Efficiency. Furthermore, the maintenance modes contribute to the satisfaction of the goal OG2 - Time-Efficiency, because these modes employ scope aggregations to decrease search spaces of view modules. Moreover, the maintenance modes satisfy the goal OG3 - Enumerate Duplicates, because they maintain scope aggregations to keep track of all matches that result in annotation duplicates.

The maintenance phases execute view modules in certain maintenance modes. Thus, also the maintenance phases satisfy the goal OG1 - Space-Efficiency and OG3 - Enumerate Duplicates.

The maintenance phases propagate only one annotation for a set of annotation duplicates between view modules. Thus, successor modules have to process only one annotation for a set of annotation duplicates. Thus, the maintenance phases satisfy the goal OG2 - Time-Efficiency.

The original view graph maintenance (cf. Chapter 8) does *not* support the handling of annotation duplicates. Thus, the original approach does *not* satisfy the goals OG1 - Space-Efficiency and OG2 - Time-Efficiency. But, the original approach enumerates all graph pattern matches including the matches that result in annotation duplicates.

The naive duplicate handling employs annotations *without* scope aggregations and only marks one graph pattern match of a set of matches that result in annotation duplicates. Therefore, the naive duplicate handling satisfies the goal OG1 - Space-Efficiency. But, the naive duplicate handling does *not* mark all matches that result in annotation duplicates and, thus, does *not* satisfy the goal OG3 - Enumerate Duplicates. Furthermore, the naive duplicate handling creates one annotation for each set of annotation duplicates and only propagates this annotation between view modules. Thus, the approach satisfies the goal OG2 - Time-Efficiency.

The duplicate handling with aggregation employs annotations *with* scope aggregations and, thus, satisfies the goal OG1 - Space-Efficiency. Consequently, the approach exchanges only one annotation for each set of annotation duplicates between view modules. Thus, the duplicate handling with aggregation satisfies the goal OG2 - Time-Efficiency. Furthermore, the approach keeps track of all graph pattern matches, which result in annotation duplicates, in a manner that all matches can be retrieved instantly without additional search, later on. Algorithm E.9 describes how view graphs with scope aggregations can be transformed into view graphs without scope aggregations. Thus, the duplicate handling with aggregation satisfies the goal OG3 - Enumerate Duplicates.

9.6.2. Requirements

Table 9.2 maps the refined requirements of Section 9.3 to the optimized view graph maintenance. The requirement R1a - Nativeness describes that graph views must be graphs as well. The requirement R1b - Memory-Efficiency describes that graph views must store matches efficiently. The requirement R1c - Match-Properties describes that graph views must enable developers to enrich graph pattern matches with additional data values. The requirement R2b - Effectiveness describes that graph views must keep track of graph nodes with certain roles in matches. The requirement R2c - Reusability describes that graph views must be able to reuse the graph pattern matches that are stored by other graph views. The requirement R2d - Nesting describes that the framework must enable to maintain matches for patterns that employ nested graph conditions. The requirement R2e - Recursion describes that the framework must enable to maintain matches for patterns that employ recursive graph conditions. The requirement R4a - Monitoring describes that the framework must keep track of base graph changes. The requirement R4b - Time-Efficiency describes that the framework must maintain graph views efficiently and scalable. The requirement R4c - Propagation describes that changes of base graphs must be propagated to graph views and between graph views.

Scope aggregations are a special kind of graph nodes that preserve the graph structure of view graphs and, thus, satisfy the requirement R1a - Nativeness.

Scope aggregations keep track of graph pattern matches that result in annotation duplicates without creating new annotations for each match. Thus, scope aggregations are memory-efficient and satisfy the requirement R1b - Memory-Efficiency, because they avoid to store roles of annotation duplicates redundantly. The maintenance modes of view modules add and remove scope aggregations to and from annotations to keep track of graph pattern matches

Table 9.2.: Mapping the requirements to the optimized view graph maintenance

		Requirements									
		R1a - Nativeness	R1b - Memory-Efficiency	R1c - Match-Properties	R2b - Effectiveness	R2c - Reusability	R2d - Nesting	R2e - Recursion	R4a - Monitoring	R4b - Time-Efficiency	R4c - Propagation
View Graph	Scope Aggregation	●	●	●	●	◐	●	●	●	●	●
	Impact Analysis	○	○	○	○	◐	○	○	◐	◐	◐
Algorithmic Framework	Maintenance Modes	○	●	●	◐	◐	◐	◐	○	◐	◐
	Maintenance Phases	○	●	●	◐	●	●	●	○	◐	●
View Graph Maintenance	No Duplicate Handling	○	○	●	●	●	●	●	○	●	●
	Naive Duplicate Handling	○	●	●	●	◐	●	●	○	●	●
	Aggregation of Duplicates	○	●	●	●	●	●	●	○	●	●

covered: ●; partially covered: ◐; not covered: ○

that result in annotation duplicates. Thus, the maintenance modes satisfy requirement R1b - Memory-Efficiency. The maintenance phases employ the maintenance modes of view modules and, thus, satisfy the requirement R1b - Memory-Efficiency as well. The view maintenance without duplicate handling marks each graph pattern match that results in an annotation duplicate by means of a single annotation and, thus, does *not* support a memory-efficient handling of annotation duplicates. The view maintenance with naive duplicate handling marks only one match and discards all additional matches that result in annotation duplicates. Thus, this approach satisfies the requirement R1b - Memory-Efficiency, but loses the capability to enumerate all matches, which result in annotation duplicates, instantly. The view maintenance with aggregation of annotation duplicates employs scope aggregations that do not store roles of annotation duplicates redundantly. Thus, this approach satisfies R1b - Memory-Efficiency.

Scope aggregations preserve the capabilities of annotations to store additional attribute values. Thus, scope aggregations satisfy the requirement R1c - Match-Properties. The maintenance modes of view modules create annotation attributes and maintain their values. Furthermore, the maintenance modes consider attribute values to detect annotation duplicates. Thus, the maintenance modes satisfy the requirement R1c - Match-Properties. The maintenance phases execute view modules in certain maintenance modes and propagate annotations with changed attribute values between view modules. Thus, the maintenance phases satisfy the requirement R1c - Match-Properties. The view graph maintenance without duplicate handling creates annotations for each annotation duplicate and maintains their attribute values. Thus, this approach satisfies the requirement R1c - Match-Properties. The view graph maintenance with naive duplicate handling creates only one annotation for each set of annotation duplicates by marking only one match and discarding all additional matches that result in annotation duplicates. Thus, this approach satisfies the requirement R1c - Match-Properties, because it checks the equality of annotation attribute values, when creating and updating annotations. The view graph maintenance with aggregation of annotation duplicates keeps track of all matches that result in annotation duplicates and handles attribute changes of these annotation duplicates. This approach moves scope aggregations to other annotations or extracts new annotations, if the matches that result in annotation duplicates result in different attribute values. Thus, this approach satisfies the requirement R1c - Match-Properties.

Scope Aggregations mark graph nodes that have no special roles in graph pattern matches and preserve the capability of annotations to mark graph nodes that have certain roles in graph pattern matches. Thus, scope aggregations satisfy the requirement R2b - Effectiveness. The maintenance modes and maintenance phases still maintain roles of annotations with and without scope aggregations and, thus, contribute to the satisfaction of the requirement R2b - Effectiveness. All three approaches for the handling of annotation duplicates employ roles to mark graph nodes with certain roles in matches. Thus, these approaches satisfy the requirement R2b - Effectiveness.

Scope aggregations aggregate graph pattern matches and enable view modules to exchange these aggregated matches by means of one single annotation. Thus, scope aggregations contribute to the satisfaction of the requirement R2c - Reusability. The impact analysis uses roles and scope aggregations between annotations to determine dependent suspicious or obsolete annotations. Thus, the impact analysis contributes to the satisfaction of the requirement R2c - Reusability. The maintenance modes of view modules provide annotations with scope aggregations as output that are created, suspicious, or obsolete. This output enables the framework to forward the provided annotations to dependent view modules. Thus, the maintenance modes contribute to the satisfaction of the requirement R2c - Reusability. The maintenance phases of the view graph maintenance algorithms forward the annotations with scope aggregations between view modules. Thus, the maintenance phases satisfy the requirement R2c - Reusability. The view graph maintenance without duplicate handling exchanges all annotation duplicates between view modules. Thus, this approach satisfies the requirement R2c - Reusability. The view graph maintenance with naive duplicate handling exchanges only one annotation from a set of annotation duplicates. This approach partially satisfies the requirement R2c - Reusability, because the successor view modules cannot enumerate all graph pattern matches that result from annotation duplicates. The view graph maintenance with aggregation of annotation duplicates exchanges one single annotation that aggregates all annotation duplicates. This approach totally satisfies the requirement R2c - Reusability, because the successor view modules can enumerate all graph pattern matches that result from annotation duplicates.

Scope aggregations have no impact on the way how developers have to model view modules and view module dependency graphs. The mapping of nested graph conditions to view modules and view module dependency graphs work the same way as for annotations without scope aggregations. The view modules internally employ scope aggregations to aggregate graph pattern matches. Thus, scope aggregations satisfy the requirement R2d - Nesting. The maintenance modes and maintenance phases handle PACs and NACs for annotations with scope aggregations in the same way as for annotations without scope aggregations. Thus, the maintenance modes contribute to the satisfaction of the requirement R2d - Nesting and the maintenance phases satisfy the requirement R2d - Nesting. All maintenance procedures employ these maintenance modes and phases. Thus, they satisfy the requirement R2d - Nesting.

The same arguments also hold for the support of recursive graph conditions. The mapping of recursive graph conditions to view module dependency graphs works the same way as for annotations without scope aggregations. Thus, scope aggregations preserve the satisfaction of the requirement R2e - Recursion. The maintenance modes and phases handle recursive definitions for annotations with scope aggregations in the same way as for annotations without scope aggregations. Thus, the maintenance modes contribute to the satisfaction of the requirement R2e - Recursion and the maintenance phases satisfy the requirement R2e - Recursion. Thus, all maintenance procedures satisfy the requirement R2e - Recursion.

Similar to scopes, scope aggregations enable the framework to derive obsolete and suspicious annotations from changes of base graphs. Therefore, the scope aggregations satisfy the requirement R4a - Monitoring. The impact analysis uses created, deleted, and modified graph nodes to determine missing, obsolete, and suspicious annotations and scope aggregations. Thus, the impact analysis contributes to the satisfaction of the requirement R4a - Monitoring.

Scope aggregations enable the framework to look up annotations that are impacted by base graph changes. The impacted annotations describe portions of view graphs that must be considered, when maintaining view graphs. Therefore, scope aggregations enable the framework to prune search spaces and satisfy the requirement R4b - Time-Efficiency. The impact analysis determines which annotations are impacted by base graph changes and, thus, contributes to the satisfaction of the requirement R4b - Time-Efficiency. The maintenance modes and phases process these portions of view graphs. However, the concrete maintenance algorithms themselves are responsible to derive these portions of view graphs that must be investigated during view graph maintenance. Thus, the maintenance modes and phases contribute to the satisfaction of the requirement R4b - Time-Efficiency. All three incremental approaches for handling annotation duplicates prune searches spaces, when maintaining view graphs. The view maintenance without duplicate handling investigates each portion of the view graph that is defined by the roles of annotation duplicates. The view maintenance with naive duplicate handling investigates only the portions of the view graph that are defined by the roles of the annotations that represent sets of annotation duplicates. The view maintenance with aggregation of annotation duplicates investigates each portion of the view graph that is defined by the roles of the annotations, which own the scope aggregations.

Similar to scopes, scope aggregations between annotations are used to determine missing, obsolete, and missing annotations. Thus, the scope aggregations satisfy the requirement R4c - Propagation. The impact analysis determines the state of annotations with scope aggregations in their overall life cycle and triggers the view graph maintenance accordingly. Thus, the impact analysis contributes to the satisfaction of the requirement R4c - Propagation. The maintenance modes of view modules return annotations with scope aggregations that become suspicious, become obsolete, or were created. Therefore, the maintenance modes contribute to the satisfaction of the requirement R4c - Propagation. The maintenance phases forward annotations provided by view modules to dependent view modules. Therefore, the maintenance phases perform the actual change propagation between view modules. Thus, the maintenance phases satisfy the requirement R4c - Propagation.

All three approaches for handling annotation duplicates employ the maintenance modes and maintenance phases. The approach without duplicate handling employs the original scopes and, therefore, satisfies the requirement R4c - Propagation. The approach with naive duplicate handling employs the same scopes as the approach without duplicate handling and, therefore, satisfies the requirement R4c - Propagation. The approach with aggregation of annotation duplicates employs scope aggregations that are used by the maintenance modes and phases to look up dependent annotations that must be maintained due to base graph and view graph changes. Thus, the aggregation of duplicates satisfies the requirement R4c - Propagation.

10. Application Evaluation

This chapter evaluates the proposed modeling language. The evaluation focuses on the effectiveness of the modeling language and the capabilities to optimize the view graph maintenance. Section 10.1 describes the goals of the evaluation. Section 10.2 summarizes two case studies that are used to evaluate the proposed modeling language. Afterwards, Section 10.3 compares the proposed approach with state-of-the-art approaches from different evaluation perspectives. Section 10.4 summarizes this comparison.

10.1. Evaluation Goals

The goal of this evaluation is to a) determine the effectiveness of the proposed modeling language in comparison to other state-of-the-art approaches, which can be used for incremental view graph maintenance, as well as b) determine capabilities for the optimization of the view graph maintenance by means of the proposed modeling language. For that purpose, this thesis derives different evaluation perspectives from the elicited requirements of Section 4.2.

Expressiveness
The expressiveness perspective discusses how the modeling language enables developers to express conjunctions, disjunctions, negations, recursions, and attribute constraints. This perspective also covers the evaluation of the required modeling effort for the definition of views. Furthermore, the perspective investigates how views can hide parts of graph pattern matches as well as enrich graph pattern matches.

Optimization
The optimization perspective discusses how the approaches for incremental view graph maintenance can steer the trade-off between memory consumption and execution time.

Generality
The generality perspective investigates whether developers can employ query languages and graph data models of their choice for view graph maintenance.

Retrieval
The retrieval perspective discusses how developers can post-processed maintained graph pattern matches effectively. This issue includes the retrieval of graph nodes with certain roles in matches, the retrieval of all matches including matches that result in annotation duplicates, and the retrieval of reused matches that are maintained by other view graphs.

10.2. Case Studies

This section describes the case studies that are used by this thesis to evaluate the modeling capabilities of the proposed modeling language in comparison to state-of-the-art approaches. For this purpose, this thesis implements the proposed modeling language using the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF). This thesis provides an editor that enables to model view modules, view module dependencies, and view graph transformations. Section 10.2.1 summarizes the case study about the recovery of design patterns and anti-patterns in ASGs of source code. Section 10.2.2 summarizes the case study about the tracing of innovation processes to keep track of design rationales.

10.2.1. Recovery Design Pattern and Anti-Patterns (CS1)

Reverse engineering software systems is a major software engineering discipline. Recovering employed design patterns and anti-patterns is one activity during reserve engineering.

The world wide web provides plenty of version control repositories with source code that is published under open source licenses. These repositories store source codes of different sizes and consist of change histories that can be used to imitate changing abstract syntax graphs (ASGs) based on real source code changes. For this case study, this thesis derives ASGs from Java source code using JaMoPP [53]. JaMoPP provides a metamodel that describes the language concepts of the Java programming language and is an extended version of the metamodel that is used by the running example (cf. Figure 3.4).

Gamma et al. [35] and Fowler et al. [33] describe design patterns and design anti-patterns, respectively. For this case study, this thesis translates the design patterns and anti-patterns into graph patterns, which conform to the JaMoPP metamodel, and embeds these patterns into view modules to maintain matches for these patterns. This case study employs 49 view modules for design patterns and anti-patterns. The Appendix F lists the employed view modules and provides short descriptions of the embodied patterns. The view modules implement low-level patterns such as the Generalization pattern (cf. Figure 3.5(a)) and high-level patterns such as the Composite pattern (cf. Figure 3.8(a)). The view modules create and maintain annotations, which mark graph nodes of the ASGs that have a certain role in design patterns and anti-patterns. For example, the Generalization view module marks the graph nodes that represent the super- and subclass of the generalizations. The running example of this thesis already explained some of these view modules from the perspective of view definition (cf. Chapter 6) and view graph transformation (cf. Chapter 7).

10.2.2. Tracing Innovation Projects (CS2)

This thesis selects the tracing of innovation projects to keep track of design rationales as second case study, because it is a non-technical example and, therefore, complements the first technical case study. This case study deals with innovation projects that employ the Design Thinking innovation methodology. The Design Thinking innovation methodology provides design phases, design activities, and design techniques that enable to solve wicked design challenges with innovative solutions. When design teams employ the Design Thinking methodology they undergo a learning and decision process. The design team documents the learnings and design decisions in design artifacts that constitute a design documentation. This case study aims for recovering traceability links in this design documentation to make the documentation traceable during and after the innovation project to enable engineers to implement the proposed design solution in a feasible, desired, and viable manner.

This case study derives the graph data from design documentation that is stored in file system structures. Therefore, the graph data consists of graph nodes that represent design artifacts and graph edges that represent containment relationships between these design artifacts. Furthermore, each graph node consists of an attribute that links the physical design artifacts to enable view modules to load and process these physical artifacts. The view modules infer traceability links between design artifacts and store these traceability links in terms of annotations in view graphs. The evolution of the design documentation is derived from creation dates of design artifacts.

For example, view modules extract keywords and creation dates from metadata of design artifacts. Furthermore, this case study employs view modules that process extracted keywords to conclude design phases, design activities, and design techniques as well as creation dates to establish the creation order of the design phases, design activities, and design techniques. With this recovered order, the employed innovation process can be recovered and analyzed. Then, transitions between design phases, activities, and techniques can be analyzed. Beyhl et al. report on the traceability of innovation processes in several publications [11, 12, 13, 14, 15]. Appendix G describes the view modules of this case study.

10.3. Comparing Approaches

Section 1.3.2 describes EMF-IncQuery as software tool for MDE that implements Rete networks and adapts the Rete matching algorithm for EMF models as special kind of graph. EMF-IncQuery is the only approach that transfers Rete networks and the Rete matching algorithm to graphs. EMF-IncQuery provides a textual syntax to enable developers to model patterns. These patterns do *not* describe the structure of Rete networks. lnstead, EMF-lncQuery generates a joint Rete network for all patterns by means of a heuristic [9]. The resulting Rete network can employ network node sharing [4] to reuse common network parts between patterns to reduce time and space complexity [9].

The textual syntax of EMF-IncQuery for graph patterns consists of a pattern header and a pattern body. The header describes the kinds of graph nodes that are returned by means of tuples, when EMF-IncQuery finds a match. The body describes the actual pattern for which EMF-IncQuery maintains matches. Bodies consist of type checks and edge checks. Type checks test whether a graph node has a certain type. Edge checks test whether two graph nodes are connected by a graph edge of a certain type. Furthermore, the textual syntax enables to express simple attribute constraints for primitive data types.

Moreover, the textual syntax provides the find keyword to refer to matches of other patterns and test for the existence of certain matches. The textual syntax also provides the neg keyword that is employed in combination with the find keyword to test for the non-existence of certain matches. Furthermore, the textual syntax provides an anonymous variable _ that matches any graph node. Additionally, the textual syntax provides an or keyword that enables to combine multiple pattern bodies in a disjunctive manner.

The following sections compare the modeling approach of this thesis with EMF-IncQuery concerning their expressiveness, capabilities of performance optimizations, generality, and retrieval of matches. This evaluation derives the modeling and functional capabilities of EMF-IncQuery from several publications [7, 9] and hands-on experiences.

10.3.1. Expressiveness

In general, the proposed modeling language and EMF-IncQuery have the same expressiveness, because both approaches support conjunctions, disjunctions, negations, and recursion. Furthermore, both approaches support arbitrarily complex attribute constraints, which must be free of side-effects. However, both approaches differ in the manner how they enable developers to model discrimination networks and logical operations for combining matches of certain patterns. The proposed approach enables developers to model the structure of generalized discrimination networks, while EMF-IncQuery enables developers to model patterns. EMF-IncQuery itself derives Rete networks (cf. Section 1.3.1) from these patterns. EMF-IncQuery does *not* support Gator networks (cf. Section 1.3.1).

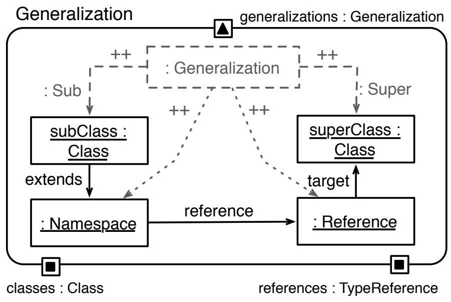
Section 6.4 and Section 7.2 describe how developers can use the modeling language and view graph transformations to map graph conditions to view modules and view module dependency graphs. Bergmann et al. [9] describe how developers can implement graph conditions with EMF-IncQuery [7]. The following sections compare the modeling of atomic conditions, conjunctions, disjunctions, negations, recursions, and attribute constraints.

Atomic Graph Conditions

The proposed modeling approach maps atomic graph conditions to view modules that only consist of artifact input connectors (cf. Section 6.4.1). EMF-IncQuery maps atomic graph conditions to patterns that do *not* employ the find keyword to refer to other matches.

Figure 10.1(a) shows the Generalization view module that implements the Generalization pattern. The module consists of two artifact connectors, which describe that the module requires Class and TypeReference artifacts to search for matches of the Generalization pattern.

Figure 10.1(b) shows the Generalization pattern in EMF-IncQuery. The pattern header describes that the pattern requires Class artifacts. The header abstracts from TypeReference artifacts that are also required to find matches of the Generalization pattern.



(a) Proposed Approach

```
pattern Generalization(subClass:Class, superClass:Class) {
  Class.^extends(subClass, namespace);
  Namespace.reference(namespace, reference);
  Reference.target(reference, superClass);

  Namespace(namespace);
  Reference(reference);
}
```

(b) EMF-IncQuery

Figure 10.1.: Modeling atomic graph conditions

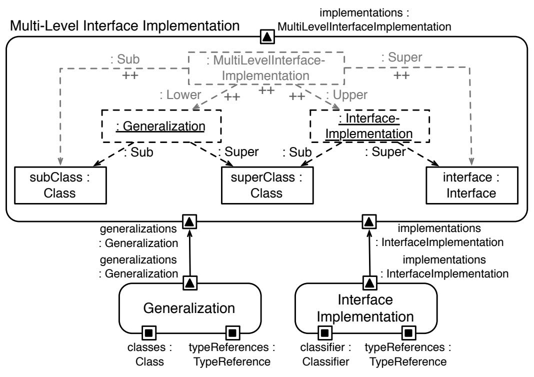
The view module in Figure 10.1(a) provides a complete interface of the encapsulated transformation, because the connectors describe all kinds of required artifacts and created annotations. In contrast, the pattern header in Figure 10.1(b) does not describe an interface of the encapsulated pattern. Búr et al. [20] describe pattern parameters as *“a subset of nodes and attributes interfacing the model elements interesting from the perspective of the pattern user”*. It is the choice of the developer to add the TypeReference artifacts to the pattern header. Thus, the pattern headers in EMF-IncQuery reflect or do not reflect all kinds of required artefacts. Therefore, pattern headers maybe do not provide a complete interface for the pattern bodies. Therefore, the pattern itself is required to determine whether base graph changes impact the stored matches. In contrast, the proposed approach of this thesis encapsulates patterns behind interfaces of view modules. Thus, the proposed modeling language makes the maintenance algorithms independent from patterns that are encapsulated by view modules.

Conjunctions

The proposed modeling language maps conjunctions to view modules with annotation input connectors that have only one incoming view module dependency (cf. Section 6.4.2). Then, the embodied view graph transformation employs annotation pattern nodes that are connected with each other via other pattern nodes and edges (cf. Section 7.2.2). EMF-IncQuery maps conjunctions to patterns that make use of other graph pattern matches by means of the find keyword. The pattern implements a conjunction, when it employs at least two find statements.

Figure 10.2(a) shows the Multi-Level Interface Implementation view module that embeds a view graph transformation, which describes that the superclass of a generalization and the subclass of an interface implementation must be the same class, when the generalization and interface implementation constitute a multi-level interface implementation. For that purpose, the view module consists of two annotation input connectors that receive annotations from the Generalization and InterfaceImplementation view module.

Figure 10.2(b) shows the equivalent pattern in EMF-IncQuery. The pattern searches for matches of the Generalization pattern and the Interface Implementation pattern that have the superclass of the generalization and the subclass of the interface implementation in common. For that purpose, the pattern employs two find statements to refer to matches of the Generalization and Interface Implementation pattern.


implementations :
MultiLevelInterfaceImplementation
Multi-Level Interface Implementation
: Sub
++
: MultiLevelInterface-Implementation
: Super
++
: Lower
++
++
++
: Upper
: Generalization
: Interface-Implementation
: Sub
: Super
: Sub
: Super
subClass : Class
superClass : Class
interface : Interface
generalizations : Generalization
generalizations : Generalization
implementations : InterfaceImplementation
implementations : InterfaceImplementation
Generalization
Interface Implementation
classes : Class
typeReferences : TypeReference
classifier : Classifier
typeReferences : TypeReference

(a) Proposed Approach

```
pattern MultiLevelInterfaceImplementation(class:Class,
interface:Interface) {
  find Generalization(class, superClass);
  find IntefaceImplementation(superClass, interface);
}

pattern InterfaceImplementation(class:Class,
interface:Interface) {
  Class.implements(class, namespace);
  Namespace.references(namespace, reference);
  Classifier.target(reference,interface);

  Namespace(namespace);
  Reference(reference);
}
```

(b) EMF-IncQuery

Figure 10.2.: Modeling conjunctions of graph conditions

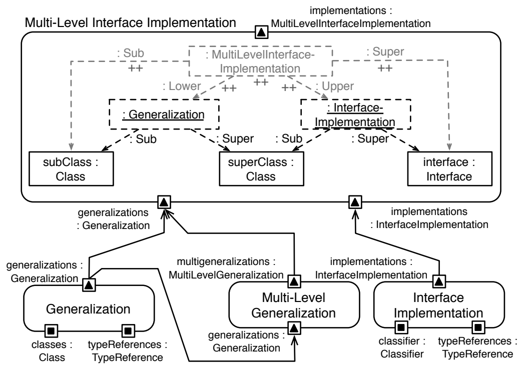
Both modeling approaches handle conjunctions in a similar way. However, pattern headers in EMF-IncQuery do *not* describe which kinds of matches are reused within the pattern. Therefore, patterns in EMF-IncQuery do *not* provide a complete interface for patterns as it is the case for view modules in the proposed modeling approach. That means, in EMF-IncQuery pattern headers do not describe dependencies between patterns. Instead, EMF-IncQuery calculates decompositions in network nodes and the dependencies between these nodes to optimize the structure of the generated Rete network [9]. In the proposed approach, these dependencies are explicitly modeled by means of view module connectors / dependencies and it is the responsibility of the developer to optimize the network structure.

Disjunctions

The proposed modeling language maps disjunctions to view modules with annotation input connectors that have at least two incoming view module dependencies (cf. Section 6.4.3). Then, the encapsulated pattern exploits the polymorphism of the received annotations to match annotations that are part of the disjunction. In contrast, EMF-IncQuery maps disjunctions to multiple pattern bodies using the or keyword. The developers must describe each operand of the disjunction by an additional pattern body.

Figure 10.3(a) shows the view module that implements the Multi-Level Interface Implementation pattern. This view module receives annotations that mark matches for the Generalization, MultiLevelGeneralization, and InterfaceImplementation pattern. The generalization annotation input connector of the view module has two incoming view module dependencies that originate from the Generalization view module and Multi-Level Generalization view module. Furthermore, the view graph transformation of the Multi-Level Interface Implementation view module employs an annotation pattern node with Generalization annotation type (super type of Multi-Level Generalization annotation type) to describe that the superclass of a single-level or multi-level generalization can implement an interface. Therefore, the view module implements a disjunction of the Generalization and Multi-Level Generalization pattern.

Figure 10.3(b) shows the implementation of the Multi-Level Interface Implementation pattern in EMF-IncQuery. The pattern consists of two pattern bodies that are part of the disjunction. The first pattern body describes a pattern that employs matches of the Generalization pattern and Interface Implementation pattern to search for matches of the Multi-Level Interface Implementation pattern. The second pattern body describes a pattern that employs matches of the Multi-Level Generalization pattern and Interface Implementation pattern to search for matches of the Multi-Level Interface Implementation pattern.


Multi-Level Interface Implementation
implementations : MultiLevelInterfaceImplementation
: Sub
++
: MultiLevelInterface-Implementation
: Super
++
: Lower
++
++
++
: Upper
: Generalization
: Interface-Implementation
: Sub
: Super
: Sub
: Super
subClass : Class
superClass : Class
interface : Interface
generalizations : Generalization
implementations : InterfaceImplementation
generalizations : Generalization
multigeneralizations : MultiLevelGeneralization
implementations : InterfaceImplementation
Generalization
Multi-Level Generalization
Interface Implementation
classes : Class
typeReferences : TypeReference
generalizations : Generalization
classifier : Classifier
typeReferences : TypeReference

(a) Proposed Approach

```
pattern MultiLevelInterfaceImplementation(class:Class,
interface:Interface) {
  find Generalization(class, superClass);
  find IntefaceImplementation(superClass, interface);
} or {
  find MultiLevelGeneralization(class, superClass);
  find InterfaceImplementation(superClass, interface);
}
```

(b) EMF-IncQuery

Figure 10.3.: Modeling disjunctions of graph conditions

In EMF-IncQuery, the developers must model each possible pattern of a disjunction explicitly. The resulting combinatorial complexity results in a higher modeling effort in comparison to the proposed approach. The proposed approach enables developers to describe the disjunction effectively, because the approach exploits the polymorphism of annotations to support disjunctions. Then, the graph pattern has to employ the annotation supertype of annotations that participate in the disjunction.

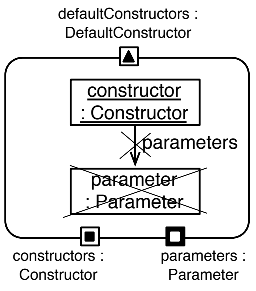
Internally, EMF-IncQuery generates Rete subnetworks for each operand of the disjunction [9], which may share network nodes [4]. These subnetworks have a common production node at the end, which performs a true union operation, to store matches that are computed by these subnetworks [9].

Simple Negations

The proposed modeling language maps simple negation to view modules with negative artifact input connectors (cf. Section 6.4.4). Then, the encapsulated view graph transformation employs negated artifact pattern nodes and relation pattern edges to describe that certain graph nodes and edges must not exist (cf. Section 7.2.4). In contrast, EMF-IncQuery maps simple negation to negated pattern calls. First, a graph pattern must be employed that searches for the existence of graph nodes and edges, which must not exist in the overall pattern. Then, a second pattern checks whether no match of the first pattern exists.

Figure 10.4(a) shows the Default Constructor view module that consists of a negative parameters artifact connector. The view graph transformation implements the simple negation by means of a negated parameters pattern edge and parameter pattern node.

Figure 10.4(b) shows the DefaultConstructor pattern in EMF-IncQuery. The DefaultConstructor pattern employs the neg and find keyword to describe that the constructor must not participate in a match of the ConstructorWithParameters pattern.


defaultConstructors : DefaultConstructor
constructor : Constructor
parameters
~~parameter : Parameter~~
constructors : Constructor
parameters : Parameter

(a) Proposed Approach

```
pattern DefaultConstructor(constructor:Constructor) {
  neg find ConstructorWithParameters(constructor);
}

pattern ConstructorWithParameters(constructor:Constructor) {
  Constructor.parameters(constructor,_);
}
```

(b) EMF-IncQuery

Figure 10.4.: Modeling simple negation

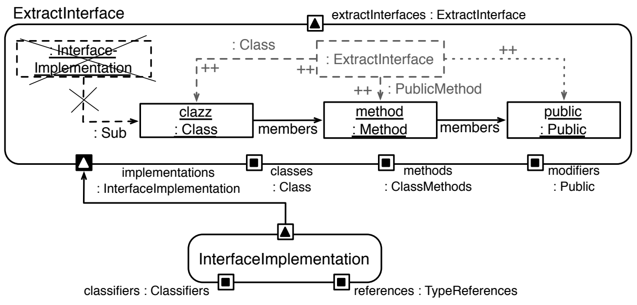
EMF-IncQuery always requires two patterns to implement simple negations. In contrast, the proposed modeling approach maps simple negations more effectively than EMF-IncQuery, because the proposed approach is able to handle simple negations within view modules and does *not* require two view modules. Note that the proposed approach can map simple negations also to two view modules, if it is desired by the developers. Furthermore, the mapping to two patterns in EMF-IncQuery results in more intermediate graph pattern matches, which must be stored, than in the proposed approach. Thus, the proposed approach enables a more memory-efficient mapping of simple negations to view modules.

Complex Negations

The proposed modeling language maps complex negations to two view modules. The first view module searches for graph pattern matches that dissatisfy the negated part of the overall graph pattern. The second view module implements the complete pattern and checks for the non-existence of the match that dissatisfies the negated part of the pattern. For that purpose, the second view module employs negative annotation input connectors for the annotations that must not exist. Furthermore, the encapsulated view graph transformation employs negated annotation pattern nodes and role pattern edges. In contrast, EMF-IncQuery maps complex negations to patterns in the same way as simple negations. Thus, EMF-IncQuery maps complex negations in the same way to patterns as the proposed modeling approach.

Figure 10.5(a) shows the Extract Interface view module and Interface Implementation view module. The Extract Interface view module consists of a negative annotation input connector with InterfaceImplementation annotation type to describe that the Extract Interface view module uses annotations, which are provided by the Interface Implementation view module, in negative sense. Furthermore, the view graph transformation of the Extract Interface view module employs a negated Interface Implementation annotation.

Figure 10.5(b) shows the Extract Interface pattern and the Interface Implementation pattern in EMF-IncQuery. The neg and find keyword in the Extract Interface pattern describe that a class must *not* participate in a match of the Interface Implementation pattern.



(a) Proposed Approach

```
pattern ExtractInterface(class:Class) {
  neg find InterfaceImplementation(class, _);

  Class.members(class, method);
  Method.modifiers(method, public);

  Method(method);
  Public(public);
}
```

(b) EMF-IncQuery

Figure 10.5.: Mapping complex negation

The proposed modeling language enables developers to map complex negations in the same manner to view modules as in EMF-IncQuery. In EMF-IncQuery, pattern headers describe which nodes of graph pattern matches are of interest to users. In EMF-IncQuery, pattern bodies explicitly call previously defined patterns to nest patterns. Thus, one must investigate pattern bodies to understand which sub-patterns are reused in negated manner by means of the neg keyword. Therefore, these pattern headers are *not* sufficient to determine whether a found graph pattern match may dissatisfy matches of a certain other graph pattern.

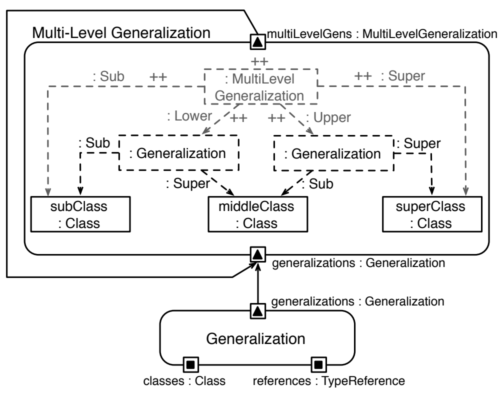
Recursion

The proposed modeling language enables developers to model recursive graph conditions (cf. Definition 15) with the help of cyclic view modules dependencies. In the proposed approach, recursion cycles can consist of multiple view modules and recursion cycles can embed other recursion cycles. In EMF-IncQuery, patterns implement recursion by calling other patterns using the find keyword. When the patterns implement cyclic calls of other patterns, then they implement a recursion. In EMF-IncQuery, recursion cycles can consist of multiple patterns and recursion cycles can embed other recursion cycles.

Figure 10.6(a) shows the Generalization view module and Multi-Level Generalization view module. The Generalization view module describes the recursion start. The Multi-Level Generalization view module describes the recursion step, because a view module dependency connects the output connector of the view module with its input connector. The encapsulated view graph transformation employs an annotation pattern node with Generalization type that can match also annotations with Generalization and Multi-Level Generalization type.

Figure 10.6(b) shows the Multi-Level Generalization pattern in EMF-IncQuery. Each pattern body calls the Generalization and Multi-Level Generalization patterns. The pattern bodies cover all possible combinations of the Generalization and Multi-Level Generalization patterns.

In EMF-IncQuery, recursion is implemented within patterns by calling other pattern. These recursive calls result in cyclic dependencies between single Rete networks, which are generated by EMF-IncQuery for each pattern [9]. In contrast, the proposed approach handles recursion cycles outside of view modules by means of explicit view module dependencies. Therefore, the framework does not have to be aware of the encapsulated patterns to maintain matches that result from recursive graph conditions. Furthermore, the patterns in EMF-IncQuery must


Multi-Level Generalization
multiLevelGens : MultiLevelGeneralization
: Sub
++
: MultiLevel Generalization
++
++ : Super
: Lower
++
++
: Upper
: Sub
: Generalization
: Generalization
: Super
: Super
: Sub
subClass : Class
middleClass : Class
superClass : Class
generalizations : Generalization
generalizations : Generalization
Generalization
classes : Class
references : TypeReference

(a) Proposed Approach

```
pattern MultiLevelGeneralization(sub:Class, super:Class) {
  find Generalization(sub, middle);
  find Generalization(middle, super);
  Class(middle);
} or {
  find Generalization(sub, middle);
  find MultiLevelGeneralization(middle, super);
  Class(middle);
} or {
  find MultiLevelGeneralization(sub, middle);
  find Generalization(middle, super);
  Class(middle);
} or {
  find MultiLevelGeneralization(sub, middle);
  find MultiLevelGeneralization(middle, super);
  Class(middle);
}
```

(b) EMF-IncQuery

Figure 10.6.: Mapping recursion

provide disjunctive pattern bodies, which a) implement recursion steps based on matches for recursion starts and b) implement recursion steps based on matches for other recursion steps.

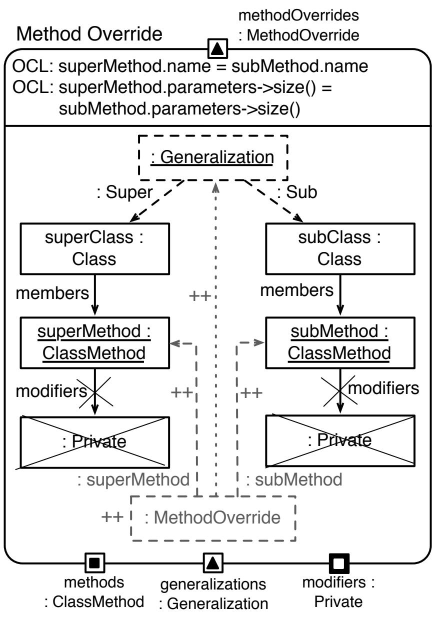
Attribute Constraints

In the proposed modeling language, the view graph transformations enable to specify attribute constraints that must hold for graph nodes of found matches. For example, the proposed modeling language supports OCL. In EMF-IncQuery, *"user-provided arbitrary Java code can be applied to model elements to check the validity of a pattern"* [9]. For the proposed approach, OCL has been extended by custom operations, which can call arbitrary Java code. Furthermore, the proposed approach enables to employ arbitrary expression languages (see Appendix B). In both approaches, the attribute constraints must be free of side-effects. Due to different expression languages, attribute constraints cannot be implemented analogously in both approaches. For example, the Method Override pattern (cf. Figure 3.7) checks whether a) the name of a method in a class and another method in a subclass of this class are equal and b) the numbers of parameters are equal for both methods. For the sake of simplicity, the example omits the check of the parameter order and type.

Figure 10.7(a) shows the Method Override view module. The Method Override view module uses Generalization annotations to look up pairs of private methods in the superclass and the subclass that have the same method names and number of method parameters. The view module employs OCL expressions to check the equality of the method names and the number of method parameters.

Analogously, Figure 10.7(b) shows the Method Override pattern in EMF-IncQuery. The pattern employs the check operation of EMF-IncQuery to check the equality of the method names. Furthermore, the pattern employs the count keyword of EMF-IncQuery to check for equal numbers of method parameters. For that purpose, the pattern makes use of an additional Param pattern that looks up all parameters that belong to a method.

Note that both approaches must ensure that the attribute constraints operate within the scope of the graph pattern matches. That means, the attribute constraints must not refer to graph nodes and edges that are not part of the graph pattern matches. Otherwise, graph changes are not related to view modules in the proposed approach and patterns in EMF-IncQuery during the view graph maintenance.



(a) Proposed Approach

```
pattern MethodOverride(subMethod:Method, superMethod:Method) {
  find Generalization(subClassifier, superClassifier);
  Class.members(subClassifier, subMethod);
  Class.members(superClassifier, superMethod);

  neg find PrivateMethod(subMethod, _);
  neg find PrivateMethod(superMethod, _);

  Method.name(subMethod, subMethodName);
  EString(subMethodName);
  Method.name(superMethod, superMethodName);
  EString(superMethodName);

  check(subMethodName.equals(superMethodName));
  count find Param(subMethod)==count find Param(superMethod);
}

pattern PrivateMethod(method:ClassMethod, modifier:Private) {
  Method.annotationsAndModifiers(method, modifier);
}

pattern Param(method:Method) {
  Method.parameters(method, _);
}
```

(b) EMF-IncQuery

Figure 10.7.: Mapping attribute constraints

Discussion

The different network structures of Rete networks and Gator networks (cf. Section 1.3.1) have several implications, when modeling such network structures. Rete networks consist of network nodes with at most two inputs. Therefore, developers of Rete networks must decompose graph patterns into multiple small-sized network nodes with two inputs. Thus, when patterns become large, it is a cumbersome task to model the network nodes of Rete networks manually. Therefore, EMF-IncQuery enables developers to define patterns that are decomposed into Rete networks by a heuristic approach [9]. EMF-IncQuery graph patterns describe the nesting of patterns by means of the find keyword. According to [9], developers can guide the heuristic by splitting patterns into smaller patterns. The structure of the Rete network in EMF-IncQuery is a result of a complex optimisation step. Gator networks consist of network nodes that can have more than two inputs. Therefore, Gator networks enable developers to choose a larger granularity for network nodes than Rete networks, when decomposing patterns into discrimination networks. Thus, developers of Gator networks can model network nodes with more than two inputs to reduce the number of network nodes that have to be modeled. Thus, Gator networks are more suited for modeling the network structure than Rete networks. For this reason, the proposed approach supports Gator networks including Rete networks. EMF-IncQuery does not support Gator networks.

Furthermore, the proposed approach exploits the polymorphism of annotations to model disjunctions and recursions. This polymorphism reduces the modeling effort. In contrast, in EMF-IncQuery the modeling of disjunctions and recursions suffers a combinatorial complexity, which results in multiple pattern bodies that must cover all cases of disjunctions and recursions.

EMF-IncQuery employs pattern headers to describe which graph nodes of matches are contained by tuples that represent matches. For example, the header of the Generalization pattern in Figure 10.1(b) describes that the tuples contain the superclass and the subclass of the generalization and hide the graph nodes with Namespace and Reference type. The proposed approach employs roles and scopes to mark graph nodes of matches. The view graph transformations, which are encapsulated by view modules, describe by means of role and scope pattern edges which graph nodes are marked by roles and scopes, respectively. Roles mark the graph nodes that are *not* filtered out from matches and, therefore, are retrievable. Scopes mark the graph nodes that are filtered out from matches, but keep track of these graph nodes, because they may dissatisfy the pattern, when they change. In summary, EMF-IncQuery and the proposed approach enable to hide graph nodes of matches, but use different concepts to describe which kinds of graph nodes are hidden.

One purpose of views is to enrich the knowledge that is stored by graphs with additional knowledge (cf. Chapter 1). EMF-IncQuery maintains matches and can be used to derive features [81] (i. e., attribute values or references that are calculated from other graph elements) and custom views [27] (i. e., graphs that abstract from details of other graphs to provide a specific point of view). The proposed approach employs attributed graph nodes to mark graph pattern matches. The attributes of these graph nodes enable to store additional data values that are initialized by view modules during the CREATE maintenance phase and are maintained by view modules during the UPDATE maintenance phase. The developers employ arbitrary expression languages to derive the attribute values from the graph nodes of matches.

10.3.2. Optimization

EMF-IncQuery generates Rete networks from patterns based on a heuristic that performs optimizations of the network structure [9]. Developers can select for each pattern, if local-search or incremental strategies shall be applied [6] by means of the search keyword [28]. This enables a trade-off between memory consumption and execution time, but loses incrementality, when local-search is employed. The user guide [28] and Bergmann et al. [6] provide technical details and a discussion, when a local search, an incremental strategy, or a combination of both strategies is beneficial.

The proposed modeling language provides developers the full freedom of generalized discrimination networks, because they allow network nodes with more than two inputs. According to Hanson et al. [48], network nodes with more than two inputs require less memory than equivalent Rete network substructures. But, network nodes with more than two inputs may have a higher execution time, because less intermediate matches are cached that can speed up the execution, when the state of the discrimination network is updated. Since the presented approach enables developers to model generalized discrimination networks, the presented approach enables developers to steer the trade-off between memory consumption and execution time by choosing an appropriate granularity level for network nodes.

The proposed approach enables to employ different graph pattern matching algorithms and languages to implement view modules. For example, the research prototype employs story diagrams as graph pattern matching language that are interpreted by a story diagram interpreter to find graph pattern matches. Also graph patterns specified by means of other graph pattern languages can be easily embedded into view modules. Therefore, the developer can choose the most beneficial graph pattern matching approach. EMF-IncQuery enables to choose between an incremental and local-search based pattern matching strategy [20]. The approach reuses *"the existing pattern language and query development environment of EMF-IncQuery [. . .] to select the most appropriate strategy separately for each pattern without any modifications to the definitions of existing patterns"* [20]. Bergmann et al. [6] propose a combination of local-search based and incremental strategies for optimization purposes. Furthermore, EMF-IncQuery employs an effective index structure [9] to implement network nodes in the Rete network. This index structure can be more efficient than pattern matching, but is limited to network nodes with at most two inputs. In summary, the proposed approach can fall back to the structure of Rete networks, but can also employ a more general network structure. In the general case, the more general network structure will be more memory-efficient than an equivalent Rete network, because the general network structure employs less network nodes than the Rete network and, thus, has to store less intermediate results. Then, the general network structure has to maintain also less intermediate results and the maintenance *can* be faster than for an equivalent Rete networks. The maintenance performance also depends on the graph pattern matching performance of network nodes with more than two inputs.

EMF-IncQuery employs an immediate maintenance, when EMF-models change. That means, EMF-IncQuery propagates each single change immediately through the network. In contrast, the proposed approach supports a deferred maintenance. That means, the approach collects changes of base graphs, filters out modification events that cancel each other, and updates the discrimination network concerning these changes later, e. g., when an end-user states a query. Thus, the proposed approach uses the net modification events that can be less than all captured modification events. Therefore, the proposed approach can propagate less changes through the network, which can speed up the maintenance.

The proposed approach employs an explicit UPDATE maintenance phase to maintain matches that are impacted by modified graph elements. In contrast, Rete-based approaches [32] map the modification of graph elements to the inefficient deletion and re-creation of matches.

10.3.3. Generality

This section discusses the support of different graph query languages and graph data models.

Graph Query Languages

EMF-IncQuery provides no concepts to abstract from employed graph patterns. Logical operations such as disjunctions, conjunctions and negations are handled within the pattern bodies by calling other patterns. The pattern headers rather describe the kinds of tuples that are stored for each found match than encapsulating the employed pattern by providing an interface for the pattern. EMF-IncQuery derives Rete networks for these pattern. Thus, the Rete network construction is coupled to the query language of EMF-IncQuery. But, EMF-IncQuery can be silently used to implement other query or validation tools using its support for derived features [81].

The proposed approach provides view modules as abstraction from employed query languages. View modules hide employed query languages, because they provide interfaces for the encapsulated patterns. These interfaces describe which kinds of artifacts and annotations must be provided to view modules. The encapsulated view graph transformation is responsible to process the provided artifacts and annotations. Thus, the proposed approach is decoupled from employed patterns and, therefore, is also independent from employed query languages. Note that this thesis provides a default view graph transformation language to describe the proposed concepts. Developers can also employ other graph transformation languages.

Graph Data Models

EMF-IncQuery provides incremental graph pattern matching for EMF models. However, EMF-IncQuery has been also adapted to other graph-like data. For example, IncQuery-D employs concepts of EMF-IncQuery for incremental graph search in the cloud [55]. IncQuery-D employs a storage and Rete layer, which can be used independently of graph-oriented data representation formats. Furthermore, EMF-IncQuery has been integrated into the Meta Programming System (MPS) [89].

The proposed approach employs view modules to abstract from graph queries, query languages, and employed graph data models. Especially, the proposed approach works for graphs in general and is *not* limited to special kinds of graphs such as EMF models.

Angles [2] provides an overview of graph data models. Angles lists simple graphs, hypergraphs[1], and nested graphs[2] as possible graph data models. Additionally, these graph data models employ labels and / or attributes for graph nodes and graph edges and either consist of directed or undirected edges. This thesis employs typed attributed graphs with directed graph edges to describe the concepts of this thesis. The proposed marking of graph nodes, which satisfy a pattern, is independent from the employed graph data model, because all kinds of graphs employ graph nodes and view modules are responsible to mark these graph nodes. Thus, view modules handle the characteristics of employed graph data models and hide these characteristics from the framework. But, these graph data models must employ at least typed graph nodes, because the approach uses the types of graph nodes to describe the interfaces of view modules. Furthermore, these types are also used to prune the search spaces of view modules during the incremental view graph maintenance.

10.3.4. Retrieval

The following sections compare EMF-IncQuery and the proposed approach concerning the retrieval of graph pattern matches. This comparison includes the discussion how users retrieve graph nodes with certain roles in matches, retrieve matches that result in duplicates of tuples / annotations, and retrieve matches that are reused to find other matches.

Retrieval of Graph Nodes

In EMF-IncQuery, the pattern headers describe the graph nodes that can be retrieved from the incremental graph pattern matcher. In general, these pattern headers describe tuples and associate a name with each position in these tuples. When retrieving matches in terms of tuples, these names are used to refer to positions in the tuples. Thus, the roles of graph nodes are encoded by the names of the tuple positions. For example, Figure 10.1(b) shows that EMF-IncQuery stores tuples for matches of the Generalization pattern that contain the superclass and subclass of the found generalizations. In EMF-IncQuery, the other graph nodes of the match are not part of the retrieved tuples. Therefore, graph nodes without certain roles in matches cannot be retrieved in EMF-IncQuery. Thus, the full match cannot be restored.

In extensions of EMF-IncQuery [27], annotations can be added to patterns. These pattern annotations describe the creation of view models explicitly, when graph pattern matches are found. These view models are instances of metamodels that serve as reference about which information can be retrieved from view models.

The proposed approach employs graph edges to mark graph nodes with certain roles in matches. These graph edges consist of types that describe the roles of the graph nodes that are the target of these graph edges. Therefore, these types are used to retrieve graph nodes with certain roles in matches. Furthermore, annotations own graph edges that do not consist of a type and are used to mark graph nodes without roles in matches. In contrast to

[1]Hypergraphs employ graph edges that connect a set of graph nodes. That means, graph edges of hypergraphs are n-ary.

[2]Nested graphs employ graph nodes that are themselves graphs.

EMF-IncQuery, these graph edges are used to retrieve graph nodes without certain roles in matches. Thus, the proposed approach can restore the full graph pattern match.

Furthermore, the proposed approach defines explicitly in the view graph schema, which roles of graph nodes can be accessed in marked matches. In contrast, EMF-IncQuery defines roles of graph nodes rather ad hoc in the pattern header. Thus, the end-users and developers must know the implementation of patterns to know which roles of graph nodes can be accessed.

Retrieval of all Graph Pattern Matches

Graph patterns in EMF-IncQuery and annotations in the proposed approach enable developers to filter out graph nodes from matches for retrieval, because these graph nodes are not relevant for the query result. In EMF-IncQuery, developers filter out graph nodes of matches by omitting these graph nodes in the pattern header. For example, the pattern header in Figure 10.1(b) omits the graph nodes with Namespace and Reference type, because both kinds of graph nodes have no explicit role in the Generalization pattern.

In the proposed approach, developers employ scopes to filter out graph nodes without roles in matches. But, these scopes keep track of these graph nodes, because they belong to the match as well. For example, in Figure 10.1(a) the annotations that mark matches of the Generalization pattern reference artifacts with Namespace and Reference artifact type by means of scopes, because these artifacts have no special role these matches.

When graph nodes are filtered out of matches, tuples in EMF-IncQuery and annotations in the proposed approach may be duplicates, because these tuples contain the same graph nodes and these annotations reference the same graph nodes by means of roles, respectively.

EMF-IncQuery filters out such duplicates of tuples and returns only one tuple for a set of duplicates. It is the responsibility of developers to enforce the retrieval of all tuples with internal nodes but identical header parameter values.

The proposed approach provides two maintenance algorithms that handle duplicates of annotations in a manner that all matches are enumerated by the approach and can be retrieved by end-users accordingly. The original view graph maintenance algorithm (Section 8.10) enumerates all annotations including annotation duplicates and, thus, enables end-users to retrieve all matches. The optimized view graph maintenance algorithm (Section 9.4.2) employs aggregations to keep track of matches that result in annotation duplicates. These aggregations enable the framework to restore and return all matches including matches that result in annotation duplicates.

Retrieval of Reused Graph Pattern Matches

EMF-IncQuery employs pattern headers to describe which graph nodes can be retrieved from maintained graph pattern matches. These headers do not describe which kinds of already found matches are reused for graph pattern matching. Instead, pattern bodies call the definition of other patterns explicitly. Therefore, EMF-IncQuery does not enable to retrieve matches that are reused by patterns to find other matches.

The proposed approach enables to keep track of annotations that are reused to find matches of patterns. Annotations reference reused annotations either with the help of roles, when reused annotations have certain roles in the matches, or scopes, when reused annotations do not have certain roles in the matches. Consequently, the roles and scopes between annotations constitute a dependency graph of annotations and end-users as well as developers can traverse these roles and scopes to retrieve reused annotations.

10.4. Discussion

This chapter compares the proposed approach with EMF-IncQuery concerning their expressiveness, capabilities for optimizations, generality, and retrieval of matches.

In general, EMF-IncQuery and the proposed approach support conjunctions, disjunctions, negations, and recursions. EMF-IncQuery derives the Rete network by means of a heuristic. In the proposed approach, developers model the Gator network explicitly. Thus, the proposed approach offers more possibilities for optimizations concerning memory consumption and execution time than EMF-IncQuery, e.g. network nodes with more than two inputs. EMF-IncQuery employs tuples to store pattern matches. The proposed approach employs attributed graph nodes that mark matches and store additional attribute values at these graph nodes. Originally, EMF-IncQuery is designed for EMF models, but is also adapted to other graph-oriented representation formats. The proposed approach is designed as framework and provides modules as network nodes of discrimination networks, which abstract from employed graph data models and pattern matching technologies.

Moreover, the proposed approach employs a deferred view graph maintenance that enables to filter out modifications events, which cancel each other. Thus, less modification events may be processed by the maintenance algorithm to speed up the maintenance. In contrast, EMF-IncQuery processes each single modification event immediately and, thus, may have to process inefficiently more modification events than the proposed approach.

Both approaches enable developers to retrieve maintained matches instantly, when the discrimination networks are up-to-date. As a difference, EMF-IncQuery enumerates only one match for an internal variable for a given tuple of header variables, thus not all matches of tuples are enumerated. Therefore, EMF-IncQuery does not enable to retrieve all matches, when duplicates of tuples exist. Furthermore, EMF-IncQuery only enables to retrieve graph nodes with explicit roles in matches and, thus, does not enable to retrieve the complete matches. In contrast, the proposed approach enables to retrieve all graph nodes of matches as well as matches that result in annotation duplicates, if required.

11. Performance Evaluation

This chapter describes the evaluation of the proposed view graph maintenance algorithms. The evaluation focuses on the memory consumption for storing view graphs and the execution time for maintaining view graphs. The evaluation aims for a verification of the analytical observations, which are described in Section 10.3.2.

Section 11.1 describes the goals of the performance evaluation. Afterwards, Section 11.2 outlines the implementation of the view graph maintenance algorithms. Section 11.3 presents an interior evaluation that compares the batch and incremental maintenance algorithms of this thesis for Rete and Gator network structures. Section 11.4 presents an exterior evaluation that compares the view graph maintenance algorithms with state-of-the-art approaches. Finally, Section 11.5 compares the analytical observations of Section 10.3.2 with the evaluation results.

11.1. Evaluation Goals

The goal of the performance evaluation is to show a) that the incremental maintenance algorithm outperforms the batch maintenance algorithm and that b) the proposed incremental maintenance algorithm can outperform state-of-the-art approaches for incremental graph pattern matching. Therefore, this thesis aims for an interior and exterior evaluation.

The *interior* evaluation (G5b - Interior Evaluation) compares the batch and incremental maintenance algorithms for Rete and Gator network structures. The *exterior* evaluation (G5b - Interior Evaluation) compares the incremental maintenance algorithm with the incremental graph pattern matching approach of EMF-IncQuery [9]. EMF-IncQuery is the only available and comparable software tool that can be employed for the evaluation. Both evaluations focus on the memory consumption and execution time, when maintaining graph pattern matches.

11.2. Realization

This thesis implements the naive, batch, and incremental view graph maintenance algorithms. The implementation bases on the Eclipse Modeling Framework (EMF). The design decision to employ EMF has several implications. For example, the evaluation employs EMF models that are specializations of typed attributed graphs (cf. Section 2). These EMF models additionally consist of graph edges that describe a) the containment hierarchy of graph nodes and b) graph edges of EMF models have no attributes.

This thesis implements an interpreter for view module dependency graphs. This interpreter generates an execution plan for view modules of the dependency graph taking into account recursion cycles and triggers view modules to execute the embedded graph transformations.

This thesis maps view graph transformation rules to story diagrams and story patterns [102]. For that purpose, this thesis implements a graph transformation from the view graph transformation language of this thesis to story diagrams for each execution mode of the view modules. When the framework executes view modules in certain execution modes, the framework executes the generated story diagrams. The Appendix B.1 shows generated story diagrams for the Generalization pattern.

The change monitoring of base graphs is implemented with the help of EMF-specific monitoring capabilities similar to EMF-IncQuery [9].

11.3. Interior Evaluation

The interior evaluation is twofold. First, the interior evaluation compares the performance of the batch and incremental view graph maintenance algorithms. Second, the interior evaluation compares the performance of these algorithms for equivalent Rete and Gator network structures using the view definition approach of this thesis.

Section 11.3.1 describes the evaluation setup. Section 11.3.2 shows the measurement results that are discussed in Section 11.3.3. Section 11.3.4 discusses the validity of the evaluation.

11.3.1. Evaluation Setup

The interior evaluation deals with the recovery of software design patterns in ASGs of Java source code that is contained by open source software repositories. This evaluation extends to approach of Niere [75] in a manner that the evolution of the ASGs is taken into account. That means, the evaluation repeats the recovery of the employed design patterns for each revision of the source code.

For the evaluation, the Java source code of several open source software repositories was pre-processed to transform the source code of each revision into XML Metadata Interchange (XMI) models. These XMI models constitute the base graphs for the evaluation. For this transformation, the evaluation employs JaMoPP [53] that provides a metamodel for the Java programming language and a parser for Java source code. The running example of this thesis employs a simplified version of the JaMoPP metamodel. The JaMoPP metamodel consists of about 230 classes [53] and, therefore, has a reasonable complexity for this evaluation.

According to the running example, the interior evaluation employs software design patterns [35] as graph queries. The evaluation makes use of a view module dependency graph with 49 view modules. The Appendix F describes the employed view modules and graph patterns. The dependency graph consists of 16 view modules that implement atomic graph conditions, 18 intermediate view modules that implement intermediate graph patterns, and 15 high-level view modules that implement software design patterns.

The evaluation employs the naive, batch, and incremental maintenance algorithms for each revision of the derived XMI models to recover employed software design patterns. The evaluation merges modified XMI models of the next revision into the current revision to modify the base graphs and proceed from the current revision to the next revision. The evaluation employs EMF-Compare [91] to merge the XMI models. The evaluation processes the first hundred revisions of the software repositories.

The evaluation uses a Dell PowerEdge R620 x8 Base server system. The server consists of two Intel Xeon E5-2630 processors. Each processor consists of six cores with 2,3 GHz. However, the implementation does *not* make use of the multi-core capabilities. Furthermore, the evaluation increases the Java heap space to 256 GB of main memory to avoid that the Java garbage collection interrupts the view graph maintenance unexpectedly.

11.3.2. Evaluation Result

This section describes the measurements for the comparison of a) the view graph maintenance algorithms and b) the performance of the Rete and Gator network structure, separately.

Comparison of Maintenance Algorithms

For each software repository, Table 11.1 shows a) the number of artifacts and annotations at revision 1 and 100, b) the memory consumption for storing annotations at revision 1 and 100, and c) the required time for the initial creation of the view graph at revision 1. Table 11.2 shows the execution time of the naive, batch, and incremental algorithms for each software repository. The execution time describes the total time that is required to perform the view graph maintenance for the first 100 revisions of the software repository. Note that the execution time for the initial creation of the view graphs is *not* included in the total execution time, because this creation is always a batch approach. For each revision, the batch and incremental algorithms result in equal annotations in comparison to the naive algorithm.

Repository	**#Artifacts**		**#Annotations**		**Memory View**		**Initial Build**
	Rev. 1	Rev. 100	Rev. 1	Rev. 100	Rev. 1	Rev. 100	Rev. 1
Ant	12242	22174	1767	2982	1,22 MB	2,15 MB	06 s
Subclipse	12467	53621	1352	6171	0,89 MB	3,97 MB	04 s
Commons IO	59423	67458	4330	5129	2,68 MB	3,23 MB	31 s
Xerces	133858	191415	20160	25828	13,39 MB	16,85 MB	11 min 52 s
Commons Collections	228423	230784	22999	23091	14,61 MB	14,66 MB	13 min 47 s

Table 11.1.: Overview of data sets

Repository	**Execution Time for 100 revisions**		
	Naive	Batch	Incremental
Ant	07 min 57 s	09 min 26 s	20 s
Subclipse	26 min 48 s	29 min 35 s	01 min 27 s
Commons IO	24 min 25 s	27 min 26 s	37 s
Xerces	13 h 47 min 52 s	16 h 23 min 24 s	10 min 37 s
Commons Collections	05 h 02 min 57 s	05 h 26 min 01 s	54 s

Table 11.2.: Execution times for each view graph maintenance algorithm

For example, the Apache Ant data set consists of 12242 artifacts at revision 1 and 22174 artifacts at revision 100. The framework derives 1767 annotations at revision 1 and 2982 annotations at revision 100. At revision 1 the view graph requires 1,22 MB of main memory. At revision 100 the view graph requires 2,15 MB of main memory. The initial creation of the view graph requires 06 s. The maintenance algorithms require 07 min 57 s (naive), 09 min 26 s (batch), and 20 s (incremental) to maintain the view graph for the first hundred revisions.

Analogously, Table 11.1 and 11.2 show the evaluation results of the Subclipse, Commons IO, Xerces, and Commons Collections data sets. Appendix H.1 shows the raw data.

Comparison of Network Structures

This section compares the performance of a Gator network structure with an equivalent Rete network structure. This comparison includes the memory consumption of view graphs and the execution times of all view graph maintenance algorithms. The employed Gator network structure consists of 4 view modules that maintain view graphs for the Generalization, Interface Implementation, ReadOperation, and WriteOperation graph pattern (cf. Appendix F). The equivalent Rete network structure consists of 22 view modules. Each view module of the Rete network structure consists of at most 2 inputs. Furthermore, the view modules of the Rete network structure employ graph (sub-)patterns with lower cardinality of graph edges first to reduce the number of intermediate matches as early as possible. The Appendix H.2 depicts the employed network structures. This evaluation employs less view modules for the Gator network structure, because an equivalent Rete network structure requires many more view modules than the Gator network structure. These view modules have to be modeled manually.

The evaluation employs the same data sets as the comparison of the maintenance algorithms. Table 11.3 shows the sizes of the selected data sets including the number of artifacts and annotations at revision 1 and 100. Table 11.3 distinguishes view graphs that are created by the Gator and the Rete network structure. Table 11.3 shows the number of the annotations that are created by both network structures at revision 1 and 100. Moreover, Table 11.4 shows the memory consumption of the annotations at revision 1 and 100 for both network structures. Table 11.5 and Table 11.6 show the total execution times of the Gator and Rete network structure for all kinds of maintenance algorithms for the first 100 revisions. The execution time for the initial creation of the view graphs is *not* included in the total execution time, because this creation is always a batch approach.

Repository	**#Artifacts**		**#Annotations (Gator)**		**#Annotations (Rete)**	
	Rev. 1	Rev. 100	Rev. 1	Rev. 100	Rev. 1	Rev. 100
Ant	12242	22174	36	25	2235	3037
Subclipse	12467	53621	27	76	1600	5834
Commons IO	59423	67458	22	27	2431	2932
Xerces	133858	191415	212	265	47452	54038
Commons Collections	228423	230784	323	323	25830	25991

Table 11.3.: Overview of data sets

Repository	**Memory View (Gator)**		**Memory View (Rete)**	
	Rev. 1	Rev. 100	Rev. 1	Rev. 100
Ant	0,06 MB	0,05 MB	1,20 MB	1,63 MB
Subclipse	0,04 MB	0,12 MB	0,86 MB	3,14 MB
Commons IO	0,03 MB	0,04 MB	1,31 MB	1,58 MB
Xerces	0,41 MB	0,44 MB	25,42 MB	28,96 MB
Commons Collections	0,5 MB	0,5 MB	13,89 MB	13,98 MB

Table 11.4.: Memory consumption of the Gator and Rete network structure

For example, the Gator network structure creates 36 annotations at revision 1 and 25 annotations at revision 100 for the Apache Ant data set. The equivalent Rete network structure creates 2235 annotations at revision 1 and 3037 annotations at revision 100. For the Gator network structure, these annotations result in a memory consumption of 0,06 MB and 0,05 MB at revision 1 and 100, respectively. The equivalent Rete network structure results in a memory consumption of 1,20 MB and 1,63 MB at revision 1 and 100, respectively. For the Gator network structure, the maintenance requires 01 min 03 s (naive), 01 min 07 s (batch), and 04 s (incremental). The equivalent Rete network structure requires 05 min 32 s (naive), 07 min 07 s (batch), and 16 s (incremental).

Analogously, the tables describe the evaluation results for the Subclipse, Commons IO, Xerces, and Commons Collections data sets. Appendix H.2 shows the raw data.

11.3.3. Evaluation Discussion

This section discusses the measurements for a) the comparison of the view graph maintenance algorithms and b) the performance of the Rete and Gator network structure.

Repository	Execution Time (Gator)		
	Naive	Batch	Incremental
Ant	01 min 03 s	01 min 07 s	04 s
Subclipse	03 min 09 s	03 min 13 s	06 s
Commons IO	01 min 25 s	01 min 28 s	03 s
Xerces	09 h 52 min 59 s	09 h 47 min 53 s	19 s
Commons Collections	02 h 15 min 37 s	02 h 15 min 43 s	04 s

Table 11.5.: Execution times of Gator network structure

Repository	Execution Time (Rete)		
	Naive	Batch	Incremental
Ant	05 min 32 s	07 min 07 s	16 s
Subclipse	30 min 13 s	33 min 45 s	47 s
Commons IO	11 min 21 s	13 min 12 s	12 s
Xerces	11 h 42 min 14 s	27 h 32 min 59 s	07 min 20 s
Commons Collections	03 h 41 min 08 s	03 h 58 min 22 s	41 s

Table 11.6.: Execution times of Rete network structure

Comparison of Maintenance Algorithms

Table 11.7 shows the speedup of the incremental algorithm in comparison to the batch algorithm. This discussion employs the batch algorithm for the comparison, because it employs a real view graph maintenance as the incremental algorithm in contrast to the naive algorithm (cf. Section 8.9). Table 11.7 shows the speedup ordered by the size of the data sets. The incremental algorithm is 20,37 to 360,82 times faster than the batch algorithm.

Repository	Speedup
	Batch / Incremental
Ant	28,42
Subclipse	20,37
Commons IO	44,72
Xerces	92,65
Commons Collections	360,82

Table 11.7.: Speedup of the incremental view graph maintenance algorithm

.

Figure 11.1 compares the execution times of the maintenance algorithms for the Apache Ant data set. Appendix H.4 shows the charts for the other data sets. The solid line depicts the number of artifacts in the base graphs for each revision. The dashed line depicts the execution time of the naive algorithm. The dotted line depicts the execution time of the batch algorithm. The dash-dotted line depicts the execution time of the incremental algorithm.

Figure 11.1 shows the increase of the number of the artifacts that are stored by the base graph. For example, the number of artifacts increases at revision 55. From this point, the naive and batch algorithm require approx. 5 seconds more than for the previous revisions. For revision 55, the incremental algorithm requires approx. 3 seconds more than for the previous revision, because the number of the base graph changes is larger due to the added artifacts. Afterwards, the execution time of the incremental algorithm decreases again and remains relatively stable, because the number of the base graph changes is lower. Thus, the execution time of the incremental algorithm only depends on the number of base graph changes.

Comparison of Network Structures

Table 11.8 shows the memory reduction of the Gator network structure in comparison to the Rete network structure. The table shows that the Gator network structure uses 20,40 to 62,07 times less memory than the Rete network structure. The Gator network structure requires

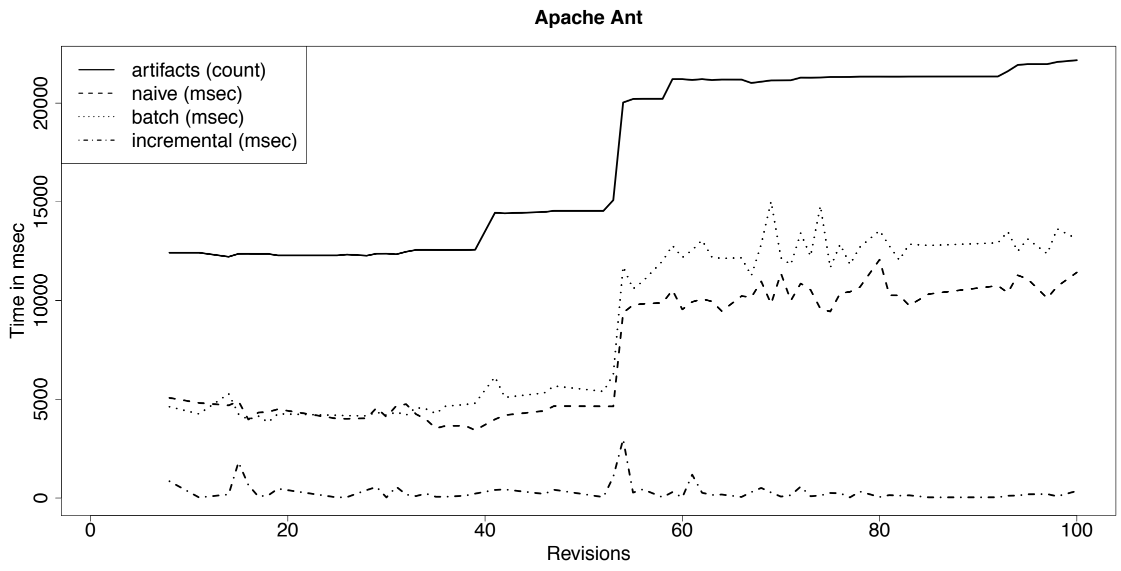


Figure 11.1.: Comparison of the maintenance algorithms for Apache Ant data set

less memory, because it consists of less network nodes than the Rete network structure. Thus, the Gator network structure has to store less intermediate annotations, which mark partial graph pattern matches, than the Rete network structure.

Repository	**Memory Reduction (Rete / Gator)**	
	Rev. 1	Rev. 100
Ant	20,40	30,63
Subclipse	21,04	26,49
Commons IO	38,40	37,49
Xerces	62,07	56,47
Commons Collections	27,64	27,78

Table 11.8.: Memory reduction in comparison to Rete network structure
.

Table 11.9 shows the speedup of the Gator network structure in comparison to the equivalent Rete network structure. For the naive and batch algorithm, the Gator network structure is up to 10,48 times faster than the Rete network structure. For the incremental algorithm, the Gator network structure is up to 22,98 times faster than the Rete network structure.

Repository	**Speedup (Rete / Gator)**		
	Naive	Batch	Incremental
Ant	5,25	6,42	4,49
Subclipse	9,60	10,48	7,44
Commons IO	8,02	8,97	4,47
Xerces	1,18	2,81	22,98
Commons Collections	1,63	1,76	10,39

Table 11.9.: Speedup in comparison to Rete network structure
.

Figure 11.2 compares the execution times of the incremental algorithm for the Gator and Rete network structure for the Apache Ant data set. Appendix H.4 shows the charts for the other data sets. The solid line denotes the execution time of the Gator network structure. The dotted line denotes the execution time of the Rete network structure.

Figure 11.2 shows for each revision that the Rete network structure is slower, when updating the state of the network, than the Gator network structure. This circumstance is especially visible for the peaks in the execution time.

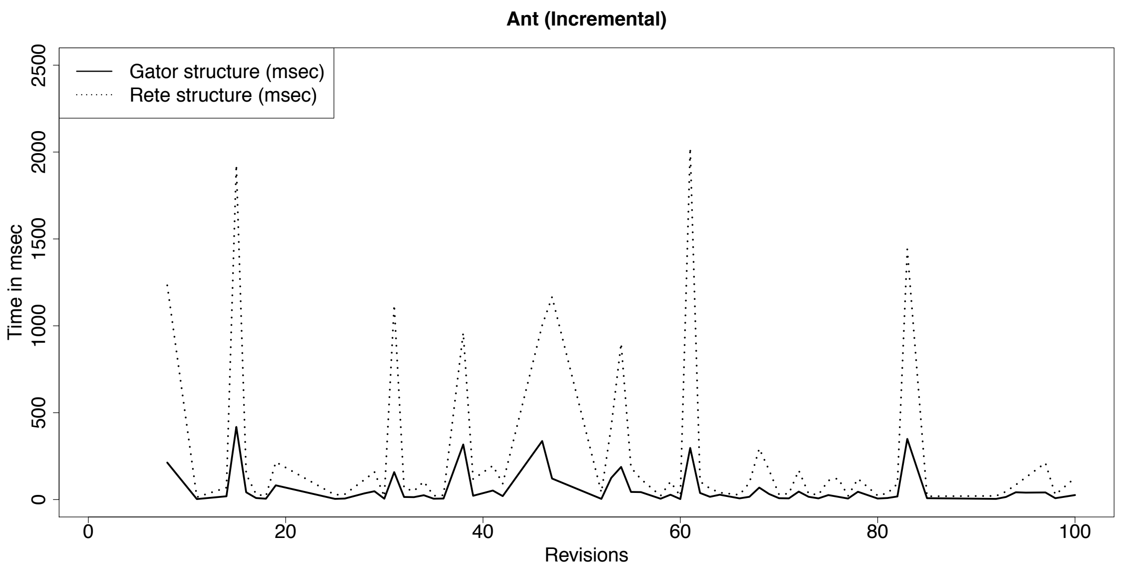


Figure 11.2.: Performance comparison of Gator and Rete network structure for Apache Ant

11.3.4. Threats to Validity

The evaluation results might not be valid due to design decisions concerning the evaluation setup. The evaluation employs one variant of possible Rete network structures that is equivalent to the employed Gator network structure. Other equivalent variants of the Rete network structures may perform better or worse. Thus, the memory reduction and the speedup of the Gator network structure may be different, when another Rete network structure is employed. The same argument holds for the employed Gator network structure that is also one variant of possible Gator network structures. However, both network structures are reasonable choices, because they employ a redundance-free and optimal matching order.

The evaluation performs each measurement only once due to the large execution time of the naive and batch maintenance algorithm. Therefore, the mean execution time of the algorithms may be different. Additional measurements are not required, because the differences of the execution times are clear. Thus, additional measurements do not change to the observation.

The annotations that are maintained by the naive, batch, and incremental algorithm mark the same graph pattern matches. The evaluation checks the equality of these annotations for each revision in comparison to the naive algorithm. Thus, the precision and recall are the same for all maintenance algorithms. Therefore, the evaluation is valid. Note that the concrete values of the precision and the recall are not in the scope of this evaluation.

11.4. Exterior Evaluation

The exterior evaluation compares the incremental maintenance algorithm of this thesis with the incremental graph pattern matching of EMF-IncQuery [9]. The exterior evaluation compares the memory consumption and execution time of the Gator network, which is modeled with the proposed approach, with the Rete network, which is generated by EMF-IncQuery.

Section 11.4.1 describes the evaluation setup. Section 11.4.2 describes the measurements that are discussed in Section 11.4.3. Section 11.4.4 discusses the validity of the evaluation.

11.4.1. Evaluation Setup

This evaluation uses 20 graph patterns for software design pattern recovery. This set of patterns provides a general spectrum of possible patterns. The evaluation implements these patterns with a) the proposed approach in terms of a Gator network and b) with EMF-IncQuery that derives a Rete network. Both implementations employ equivalent patterns (cf. Appendix F and Appendix H.5). The evaluation employs 25 test cases for the 20 graph patterns. These test cases cover a general set of patterns and graph changes. These test cases modify the base graphs to measure the performance of both approaches, when updating the network state. The test cases check whether both approaches retrieve the same matches.

The evaluation executes all test cases in one warmup phase and, afterwards, executes all test cases in ten measurement phases. Then, the evaluation computes the average execution time for the maintenance of the networks. Moreover, the evaluation measures the memory consumption in one additional measurement phase that is independent from the measurement phases for the execution time. One measurement of the memory consumption is sufficient, because the number of required annotations / tuples is the same for all measurement phases.

For the proposed approach, the evaluation measures the memory consumption of annotations, roles, and scopes that are required to mark all graph pattern matches. For EMF-IncQuery, the evaluation traverses the generated Rete network and measures the memory consumption of the network nodes. For example, the evaluation measures the memory consumption of the primary and secondary input of join nodes.

The evaluation only compares the execution time and memory consumption of the incremental algorithms, because EMF-IncQuery does not provide a batch maintenance algorithm. Furthermore, the evaluation employs the *managed* incremental graph pattern matching engine of EMF-IncQuery, because this kind of engine is recommended by the documentation.

The setup of the exterior evaluation is different to the setup of the interior evaluation, because the proposed approach and EMF-IncQuery differ in technical details that make it difficult to a) ensure equivalent search spaces and b) compare the retrieved graph pattern matches. For example, EMF-IncQuery does not return all graph nodes of matches depending on the user-defined pattern header and, thus, the retrieved matches cannot be compared easily in an automated manner with the matches that are retrieved with the proposed approach.

11.4.2. Evaluation Result

Table 11.10 summarizes the execution time and memory consumption. Appendix H.3 shows the raw data. The total execution time describes the time that is required to maintain the discrimination network for all test cases. In total, the proposed approach requires 5,34 s to maintain the modeled Gator network. EMF-IncQuery requires 4,96 s to maintain the generated Rete network, in total. The average execution time describes the mean of all execution times. In average, the maintenance of the Gator network in the proposed approach requires 0,21 s. The maintenance of the Rete network in EMF-IncQuery requires 0,20 s in average. The average memory consumption of the discrimination networks is 13,19 kb in the proposed approach and 21,40 kb in EMF-IncQuery.

Approach	Execution Time		Memory Consumption
	Total	Avg.	Avg.
Proposed Approach	5,34 s	0,21 s	13,19 kb
EMF-IncQuery	4,96 s	0,20 s	21,40 kb

Table 11.10.: Comparison of execution time and memory consumption
.

Table 11.11 shows the measurements for six test cases in detail. For example, the test case for the FieldAssignment pattern (cf. Appendix H.5) requires in average 0,17 s and 0,21 s in the proposed approach and EMF-IncQuery, respectively. Furthermore, the proposed approach requires 11,00 kb and EMF-IncQuery requires 20,91 kb to store the found graph pattern matches. Table 11.11 describes the measurements for the other test cases analogously.

		Graph Patterns					
		Composite	FieldAssignment	Generalization	Multi-Level Generalization	PublicInstanceMethod	Singleton
Time	Proposed Approach	0,26 s	0,17 s	0,18 s	0,18 s	0,19 s	0,19 s
	EMF-IncQuery	0,25 s	0,21 s	0,20 s	0,19 s	0,19 s	0,18 s
	Speedup	0,96	1,24	1,11	1,06	1,0	0,95
Memory	Proposed Approach	12,25 kb	11,00 kb	8,01 kb	11,51 kb	10,68 kb	11,65 kb
	EMF-IncQuery	33,14 kb	20,91 kb	9,04 kb	32,39 kb	22,87 kb	27,42 kb
	Memory Reduction	2,71	1,90	1,13	2,81	2,14	2,35

Table 11.11.: Speedup and memory reduction in comparison to EMF-IncQuery
.

11.4.3. Evaluation Discussion

Table 11.10 shows that the proposed approach is with a speedup of 0,93 slightly slower than EMF-IncQuery, but EMF-IncQuery requires 1,62 times more memory.

Table 11.11 shows that the overall performance is a trade-off between the memory consumption and execution time. For example, the Composite pattern and Singleton pattern are with a speedup of 0,96 and 0,95 a little bit slower for the proposed approach in comparison to EMF-IncQuery. However, EMF-IncQuery consumes 2,71 and 2,35 times more memory than the proposed approach.

Moreover, Table 11.11 shows that the test cases for the FieldAssignment, Generalization, Multi-Level Generalization, and PublicInstanceMethod patterns perform better in time and space at the same time for the proposed approach in comparison to EMF-IncQuery. For example, the test case for the FieldAssignment pattern is 1,24 times faster than EMF-IncQuery. Furthermore, EMF-IncQuery uses 1,90 more memory in comparison to the proposed approach.

11.4.4. Threats to Validity

The evaluation results might not be valid due to design decisions of the evaluation setup. The following paragraphs distinguish threats concerning the employed network structures and the measurements of execution times and memory consumptions.

The proposed approach employs a modeled Gator network structure. EMF-IncQuery generates the Rete network structure by means of a heuristic. Thus, the evaluation employs one variant of possible Gator networks and Rete networks in the proposed approach and EMF-IncQuery, respectively. Other network structures may perform better or worse and, therefore, may result in different speedups and memory reductions.

For the proposed approach, it is well-known which objects must be taken into account to measure the memory consumption. In contrast, the evaluation employs a reverse engineered solution for EMF-IncQuery that measures the memory consumption of network nodes for storing partial graph pattern matches. This solution may overestimate or underestimate the memory consumption. Another memory consumption of EMF-IncQuery results in another memory reduction of the proposed approach.

EMF-IncQuery performs an immediate maintenance of the internal Rete network. This makes it difficult to measure the time that is required by EMF-IncQuery to update the state of the Rete network, because the measurement instrumentation cannot distinguish the time that is required to apply the graph changes and the time that is required to update the Rete network. In contrast to EMF-IncQuery, the proposed approach enables to defer the view maintenance to a later point in time. Therefore, the time that is required by the proposed approach to update the Gator network can be measured independently from the time that is required to modify the base graph. Consequently, the measured execution times are more precise for the proposed approach. In contrast, the execution times for EMF-IncQuery are rather pessimistic, because they include also the time that is required to modify the graphs.

Furthermore, network nodes in EMF-IncQuery employ an effective index structures to lookup matching tuples [9] and, thus, these network nodes do *not* employ real graph pattern matching. In contrast to EMF-IncQuery, the network nodes of the proposed approach employ graph pattern matching. Thus, the comparison is unbalanced.

11.5. Discussion

The evaluation shows the trade-off between fine- and coarse-grained network structures and, thus, verifies the analytical observations of Section 10.3.2. Fine-grained network structures consist of network nodes with few inputs. These network nodes require low effort for pattern matching, because they encapsulate small patterns. However, the memory consumption of the overall network is high, because many network nodes are required and many partial matches are stored. Many partial matches are used to continue the matching. Coarse-grained network structures consist of network nodes with many inputs. These network nodes require high effort for pattern matching, because they encapsulate large patterns. However, the memory consumption of the overall network is low, because few network nodes are required and few partial matches are stored. Few partial matches are used to continue the matching.

Other performance variables are the employed pattern matching algorithm of network nodes, the efficient aggregation of modification events, and an explicit update maintenance phase for modified graph nodes to avoid deletions and re-creations of matches. The proposed approach employs these additional performance optimization in contrast to EMF-IncQuery.

The proposed approach also enables to model Rete networks. Thus, the proposed approach can fall back to the performance of Rete network structures. As the evaluation results show, the proposed approach *can* be better in space during performing equally in time. Gator networks enable to move the space complexity of the overall network into network nodes with a higher time complexity and vice versa.

12. Related Work

This chapter describes related work and compares it with the proposed approach. Section 12.1 describes discrimination networks and discusses how the proposed approach extends these discrimination networks. Section 12.2 describes application domains for view maintenance and discusses how the proposed approach can be employed in the context of these domains.

12.1. Discrimination Networks

Discrimination networks enumerate all objects that satisfy certain conditions and enable to update these enumerations efficiently, when objects change. Discrimination networks consist of networks nodes that implement condition tests. When an object satisfies a condition test, the network node stores the object for later retrieval. The network edges between the network nodes forward the objects, which passed the condition tests, to successor network nodes.

Different kinds of discrimination networks exist. The literature distinguishes Rete networks, Treat Networks, and Gator Networks as most important kinds of discrimination networks. The following sections describe their network structures and maintenance algorithms.

Rete Networks

Forgy [32] introduces the Rete matching algorithm to efficiently compare *"a large collection of patterns to a large collection of objects"* [32] for the purpose of finding *"all objects that match each pattern"*. The Rete matching algorithm efficiently enumerates all objects that match a pattern, because the algorithm does not iterate over the complete set of all objects, when just a few objects changed in this set. This is achieved by storing information that describes which object matches which pattern or sub-patterns of this pattern. When this information is kept up-to-date, iterating the complete set of objects for pattern matching is avoided, because the information which patterns match which objects is still stored for easy retrieval.

Rete networks consist of one root node, one-input nodes, two-input nodes, and terminal nodes. Root nodes distribute changes of objects to other network nodes. One-input nodes consist of one input and perform *"intra-element tests"* [32], such as testing conditions on attribute values. Two-input nodes consists of two inputs and perform *"inter-element test"* [32], such as testing conditions that refer to objects of both inputs, e. g. join conditions. Two-input nodes store objects received from predecessor network nodes of the left and right input separately. Terminal nodes consist of one input and terminate the network. Terminal nodes store objects that satisfy a certain pattern.

The Rete matching algorithm propagates tokens that represent object changes through the Rete network. A token indicates whether it is a positive or negative token and carries an object. A positive token indicates that the carried object was added to the overall set of objects. A negative token indicates that the carried object was removed from the overall set of objects. When an object is modified two tokens are propagated through the network. First, a negative token indicates that the old state of the object was deleted. Afterwards, a positive token indicates that the new state of the object was added.

When a positive token arrives at an one-input node, the network node checks whether the carried object satisfies the intra-element test. If yes, the network node sends this positive token to the successor network nodes. Otherwise, the network node rejects the token. When a negative token arrives at an one-input node, the network node sends this negative token to the successor network nodes, because the object that previously satisfied the intra-element test does not exist anymore.

When a positive token arrives at a two-input node, the network node checks whether the carried object satisfies the inter-element test together with any object that arrived at the second input of the network node. For each combination of objects that satisfies the inter-element test, a positive token is send to successor network nodes. This positive token carries both objects that satisfy the inter-element test of the network node. Combinations of objects that do not satisfy the inter-element test are rejected. When a negative token arrives at a two-input node, *"a token with an identical part is deleted"* [32] from the internal memory of the network node and a negative token is send to successor network nodes to inform them that the object carried by the token does not satisfy the inter-element test anymore.

When a positive token arrives at a terminal node, this token is added to the internal memory of the terminal node. When a negative token arrives at a terminal node, this token is removed from the internal memory of the terminal node.

Treat Networks

Miranker [73] presents Treat as a better matching algorithm for finding all objects that satisfy certain conditions. Treat aims for overcoming the disadvantages of Rete networks such as high memory consumption and deletions of objects from the network that are as expensive as the additions of objects to the network.

Treat networks only consist of network nodes with at most one input. These network nodes perform intra-element tests. Objects that satisfy the intra-element test are stored by so called alpha-memories. Treat networks do not employ two-input nodes and, thus, do not store results of partial inter-element tests, e. g. join conditions. Treat networks store objects that satisfy a complete condition in so called conflict sets that are similar to terminal network nodes.

The Treat algorithm employs additional internal memories per network node in comparison to the Rete matching algorithm. Treat networks partition these memories into three parts to store already processed objects (old memory), new added objects (new-add memory), and new deleted objects (new-delete memory). Treat networks use these kinds of memories to constrain the search space for new pattern matches and pattern matches that must be deleted. New added objects are temporarily added to the new-add memory and deleted objects are temporarily added to the new-delete memory. For additions of objects, Treat networks compare objects in the old-memory and new-add memory. Then, the algorithm evaluates the overall condition test and adds found matches to the conflict set. Afterwards, the algorithm adds objects of the new-add memory to the old-memory for later comparisons. For deletions of objects, Treat networks compare objects in the old-memory and new-delete memory. Then, the algorithm removes matches, which contain the deleted objects, from the conflict set. Afterwards, the algorithm removes the deleted objects from the old-memory.

Gator Networks

Hanson et al. [48] propose a generalized discrimination network structure called Gator network. Gator networks enable to create and maintain intermediate forms of discrimination networks by taking the advantages and disadvantages of Rete and Treat networks into account. Hanson et al. state that *"with Gator, it is possible to get additional advantages from* *optimization"* [48], because network nodes that compute and maintain results of inter-element tests *"are only materialized when they are beneficial"* [48]. As a result, Gator networks allow to control the trade-off between memory consumption and the time that is required to update the state of the network.

In general, the network node types of Gator networks are similar to the network node types of Rete and Treat networks. The main difference is that Gator networks can employ network nodes with more than two inputs for inter-element tests. Then, an internal evaluation order plan describes the order of the internal evaluation of the inter-element test. The evaluation order plan is defined, when the Gator network is constructed.

Moreover, Gator networks can employ additional memory network nodes that store or do not store objects that satisfy condition tests. These memory network nodes enable to materialize intermediate results only when they are beneficial concerning the performance of the network. For the sake of simplicity, this thesis omits the description of these memory network nodes. Hanson et al. [48] give a description of these memory network nodes.

Similar to Rete networks, Gator networks propagate tokens that carry added or deleted objects through the network to maintain the internal memory of network nodes.

Discussion

The discussion of the discrimination networks focuses on their network structures and maintenance algorithms. The following paragraphs consider only the original kinds of discrimination networks. This chapter considers adaptations of these discrimination networks, later on.

In general, all three kinds of discrimination networks employ *acyclic* directed graphs as network structures. According to [67], discrimination networks must be either left- or right-associative and must not consist of convergent network paths. Convergent network paths are branched network paths that join again with the network path from which they originate. If discrimination networks consist of convergent network paths, the maintenance algorithms can lead to missing or duplicated objects at the terminal network nodes. Lee et al. [67] provide a counter-example for Rete networks with convergent network paths. Thus, discrimination networks are limited concerning their overall topology.

The approach that is presented in this thesis extends the concept of discrimination networks by *cyclic* directed network structures. These cyclic network structures enable developers to express recursive definitions of condition tests. The original discrimination networks do not support these recursive definitions.

Furthermore, the presented approach supports network nodes that can have more than two inputs in contrast to Rete networks and, therefore, the presented approach adapts the generalized network nodes of Gator networks. However, the network nodes of the original discrimination networks have rather a symbolic nature than a functional purpose, because they neither hide the condition test nor provide any information about the condition test. The approach that is presented in this thesis extends to concept of these network nodes by understanding them as an abstraction for employed condition tests. Therefore, in the proposed approach network nodes consist of interfaces that hide conditions and enable the maintenance algorithm to process network nodes uniformly and independent from the employed condition test. Therefore, also different languages can be employed to specify these condition tests. Consequently, in the proposed approach the network nodes are generic, because they are not dedicated to a specific purpose such as intra- or inter-element tests.

The maintenance algorithms of all three kinds of discrimination networks employ an immediate maintenance of the network state, when objects change. For that purpose, the algorithms propagate the creation or deletion of a single object through the network by means of tokens that carry this object. In contrast, the proposed approach is able to aggregate object changes in an efficient manner by cleaning up object changes that cancel each other to employ a deferred maintenance of the network state. The original discrimination networks do not support such a deferred maintenance and propagate each change as single token immediately.

For a modified object, the original algorithms propagate one token that describes the deletion and one token that describes the creation of the object through the network. This mapping of object modifications is *no* real maintenance, because the identity of (partial) condition test results changes, when they are deleted and re-created, afterwards. In contrast, the proposed approach avoids the deletion and re-creation of condition test results by an update of condition test results. This update only deletes (partial) test results, when they do not satisfy the condition test anymore. Otherwise, the update preserves the (partial) test result. Therefore, (partial) test results keep their identity and, thus, the proposed approach performs a real maintenance of test results.

The proposed approach propagates changes through the network in a different manner than the original discrimination networks. The original discrimination networks propagate objects through the network without respecting the topological order of network nodes efficiently. For example, when a network node with two inputs receives a token at the first input, the network node does not wait until the network that is connected to the second input processed all object changes. In contrast, the proposed approach respects the topological order of the network nodes and executes network nodes only, when all their predecessor network nodes processed all object changes already. Due to this enhancement, the proposed approach supports convergent network paths in contrast to the original discrimination networks. Furthermore, the proposed approach employs ordered maintenance phases to process object changes. These maintenance phases are not present in the algorithms of the original discrimination networks.

In the original discrimination networks, the network nodes with two inputs check whether objects that are received at the first input satisfy the condition test of the network node together with any object that was received at the second input. Furthermore, network nodes with more than two inputs employ a static evaluation plan to evaluate the condition test of the network node in an efficient manner. In contrast, the proposed approach employs a reachability test for graphs to prune the search space and it is up to the network node to evaluate the condition test efficiently. Thus, the proposed approach extends the original discrimination networks by transferring the optimization task to the developer of the condition test. The existing approaches employ fixed heuristics to find a good network structure. In contrast, the proposed approach enables to model and optimize discrimination networks manually.

In summary, the proposed approach combines all three kinds of discrimination networks and is able to emulate the original discrimination network structures. The proposed network structure enables to mix the associativity of the network, supports convergent network paths due to a maintenance algorithm that takes the topological order of network nodes into account, supports network nodes with more than two inputs to steer the trade-off between memory consumption and execution time of the network, supports recursive condition tests, performs a real maintenance for modified objects, and enables developers to manually optimize the network structure and the encapsulated condition tests.

12.2. Applications of Incremental Graph Processing

This thesis proposes modeling techniques and maintenance algorithms that can be employed for several domains that deal with graphs. The following paragraphs describe application domains, which are related to the incremental processing of graphs. Section 12.2.1 describes view maintenance for databases. Section 12.2.2 outlines graph indexing approaches that enable the efficient lookup of graph nodes and subgraphs that satisfy certain constraints. Section 12.2.3 summarizes approaches for querying graphs. Section 12.2.4 describes techniques for incremental graph matching in Model-Driven Engineering (MDE). Section 12.2.5 describes dedicated approaches for incremental activities in MDE.

12.2.1. Database Views

Databases employ different data models to store graphs, tables, and objects. In general, databases provide declarative (e. g., Cypher [82]) and imperative (e. g. Neo4j API [82]) query languages. Declarative query languages describe how expected query results look like that satisfy the query. Imperative query languages enable to implement the algorithm that retrieves the query results. In both cases, evaluating queries can become very time-consuming, when the size of the stored data and the queries become large. Views can increase the throughput of databases by maintaining query results. The following sections summarize techniques for the view maintenance of graph-structured, relational, and object-oriented databases.

Graph Database Views

Kiesel et al. [62, 61] present GRAS. GRAS is a database system for software engineering that employs typed attributed graphs. Kiesel et al. [61] *"state that a graph database system should support the incremental computation of derived data"* without providing a clear definition of derived data. However, their papers [62, 61] provide no hints that GRAS supports views for graphs and, thus, also supports no view maintenance for graphs.

Zhuge et al. [101] define the notion of graph-structured databases as well as the notion of virtual and materialized views for graph-structured databases. According to Zhuge et al. [101], materialized views employ delegates that reference nodes in the graph-structured data and other materialized views. Their approach employs selection paths and conditions to define the content of these views. Their maintenance procedure re-evaluates these selection paths and conditions for created, deleted, and modified nodes to maintain the materialized views. However, the authors limit their approach to tree-structured data and do not support graph patterns for the definition of views. Their approach does not employ discrimination networks.

Srinivasa et al. describe GRACE [88]. GRACE is a graph database system that enables to search for graphs and subgraphs. For that purpose, GRACE employs an attribute value index, a graph location index, and path index. However, the authors do not comment on how these indexes are maintained. Furthermore, GRACE does not support views for graphs.

Khurana et al. [60] present a system for snapshot retrieval of graph data. These snapshots are copies of graph data and represent one kind of database view. Such copies are very inefficient concerning storage consumption. For that purpose, the authors reduce the storage consumption of these snapshot by deriving graph deltas from the modifications of the graphs. When graphs change, the approach creates a new snapshot using these graph deltas. Thus, the approach does not support the maintenance of graph views.

Angles [2] compares current graph database models that are used in practice such as Neo4j [82]. Current graph databases only support graph indexes. Angles does not compare graph databases concerning their support of graph views. The manuals of current graph databases (e. g., the Neo4j manual [90]) provide no hints that they support graph views.

Relational Database Views

Varró et al. [93] show that views of *relational* databases can be used to store graph pattern matches. However, their approach employs no incremental maintenance of these *relational* views and employs no native graph data model for storing graphs and views. Thus, these relational views suffer the same problems as databases that store graphs in a relational data model such as expensive join operations to construct relationships between entities. However, Varró et al. [94] and Bergmann et al. [8] consider incremental graph transformations based on relational databases.

Several kinds of view maintenance algorithms for relational database views exist that employ an impact analysis or derive maintenance rules to maintain the views. The approaches, which employ an impact analysis, determine tuples that must be added to, removed from, or modified in views. For example, Shmueli et al. [86] employ a good-bad marking scheme, Blakeley et al. [17] determine whether view definitions become satisfied or dissatisfied due to changes of base tables, and the Propagation and Filtration algorithm by Harrison et al. [49] approximates tuples, which may impact views.

The approaches, which employ maintenance rules to keep derived views consistent, use the view definition and applied queries to derive incremental update queries for views. For example, Ceri et al. [22] derive production rules to propagate changes from base tables to view tables and Qian et al. [78] derive incremental relational expressions from relational expressions of view definitions by means of equivalence-preserving transformation rules.

Other approaches for view maintenance of relational databases, deal with duplicate handling [46], usage of partial information for view maintenance [45], and minimization of view downtime, when maintaining views [24]. Furthermore, approaches exist that aim for cost reduction, when maintaining views. For example, Ross et al. [84] propose to maintain additional views that reduce the overall cost of the view maintenance. Mistry et al. [74] exploit query expressions that are common to multiple view definitions. Colby et al. [25] propose to employ multiple view maintenance policies at once.

Also discrimination networks are used for view maintenance of relational databases. Then, the terminal nodes of the networks are considered as views. For example, Hanson et al. [48] employ Gator networks for incremental view maintenance of relational databases.

Object-Oriented Database Views

Object-oriented databases extend relational database. Object-oriented databases map objects to tables and consider the polymorphism of objects. The view maintenance must consider these object-oriented concepts. MultiView by Kuno et al. [65] derives and maintains virtual classes from base classes and other virtual classes. For view maintenance, MultiView exploits membership- and value-dependencies between objects. Liu et al. [68] employ Object Relational SQL (QR-SQL) and perform a query-rewriting to make object-oriented concepts explicit in view definitions for view maintenance. Akhtar et al. [1] employ the Object Query Language (OQL) and analyze view definitions to derive incremental view maintenance plans.

Discussion

In summary, current graph databases do not provide capabilities for the definition and maintenance of graph views. Only view maintenance approaches for relational and object-oriented databases are available. However, mapping view maintenance for graphs back to the relational domain is inefficient, because then the graphs are not stored natively anymore and suffer additional join-performance overhead of relational databases, when relationships between entities are constructed. Therefore, the approach of this thesis should be employed for view maintenance of graph databases. Then, each view module can be considered as view, because each module knows all annotations that mark matches of the encapsulated patterns.

12.2.2. Graph Indexing

Graph indexes aim for a faster graph query evaluation. They avoid the sequential scanning of graphs, when graph nodes and edges with certain properties have to be looked up during graph pattern matching. In general, two kinds of graph indexing approaches exist. The first kind indexes graph nodes that are reachable by means of the same paths in graphs. This thesis refers to this kind of graph indexing as *path-based indexing.* The second kind indexes graphs that have the same or similar structures in comparison to a given graph. This thesis refers to this kind of graph indexing as *structure-based indexing.*

Path-Based Indexing

Goldman et al. describe *DataGuides* [42]. DataGuides are graphs that describe the structure of indexed graphs. A graph node in a DataGuide represents graph nodes that are reachable by means of the same paths in an indexed graph. A graph edge in a DataGuide represents graph edges with the same label between two graph nodes in an indexed graph.

Milo et al. propose *T-Index* [72]. The approach constructs a non-deterministic automaton with states that represent equivalence classes of graph nodes and transitions that represent graph edges between graph nodes of these equivalence classes. The equivalence classes group graph nodes in the index structure in a manner that these graph nodes are reachable by means of the same paths in an indexed graph.

Cooper et al. present *Fabric* [26]. Fabric translates graphs into prefix strings and indexes these strings by means of Patricia Tries. Then, Fabric translates graph queries into prefix strings to lookup graphs in the index structure. Furthermore, Fabric supports so called refined paths that are manually added to the index to mark the answer for certain queries. However, the maintenance of these refined paths is not discussed by the authors.

Chung et al. propose *APEX* [23]. APEX is a graph index that adapts its index structure, when the workload of queries changes. For that purpose, APEX employs a hash tree and adapts this hash tree, when users state queries more often than other queries.

Kaushik et al. present *A(k)-Index* [59] as extension of 1-Index [72]. A(k)-Index groups graph nodes in the index that are k-bisimilar to reduce the index size. Two nodes are k-bisimilar, if the sets of paths with length k that reach these graph nodes are identical.

Srinivasa et al. describe the *label walk index* [87]. The approach employs a tree that indexes sequences of nodes with the help of node labels. For each indexed node sequence, a list of graphs is provided that contain the node sequence.

Structure-Based Indexing

Messmer et al. [71] propose a graph index that enables to retrieve graphs based on graph and subgraph isomorphism. The approach employs decision trees that index permutations of adjacency matrices. These matrices represent mapping between graphs. The approach maps the graph and subgraph isomorphism problem to the problem of finding these permutations.

Yan et al. present *gIndex* [97, 98]. The approach uses discriminative frequent structures of graphs as indexing feature. The index lists all graphs that contain certain graph structures. Then, queries use the index to prune search spaces and, afterwards, employ the isomorphism test to check for real isomorphic subgraphs.

Yan et al. also present *Grafil* [99]. Grafil aims for retrieving graphs that are similar to graph queries. Grafil employs a feature-graph matrix and edge-feature matrix that index occurrences of subgraphs and edges of these subgraphs, respectively. Furthermore, Grafil relaxes graph queries by removing graph edges and employs a similarity measurement to find graphs that are similar to a graph query.

He et al. describes *Closure-Tree* [50]. Closure-Tree employs an index tree. Nodes in this tree represent graph closures, which capture the structural information of graphs that are represented by child nodes in the tree. When users state queries, Closure-Tree looks up a candidate set of graphs by means of an approximated subgraph isomorphism test. Afterwards, Closure-Tree employs a real subgraph isomorphism test to the remaining graphs.

Williams et al. [95] propose a graph index that aims for the improvement of subgraph isomorphism tests. Their approach employs decomposition graphs that index all connected and induced subgraphs of given graphs.

Zhang et al. describe *TreePi* [100]. TreePi employs frequent trees in graphs as indexed features instead of frequent subgraphs. TreePi uses these frequent trees to derive a candidate set, when users state a graph query. Then, TreePi constructs the query result by means of the trees in the candidate set.

Discussion

In summary, graph indexes can increase the performance of graph pattern matching by enabling a fast lookup of graph nodes, graph edges, and graph-structures based on certain graph properties such as attribute values, reachability and similarity. However, graph indexes do *not* index graph pattern matches of user-defined patterns. Furthermore, the incremental maintenance of the graph indexes is *not* in the scope of the presented approaches.

12.2.3. Graph Querying

This section describes existing graph querying approaches. This thesis distinguishes graph search and model search. Graph search deals with the retrieval of graph nodes and edges in a broader sense and abstracts from graph languages and application domains. Model search deals with the retrieval of models as special kinds of graphs that are expressed in certain modeling languages, e. g. UML.

Graph Search

Fan et al. [30] describe algorithms for incremental graph pattern matching. They describe algorithms for graph simulation, bounded simulation, and subgraph isomorphism. The authors prove the complexities of these algorithms and conclude that the cost of incremental graph pattern matching *"is not determined by the size of the changes alone"* [30]. They do not employ discrimination networks for incremental graph pattern matching.

In another paper, Fan et al. [31] describe how views for graphs can be used to answer graph queries. They provide an algorithm that determines whether a query can be answered with a given set of views without the need to access base graphs. Furthermore, they provide an algorithm that enables to compute query results efficiently with a given set of views. Moreover, they provide an algorithm that enables to determine a minimal set of views which should be used for answering a query. However, the authors do not consider the maintenance of views.

Giugno et al. [40] describe *GraphGrep* as application-independent method for retrieving all occurrences of a subgraph in a graph database. Their approach employs regular expressions as query language and employs a hashing approach to represent graphs in an abstract form. This hashing is used the prune the search space, when users state queries. Afterwards, the approach looks up all exact subgraphs in the set of the remaining graphs.

Model Search

Models are special kinds of typed attributed graphs. In practice, model repositories store these models. Then, model search engines create search indexes for these model repositories and query engines browse these indexes to retrieve models and model elements as answer for model queries. Queries can be keywords, OCL expressions, or graph patterns.

Gomes et al. [43] describe an approach for the case-based retrieval of UML models that are similar to other models. For that purpose, their approach employs similarity metrics that exploit synonyms of UML element names and semantic relations between these names.

Kling et al. describe *MoScript* [63] as domain-specific language for querying model repositories. MoScript employs a megamodel [52] to capture models and their relationships. This megamodel acts as search index. MoScript uses OCL to lookup model representations in the megamodel and dereference these representations to retrieve physical models.

Bozzon et al. [18] describe an approach that enables to search for models that are developed by means of MDE practices. Their approach exploits metadata of models and mines meaningful information to create a search index for these models.

Lucrédio et al. present *Moogle* [69]. Moogle maps model search to text-based search. Moogle employs the full-text search engine Apache SOLR and, therefore, creates model descriptors that conform to the schema of Apache SOLR.

Discussion

In summary, graph search and model search approaches aim for retrieving all (sub-)graphs and models that satisfy a graph query. For that purpose, the approaches propose lookup algorithms, map the search to another search engine, or generate search indexes. The incremental maintenance of these search indexes is not described by the presented approaches. Furthermore, these approaches do not maintain graph pattern matches. But, graph search and model search engines can make use of maintained matches as shown by Fan et al. [31]. However, the approach of Fan et al. [31] does *not* consider the maintenance of views.

12.2.4. Discrimination Networks in Model-Driven Engineering

Bunke et al. [19] transfer the concepts of Rete networks and the Rete matching algorithm to the efficient implementation of graph grammars for directed labeled graphs. Their approach derives the Rete network from the left-hand side of graph grammar productions. These Rete networks consist of five different network node types called root node, node checker, edge checker, subgraph checker, and production nodes.

Similar to original Rete networks, a root node serves as input of the Rete network. This root node is connected to all node checkers and edges checkers in the Rete network and sends incoming graph nodes and edges to node checkers and edge checkers, respectively. Node checkers test whether nodes have a certain label. If yes, these nodes are forwarded to production nodes, which store nodes that are a match of the left-hand side of a production. Node checkers consist only of one outgoing edge and are not connected to other network nodes than production nodes. Edge checkers test whether edges have a certain label and whether the source and target nodes of these edges have a certain label. If all three conditions are satisfied, edges are send to subgraph checkers and production nodes. Subgraph checkers have two inputs and receive subgraphs from different edge checkers. Subgraph checkers combine two subgraphs to larger subgraphs, when both subgraphs have certain nodes in common. Subgraphs that result from the combination of subgraphs are send to successor subgraph checkers as well as production nodes, if required. Production nodes receive graph nodes, graph edges, and subgraphs from node checkers, edge checkers, and subgraph checkers. Thus, production nodes enumerate all matches of the left-hand sides of graph grammar productions and, therefore, describe which graph grammar productions are applicable.

Changes that result from the application of graph grammar productions are propagated through the network similar to the original Rete matching algorithm. When graph nodes and graph edges are added, they are propagated as added elements through the network and, if necessary, resulting subgraphs are computed by subgraph checkers and stored by production nodes. When graph nodes and graph edges are deleted, they are propagated as deleted elements through the network. Then, subgraph checkers and production nodes remove subgraphs that contain these elements.

Bergmann et al. [9] extend the approach of Bunke et al. [19] by transferring the concepts of Rete networks from graph grammars to graph transformations in the context of the MDE domain. Graph transformations perform model manipulations and, thus, matches of graph patterns appear and disappear. For that purpose, Bergmann et al. [9] describe an incremental matching engine that explicitly stores graph pattern matches and maintains these matches, when the models change. This incremental matching engine adapts Rete networks and the Rete matching algorithm [32] to enable the transformation language of the VIATRA2 framework to make use of matches that are maintained within a Rete network. Then, matches can be retrieved *"in constant time excluding the linear cost induced by the size of the result set itself"* [9]. Bergmann et al. [9] state that *"the main ideas behind the incremental pattern matcher are conceptually similar to relational algebra"*. In the approach of Bergmann et al. [9], the internally employed Rete network represents graph pattern matches by means of tuples that consist of model elements. Furthermore, each network node in the Rete network is related to a (partial) pattern and stores a set of tuples that satisfy this (partial) pattern. Bergmann et al. [9] state that *"this set of tuples is in analogy with the relation concept of relational algebra"* and, therefore, the authors map incremental pattern matching as challenge of the graph domain back to the relational domain. Indeed, that means the authors do not provide a native solution by using concepts of the graph domain. Instead, Bergmann et al. [9] provide several extensions of the network structure that make Rete networks applicable to graph transformations. The authors [9] state their *"solution provides full support for the rich language constructs of VIATRA2"* and, therefore, *"significantly supersede and extend the first (and relatively old) RETE-based graph transformation approach"* that is presented by Bunke et al. [19]. The following paragraphs describe the extension of the original Rete network approach.

The approach of Bergmann et al. [9] extends Rete networks with additional kinds of network nodes. In general, Rete networks in VIATRA2 distinguish input nodes, intermediate nodes, and production nodes.

Input nodes represent the knowledge contained by models. These models must conform to metamodels. Rete networks in VIATRA2 employ one input node per node type and edge type of these metamodels. These input nodes implement node type constraints and edge type constraints. Furthermore, Bergmann et al. [9] state that *"miscellaneous input nodes represent containment, generic type information, and other relationship between model elements"* and, therefore, multiple kinds of input nodes seem to exist without explicit discussion.

Intermediate nodes store matches of partial graph patterns. Rete networks in VIATRA2 distinguish intermediate nodes such as join nodes, negative nodes, and term evaluator nodes. All intermediate nodes consist of two inputs. Join nodes implement natural joins for tuples that are received by the first and second input. These joins use an effective index structure to check whether tuples of the first input can be joined with tuples of the second input. Negative nodes implement anti-joins for tuples that are received by the first and second input. Negative nodes store tuples that are received by the first input and cannot be joined with any tuple that is received by the second input. Term evaluator nodes implement attribute conditions such as arithmetical and logical functions. Term evaluator nodes only propagate tuples to successor nodes, when they satisfy the attribute conditions.

Production nodes store complete matches and also perform additional tasks such as projections to filter out elements from tuples that are not required.

The approach of Bergmann et al. [9] derives Rete networks from graph patterns by means of a heuristic. These patterns can make use of graph pattern matches that are maintained by Rete networks, which are derived from other patterns. Thus, production nodes in Rete networks of VIATRA2 can have successor nodes that reuse graph pattern matches, in contrast to original Rete networks [32]. Thus, intermediate nodes and production nodes can interleave.

Furthermore, one important issue is that production nodes are used to implement disjunctions of two graph patterns. For two patterns that are combined in terms of a disjunction, each pattern is matched by a separate Rete network. Then, both networks send the resulting tuples to a common production node. Thus, production nodes with multiple ingoing edges implement disjunctions by employing a true union operation.

Moreover, the approach of Bergmann et al. [9] also supports graph patterns that make use of matches of other graph patterns that have cyclic dependencies. Then, cyclic dependencies between the generated Rete networks exist. Note that a single Rete network is still acyclic. These cycles are used to implement recursive graph conditions.

In the approach of Bergmann et al. [9], the change propagation is very similar to the change propagation of original Rete networks [32] and Rete networks for graph grammars [19]. Note that the approach of Bergmann et al. [9] employs graphs that consist of nodes with polymorphic types. Thus, when a graph node of a certain type changes, positive / negative tokens are propagated to input nodes that consider these types and their super types.

The Rete-based incremental pattern matching approach of Bergmann et al. [9] is adapted in multiple other contexts of the MDE domain such as model queries, live model transformations, model synchronization, and derivation of model features. This research finally results in the PhD thesis of Gábor Bergmann [4]. The following approaches employ the Rete-based pattern matching of the VIATRA2.

Bergmann et al. [7] present EMF-IncQuery that enables to store graph pattern matches as query results of model queries over EMF models by means of the Rete-based pattern matching. Then, model queries can be answered instantly, when the Rete network is up-to-date.

Ráth et al. [80] present a live model transformation approach that enables to propagate changes of source models to target models to keep target models consistent with their source models. They use the Rete-based pattern matching to transform local portions of source models into the corresponding local portions of target models.

Ráth et al. [81] employ the Rete-based pattern matching to update derived features of models, when models change. Derived features are attributes and references, which are not explicitly stored in models and can be computed from other model elements. In their approach, derived features can be also derived from other derived features.

Debreceni et al. [27] extend the approach of Ráth et al. [81] by enabling to derive also objects in terms of view models. Their approach enables to derive view models from source models and other view models and ensures that view models are consistent with source models. The authors employ the Rete-based pattern matching to store matches of patterns that describe preconditions of derivation rules, which are used to create view models.

Ghamarian et al. [37] adapt the Rete networks for state space exploration in Groove.

Discussion

In summary, the approaches for incremental graph pattern matching transfer Rete networks from the relational domain to the graph domain. However, these approaches map the challenge of incremental graph pattern matching back to the relational domain by adding additional network node types to the Rete networks such as negative nodes to support the negation of graph nodes. These approaches map graph pattern matches to tuples and employ relational operations such as joins and anti-joins to process these tuples. In contrast to the approach that is presented in this thesis, these approaches do not provide native implementations of Rete networks for incremental graph pattern matching by means of graph operations.

The approaches that base on Rete networks, propagate positive and negative tokens through the network to update the matches that are stored by network nodes. These positive and negative tokens represent creations and deletions of graph nodes and edges, respectively. The approaches that base on Rete networks do not consist of an explicit maintenance step for modified graph nodes and edges. Instead, modifications of graph nodes and edges are mapped to subsequent positive and negative tokens. Thus, these approaches perform no real maintenance of graph pattern matches, because the maintenance process removes these graph pattern matches first and recreates these matches, afterwards. Consequently, the identity of maintained graph pattern matches changes. In contrast, the proposed approach consists of an explicit UPDATE phase for modified graph nodes and edges. Therefore, the proposed approach performs a real maintenance of matches, because the markings of the graph pattern matches keep their identity during maintenance.

The existing approaches for incremental graph pattern matching are limited to Rete network structures. Currently, no approach exists that employs Gator network structures for incremental graph pattern matching. In contrast, the proposed approach enables to employ Gator network structures for incremental graph pattern matching and uses graph transformations instead of relational operations to implement the behavior of network nodes, i. e. view modules. The view modules are generic and are not dedicated to a specific type of condition test as in EMF-IncQuery [9], e. g. negative nodes to implement negations. Instead, the view modules provide an interface to describe the input and output of view modules in a generic manner. This interface enables view modules to encapsulate graph transformations, because only these interfaces are considered by the framework during the view maintenance. The network nodes of Rete networks do not provide such an interface and, thus, are no modules. However, EMF-IncQuery can be adapted to other graph data formats and can be embedded into other query techniques.

The approaches that base on EMF-IncQuery (e. g. live model transformations [80], derivation of model features [81], and synchronization of view models [27]) inherit the disadvantages of EMF-IncQuery. These approaches can also benefit from the advantages of the proposed approach such as Gator network structures that can be more efficient than Rete network structures. The proposed approach can extend and improve all of these approaches and, therefore, has a major impact on the state of the art of incremental graph pattern matching.

Furthermore, Aref et al. [3] and Bergmann et al. [10] aim for a parallelization of the Rete-Matching algorithm. A parallelization of the incremental graph pattern matching is not in the scope of this thesis. Moreover, Varró et al. [92] present a construction approach for Rete networks and Bergmann et al. [5] present a benchmark for Rete-based matching approaches. Both approaches do not support the construction and benchmarking of Gator networks.

12.2.5. Dedicated Approaches for Incremental Activities in MDE

In MDE, several frequent modeling activities involve graph processing tasks. For example, model transformations and model synchronizations are often performed modeling activities to derive and synchronize models. When models are transformed and synchronized, traceability links that describe related model elements must be maintained to trace the impact of model changes throughout model transformation chains. Another example is model constraint evaluation that is employed to check whether models satisfy certain requirements. Both kinds of modeling activities suffer the same problem. They have to be performed efficiently, when developers change models. Otherwise, the whole modeling process is not interactive and hinders effective modeling. The following paragraphs describe techniques for incremental model transformations, model synchronizations, model constraint evaluation, and maintenance of traceability links. These approaches are dedicated to specific problems in MDE.

Model transformations that enable to uni-directionally propagate changes of source models to target models without a complete re-execution of the model transformation and subsequent merges of target models are called live model transformation. Live model transformations do not terminate and continuously keep the transformation context up-to-date. Therefore, changes of source models can be tracked and propagated to derived target models.

Hearnden et al. [51] present an approach for live model transformations using Selective Linear Definite clause (SLD) resolution trees that are tagged to trace model elements in source models concerning their impact on model elements in derived target models. In doing so, the SLD resolution trees represent traces of transformation executions. Adding model elements to source models results in additional tree branches and subtrees that need to be processed to derive corresponding model elements in target models. Deleting model elements results in tree branches that must be pruned to maintain corresponding model elements in target models accordingly. The approach of Hearnden et al. [51] supports changes to source models and model transformations.

Jouault et al. [57] present a live model transformation approach for a subset of the ATLAS Transformation Language (ATL). Their approach enables to propagate changes of source models immediately to derived target models. For that purpose, they present a mechanism that determines parts of model transformation rules that must be re-executed to keep derived target models synchronized. When ATL transformations rules evaluate OCL expressions, the approach tracks which model elements are accessed by these OCL expressions. With this dependency information at hand, model transformation rules can be determined that must be re-executed, when certain model elements change. Then, the approach only executes the model transformation rules that must be re-executed to keep derived target models synchronized.

In contrast to model transformations, model synchronizations deal with the bi-directional propagation of changes, when source and target models of model transformation rules change. Giese et al. [39] describe an approach for incremental model synchronization. Triple Graph Grammar (TGG) rules describe how source and target models are translated in forward and backward direction in a hierarchic manner. Their approach employs a correspondence model that contains correspondence nodes, which track corresponding model elements in source and target models. Moreover, dependency links between correspondence nodes exist that describe the order in which TGG rules are executed. These correspondence nodes and, especially, the dependency links between correspondence nodes enable to synchronize source and target models in forward and backward direction. Information about modified model elements is used to lookup correspondence nodes that are referenced by modified model elements. Then, the algorithm traverses the directed acyclic graph (DAG) of correspondence nodes in a breadth-first search. Their algorithm reverts execution results of TGG rules, when model elements are deleted and applies TGG rules, when model elements are added. Changes of attribute values are propagated between source and target models.

Model constraint evaluation is employed to check whether models satisfy certain constraints, e. g., well-formedness constraints. When models change, such model constraints must be re-evaluated to ensure that these models still satisfy the constraints. However, evaluating model constraints always from scratch is inefficient. Instead, only changed parts of models should be re-evaluated. The following approaches exist.

Egyed [29] describes an approach for consistency checking in UML models. His approach incrementally re-evaluates model constraints, when models change. In this approach, model constraints are considered as black boxes and, therefore, it cannot be determined statically which kinds of model elements are traversed, when evaluating model constraints. The approach consists of two parts. First, the approach employs a model profiler that collects model elements that are traversed, when model constraints are evaluated. This collection of model elements is called scope of a model constraint. Such scopes are used to determine with the help of a lookup table which model constraints must be re-evaluated, when certain model elements change. One advantage of the approach is that model constraints do not have to be annotated for the purpose of incremental re-evaluation, in contrast to other approaches.

Furthermore, developers also want to modify model constraints and get instant feedback on these changes. The approach of Egyed [29] is limited to changeable models and does not support changeable model constraints. Thus, Groher et al. [44] extend the approach of Egyed [29] to support also changeable constraints.

In contrast to the approaches of Egyed [29] and Groher et al. [44], Cabot et al. [21] describe an approach that rewrites model constraints based on structural changes of models to lower the computational complexity of model constraints, when they have to be re-evaluated. Their rewriting approach aims for finding the most incremental expressions that allow an efficient re-evaluation of model constraints. That means, they aim for incremental expressions, which consider the smallest number of model elements, when they are evaluated. However, the approach of Cabot et al. [21] cannot assume black boxes for model constraints, because their approach must be aware of the expression that specifies the model constraints.

Seibel et al. [85] describe an approach for the incremental maintenance of traceability links based on dynamic hierarchical megamodels. Their approach employs maintenance phases in which creation rules and deletion rules are executed to create and delete traceability links depending on the actual model elements that changed. When model elements are created, creation rules are triggered that consider the context in which model elements were created to search for traceability links. When model elements are deleted, deletion rules are triggered that consider the deleted model elements as context were traceability links must be deleted. Modified model elements can lead to the deletion or creation of traceability links.

Discussion

Dedicated approaches for incremental modeling activities in MDE such as model constraint checking a) track the access to model elements (e. g. [29, 57]), b) keep track of dependencies between model elements (e. g.[39]), and c) rewrite model constraints to lower the computational complexity (e. g.[21]), when re-evaluating model constraints. These approaches serve specific purposes and, therefore, are not generally applicable for incremental graph pattern matching. These approaches do not maintain graph pattern matches. However, the proposed approach can extend approaches for model checking and maintenance of traceability links that base on graph pattern matching.

13. Conclusion

This thesis presents a framework for the incremental maintenance of graph views. These views store markings of graph pattern matches. The framework maintains these markings, when graphs change in a way that matches appear or disappear. This thesis refers to this maintenance as incremental graph pattern matching.

The framework is motivated by the fact that existing approaches for incremental graph pattern matching are a) limited to certain graph pattern languages, b) implement the incremental graph pattern matching by means of relational operations, and c) are limited to Rete networks although more generalized discrimination network structures can perform better in time and space at the same time.

Due to this observation, the goal of this thesis is to describe a framework for incremental graph pattern matching that a) enables the effective modeling of graph views (G1 - Modeling Language), b) is independent from employed graph pattern languages (G2 - Embed Graph Queries), and c) enables a native memory- and time-efficient maintenance of graph pattern matches (G3 - Incremental Maintenance). Furthermore, this thesis aims for the realization (G4 - Concept Realization) and evaluation (G5 - Concept Evaluation) of the proposed concepts.

First, this thesis contributes the notion of view graphs that store markings of graph pattern matches (G1a - Notion of Views). Based on this notion, this thesis contributes a modeling language that enables developers to define the content of view graphs by means of graph patterns (G1b - Definition of Views) and enables views to build on the content of other views (G1c - Combination of Views). The modeling language provides view modules that encapsulate graph patterns. Thus, view modules hide the employed graph data models (G2a - Kinds of Graph Models), kinds of employed graph pattern matchings (G2c - Kinds of Pattern Matching), and kinds of employed graph queries (G2b - Kinds of Graph Queries). The presented modeling language support conjunctions, disjunctions, negations, and recursions of graph conditions.

This thesis describes batch and incremental maintenance algorithms for the consistent enumeration of graph pattern matches (G3a - Enumeration of Matches) by means of view graphs. For that purpose, the framework executes the view modules. For the incremental maintenance of the view graphs, the framework captures modifications of base graphs and exploits these modifications to prune the search space of view modules (G3b - Impact Analysis). Then, the view modules employ an efficient local search to find missing, obsolete, and suspicious markings of graph pattern matches in view graphs. For that purpose, the view modules update, delete, and create markings of these matches (G3c - View Maintenance). The maintenance algorithms support conjunctions, disjunctions, negations, and recursions of graph conditions.

As proof of concept, this thesis implements the proposed concepts based on EMF and GMF (G4 - Concept Realization). This thesis uses the prototype for the evaluation of the maintenance algorithms. This thesis provides a twofold evaluation. The application evaluation shows by means of case studies (G5a - Case Studies) that the native realization of view graphs reduces the modeling effort, because graph patterns can exploit the polymorphism of graph nodes in view graphs to effectively support disjunctions and recursions. The performance evaluation shows that the incremental maintenance algorithm outperforms the naive and batch algorithms in time (G5b - Interior Evaluation). Furthermore, the performance evaluation shows that the generalized network structures of Gator networks can perform better in time and space at the same time than Rete network structures for incremental pattern matching, when the proposed maintenance algorithms are employed. Moreover, the performance evaluation shows that the proposed approach can compete with EMF-IncQuery (G5c - Exterior Evaluation).

Furthermore, this thesis enhances the proposed marking mechanism and maintenance algorithms to a) remove redundant markings of matches (OG1 - Space-Efficiency) without losing the capability to enumerate all matches (OG3 - Enumerate Duplicates) and to b) efficiently investigate the search spaces that are common to multiple redundant markings of matches (OG2 - Time-Efficiency). For that purpose, this thesis extends annotations with aggregations, which enable to aggregate matches that result in redundant markings. This aggregation enables to reduce the memory consumption of view graphs and the cost of the graph pattern matching that is imposed by redundant markings of matches.

This thesis has a major impact on the state of the art of incremental graph pattern matching, because this thesis employs Gator networks with generalized discrimination network structures and, therefore, overcomes the limitations of existing approaches that base on Rete networks. By employing Gator networks for incremental graph pattern matching, this thesis enables to steer the trade-off between memory consumption to store the network state and execution time to update the network state. Gator network structures also enable to constitute equivalent Rete network structures. Therefore, this thesis extends Rete-based to Gator-based incremental graph pattern matching. The proposed framework is the first approach that employs Gator networks for incremental graph pattern matching.

This thesis supports incremental graph pattern matching for arbitrary kinds of graphs and graph pattern matching technologies due to the encapsulation of graph queries by means of view modules. Therefore, the proposed approach constitutes a framework for incremental graph pattern matching according to the notion of a framework (cf. Section 1.2).

Additionally, this thesis describes maintenance algorithms that employ an explicit maintenance phase for modified graph nodes and edges to implement a *real* view graph maintenance, which ensures that maintained graph pattern matches keep their identity until they finally disappear from graphs. This explicit maintenance phase is not present in the Rete algorithm for incremental graph pattern matching. Thus, the existing approaches do not perform a *real* maintenance of graph pattern matches, because they process modified graph nodes and edges by means of a deletion and re-creation of graph pattern matches.

Furthermore, incremental graph pattern matching has many application domains such as model queries, model checking, model transformations, model synchronization, or graph database views and has a large range of graph domains such as social networks, models in MDE, or biological and chemical compounds. Thus, the proposed framework enables an enhanced incremental pattern matching for many different graph domains. Thus, this thesis has a major impact on the state of the art of incremental graph pattern matching.

This thesis enables several directions of future work. One direction may investigate the *parallel* incremental processing of graph changes to increase the performance of the view maintenance, when the number of graph changes becomes large. Another direction may investigate the optimality of the employed Gator network structure from the perspective of different parameters such as update frequency of graphs and the size as well as structure of graph patterns that are encapsulated by view modules. Based on this direction, another research branch may investigate the optimality of Gator network structures, when graphs change, and adapt the network structure accordingly by means of operations that efficiently transform the network structure into a more efficient network structure.

Bibliography

[1] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. "MOVIE: An incremental maintenance system for materialized object views". In: *Data & Knowledge Engineering* 47.2 (2003), pp. 131–166. DOI: `10.1016/S0169-023X(03)00048-X`.

[2] Renzo Angles. "A Comparison of Current Graph Database Models". In: *Proceedings of the 28th International Conference on Data Engineering*. IEEE, Apr. 2012, pp. 171–177. DOI: `10.1109/ICDEW.2012.31`.

[3] Mostafa M. Aref and Mohammed A. Tayyib. "Lana–Match algorithm: a parallel version of the Rete–Match algorithm". In: *Parallel Computing* 24.5 (1998), pp. 763–775. DOI: `10.1016/S0167-8191(98)00003-9`.

[4] Gábor Bergmann. "Incremental Model Queries in Model-Driven Design". PhD thesis. Budapest University of Technology and Economics, 2013.

[5] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. "A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation". In: *Graph Transformations*. Springer, 2008, pp. 396–410. DOI: `10.1007/978-3-540-87405-8_27`.

[6] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. "Efficient Model Transformations by Combining Pattern Matching Strategies". In: *Theory and Practice of Model Transformations*. Ed. by Richard F. Paige. Vol. 5563. LNCS. Springer, 2009, pp. 20–34. ISBN: 978-3-642-02407-8. DOI: `10.1007/978-3-642-02408-5_3`.

[7] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. "Incremental Evaluation of Model Queries over EMF Models". In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. LNCS. Springer, 2010, pp. 76–90. ISBN: 978-3-642-16144-5. DOI: `10.1007/978-3-642-16145-2_6`.

[8] Gábor Bergmann, Dóra Horváth, and Ákos Horváth. "Applying Incremental Graph Transformation to Existing Models in Relational Databases". In: *Sixth International Conference on Graph Transformation*. Springer, Sept. 2012, pp. 371–385. DOI: `10.1007/978-3-642-33654-6_25`.

[9] 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. ACM, 2008, pp. 25–32. ISBN: 978-1-60558-033-3. DOI: `10.1145/1402947.1402953`.

[10] Gábor Bergmann, István Ráth, and Dániel Varró. "Parallelization of Graph Transformation Based on Incremental Pattern Matching". In: *Electronic Communications of the EASST* 18.0 (2009). DOI: `10.14279/tuj.eceasst.18.265.249`.

[11] Thomas Beyhl, Gregor Berg, and Holger Giese. “Connecting Designing and Engineering Activities”. In: *Design Thinking Research - Building Innovation Eco-Systems.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2013, pp. 153–182. ISBN: 978-3-319-01303-9. DOI: 10.1007/978-3-319-01303-9_11.

[12] Thomas Beyhl, Gregor Berg, and Holger Giese. “Why Innovation Processes Need to Support Traceability”. In: *Proceedings of 7th International Workshop on Traceability in Emerging Forms of Software Engineering.* IEEE, 2013, pp. 1–4. DOI: 10.1109/TEFSE.2013.6620146.

[13] Thomas Beyhl and Holger Giese. “Connecting Designing and Engineering Activities II”. In: *Design Thinking Research - Building Innovators.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2015, pp. 211–239. ISBN: 978-3-319-06823-7. DOI: 10.1007/978-3-319-06823-7_12.

[14] Thomas Beyhl and Holger Giese. “Connecting Designing and Engineering Activities III”. In: *Design Thinking Research - Making Design Thinking Foundational.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2016, pp. 265–290. ISBN: 978-3-319-19641-1. DOI: 10.1007/978-3-319-19641-1_16.

[15] Thomas Beyhl and Holger Giese. “The Design Thinking Methodology at Work: Capturing and Understanding the Interplay of Methods and Techniques”. In: *Design Thinking Research - Taking Breakthrough Innovation Home.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2016. ISBN: 978-3-319-40381-6. DOI: 10.1007/978-3-319-40382-3_5.

[16] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. “Formal foundation of consistent EMF model transformations by algebraic graph transformation”. In: *Software & Systems Modeling* 11.2 (2012), pp. 227–250. DOI: 10.1007/s10270-011-0199-7.

[17] 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. ACM, 1986, pp. 61–71. DOI: 10.1145/16894.16861.

[18] Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. “Searching Repositories of Web Application Models”. In: *Web Engineering.* Springer, 2010. DOI: 10.1007/978-3-642-13911-6_1.

[19] H. Bunke, T. Glauser, and T.-H. Tran. “An efficient implementation of graph grammars based on the RETE matching algorithm”. In: *Graph Grammars and Their Application to Computer Science.* Ed. by Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 532. LNCS. Springer, 1991, pp. 174–189. DOI: 10.1007/BFb0017389.

[20] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. “Local Search-Based Pattern Matching Features in EMF-IncQuery”. In: *Graph Transformation: 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 21-23, 2015. Proceedings.* Ed. by Francesco Parisi-Presicce and Bernhard Westfechtel. Springer, 2015, pp. 275–282. ISBN: 978-3-319-21145-9. DOI: 10.1007/978-3-319-21145-9_18.

[21] Jordi Cabot and Ernest Teniente. “Incremental Evaluation of OCL Constraints”. In: *Advanced Information Systems Engineering.* Vol. 4001. LNCS. Springer, 2006, pp. 81–95. DOI: 10.1007/11767138_7.

[22] 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. Morgan Kaufmann Publishers Inc., 1991, pp. 577–589. ISBN: 1-55860-150-3.

[23] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. "APEX: An Adaptive Path Index for XML Data". In: *Proceedings of the International Conference on Management of Data*. SIGMOD '02. ACM, 2002, pp. 121–132. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564706.

[24] 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*. ACM, 1996, pp. 469–480. DOI: 10.1145/233269.233364.

[25] 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. ACM, 1997, pp. 405–416. DOI: 10.1145/253260.253353.

[26] 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. Morgan Kaufmann Publishers Inc., 2001, pp. 341–350. ISBN: 1-55860-804-4.

[27] 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. ACM, 2014, pp. 31–38. ISBN: 978-1-4503-2900-2. DOI: 10.1145/2631675.2631677.

[28] VIATRA Developers. *Website: https://wiki.eclipse.org/VIATRA/Query/UserDocumentation/API/LocalSearch (last access: 15 July 2017)*. 2017.

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

[30] 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*. ACM, 2011, pp. 925–936. DOI: 10.1145/1989323.1989420.

[31] Wenfei Fan, Xin Wang, and Yinghui Wu. "Answering Graph Pattern Queries Using Views". In: *Proceedings of 30th International Conference on Data Engineering*. IEEE, Mar. 2014, pp. 184–195. DOI: 10.1109/ICDE.2014.6816650.

[32] Charles L. Forgy. "Rete: A Fast Algorithm for the Many Pattern/Many object Pattern Match Problem". In: *Artificial Intelligence* 19.1 (1982), pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0.

[33] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7.

[34] Inc. Franz. *Website: http://allegrograph.com/allegrograph/ (last access: 14 March 2016)*. 2016.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 978-0-201-63361-0.

[36] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Company, 1979. ISBN: 978-0716710455.

[37] Amir Hossein Ghamarian, Arash Jalali, and Arend Rensink. "Incremental Pattern Matching in Graph-Based State Space Exploration". In: *Proceedings of the* 4th *International Workshop on Graph-Based Tools*. Electronic Communications of the EASST, 2010. DOI: `10.14279/tuj.eceasst.32.520`.

[38] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. "Graph Transformations for MDE, Adaptation, and Models at Runtime". In: *Formal Methods for Model-Driven Engineering*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Vol. 7320. LNCS. Springer, June 2012, pp. 137–191. DOI: `10.1007/978-3-642-30982-3_5`.

[39] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars". In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio. Vol. 4199. LNCS. Springer, Oct. 2006, pp. 543–557. DOI: `10.1007/11880240_38`.

[40] Rosalba Giugno and Dennis Shasha. "GraphGrep: A fast and universal method for querying graphs". In: *Proceedings of the 16th International Conference on Pattern Recognition*. Vol. 2. IEEE, 2002, pp. 112–115. DOI: `10.1109/ICPR.2002.1048250`.

[41] Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Fernando Orejas. "Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs". In: *Theoretical Computer Science* 424.1 (2012), pp. 46–68. DOI: `10.1016/j.tcs.2012.01.032`.

[42] 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. Morgan Kaufmann Publishers Inc., 1997, pp. 436–445. ISBN: 1-55860-470-7.

[43] 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". In: *AI Communications* 17.1 (Jan. 2004), pp. 13–23.

[44] Iris Groher, Alexander Reder, and Alexander Egyed. "Incremental Consistency Checking of Dynamic Constraints". In: *Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 203–217. DOI: `10.1007/978-3-642-12029-9_15`.

[45] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In: *Data Engineering* 18.2 (1995), pp. 3–18.

[46] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally". In: *Proceedings of the International Conference on Management of Data*. SIGMOD '93. ACM, 1993, pp. 157–166. ISBN: 0-89791-592-5. DOI: `10.1145/170035.170066`.

[47] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. "Graph Grammars with Negative Application Conditions". In: *Fundamenta Informaticae* 26.3 (Dec. 1996), pp. 287–313.

[48] Eric N. Hanson, Sreenath Bodagala, and Ullas Chadaga. “Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks”. In: *Transactions on Knowledge and Data Engineering* 14.2 (Mar. 2002), pp. 261–280. DOI: `10.1109/69.991716`.

[49] 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, 1992, pp. 56–65.

[50] 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. IEEE, 2006, pp. 38–50. ISBN: 0-7695-2570-9. DOI: `10.1109/ICDE.2006.37`.

[51] 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.* Springer, 2006, pp. 321–335. DOI: `10.1007/11880240_23`.

[52] Regina Hebig, Andreas Seibel, and Holger Giese. “On the Unification of Megamodels”. In: *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling.* Vol. 42. Electronic Communications of the EASST, 2011. DOI: `10.14279/tuj.eceasst.42.704`.

[53] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. *JaMoPP: The Java Model Parser and Printer.* Tech. rep. TUD-FI09-10. TU Dresden, 2009.

[54] Michael Hunger. *Neo4j 2.0 Eine Graphdatenbank für alle.* entwickler.press, 2014. ISBN: 9783-86802-315-2.

[55] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. “IncQuery-D: Incremental Graph Search in the Cloud”. In: *Proceedings of the Workshop on Scalability in Model Driven Engineering.* BigMDE ’13. ACM, 2013, pp. 1–4. ISBN: 978-1-4503-2165-5. DOI: `10.1145/2487766.2487772`.

[56] Ralph E. Johnson and Brian Foote. “Designing reusable classes”. In: *Journal of Object-Oriented Programming* 1.2 (1988), pp. 22–35.

[57] 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. Springer, 2010, pp. 123–137. ISBN: 978-3-642-13687-0. DOI: `10.1007/978-3-642-13688-7_9`.

[58] Gabor Karsai. “Lessons Learned from Building a Graph Transformation System”. In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday.* Ed. by Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel. Springer, 2010, pp. 202–223. ISBN: 978-3-642-17322-6. DOI: `10.1007/978-3-642-17322-6_10`.

[59] 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.* IEEE, Mar. 2002, pp. 129–140. DOI: `10.1109/ICDE.2002.994703`.

[60] U. Khurana and A. Deshpande. “Efficient Snapshot Retrieval over Historical Graph Data”. In: *Proceedings of the 29th International Conference on Data Engineering.* IEEE, Apr. 2013, pp. 997–1008. DOI: `10.1109/ICDE.2013.6544892`.

[61] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. “GRAS, a Graph Oriented (Software) Engineering Database System”. In: *Information Systems* 20.1 (Mar. 1995), pp. 21–51. DOI: 10.1016/0306-4379(95)00002-L.

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

[63] 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.* Springer, 2012, pp. 180–200. DOI: 10.1007/978-3-642-28830-2_10.

[64] Christian Krause, Matthias Tichy, and Holger Giese. “Implementing Graph Transformations in the Bulk Synchronous Parallel Model”. In: *Fundamental Approaches to Software Engineering.* Springer, 2014, pp. 325–339. DOI: 10.1007/978-3-642-54804-8_23.

[65] Harumi A. Kuno and Elke A. Rundensteiner. “Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation”. In: *Transactions on Knowledge and Data Engineering* 10.5 (Sept. 1998), pp. 768–792. DOI: 10.1109/69.729731.

[66] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. “Attributed graph transformation with node type inheritance”. In: *Theoretical Computer Science* 376.3 (2007). Fundamental Aspects of Software Engineering, pp. 139–163. DOI: 10.1016/j.tcs.2007.02.001.

[67] Ho Soo Lee and Marshall I. Schor. “Match Algorithms for Generalized Rete Networks”. In: *Artificial Intelligence* 54.2 (1992), pp. 249–274. DOI: 10.1016/0004-3702(92)90047-2.

[68] Jixue Liu, Millist Vincent, and Mukesh Mohania. “Maintaining Views in Object-Relational Databases”. In: *Proceedings of the 9th International Conference on Information and Knowledge Management.* CIKM ’00. ACM, 2000, pp. 102–109. ISBN: 1-58113-320-0. DOI: 10.1145/354756.354807.

[69] Daniel Lucrédio, Renata Fortes, and Jon Whittle. “MOOGLE: a metamodel-based model search engine”. In: *Software & Systems Modeling* 11.2 (2010), pp. 183–208. DOI: 10.1007/s10270-010-0167-7.

[70] Tom Mens and Tom Tourwé. “A Survey of Software Refactoring”. In: *IEEE Transactions on Software Engineeing* 30.2 (Feb. 2004), pp. 126–139. DOI: 10.1109/TSE.2004.1265817.

[71] B. T. Messmer and H. Bunke. “A decision tree approach to graph and subgraph isomorphism detection”. In: *Pattern Recognition* 32.12 (1999), pp. 1979–1998. DOI: 10.1016/S0031-3203(98)90142-X.

[72] Tova Milo and Dan Suciu. “Index Structures for Path Expressions”. In: *Proceedings of the 7th International Conference on Database Theory.* ICDT ’99. Springer, 1999, pp. 277–295. ISBN: 3-540-65452-6.

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

[74] 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. ACM, 2001, pp. 307–318. ISBN: 1-58113-332-4. DOI: `10.1145/375663.375703`.

[75] Jörg Niere. “Inkrementelle Entwurfsmustererkennung”. PhD thesis. Universität Paderborn, 2004.

[76] Jörg Niere, Lothar Wendehals, and Albert Zündorf. *An interactive and scalable approach to design pattern recovery.* Tech. rep. tr-ri-03-236. University of Paderborn, 2003.

[77] Object Management Group. *Object Constraint Language 2.4.* 2014.

[78] Xiaolei Qian and Gio Wiederhold. “Incremental Recomputation of Active Relational Expressions”. In: *Transactions on Knowledge and Data Engineering* 3.3 (Sept. 1991), pp. 337–341. DOI: `10.1109/69.91063`.

[79] Ghulam Rasool and Detlef Streitfdert. “A Survey on Design Pattern Recovery Techniques”. In: *International Journal of Computer Science Issues* 8.2 (2011).

[80] 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.* Springer, 2008, pp. 107–121. DOI: `10.1007/978-3-540-69927-9_8`.

[81] 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. Springer, 2012, pp. 102–117. ISBN: 978-3-642-31490-2. DOI: `10.1007/978-3-642-31491-9_10`.

[82] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases (Second Edition).* O’Reilly Media, 2015. ISBN: 978-1-491-93200-1.

[83] Marko A. Rodriguez and Peter Neubauer. “The Graph Traversal Pattern”. In: *CoRR Journal* 1004.1001 (2010).

[84] 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. ACM, 1996, pp. 447–458. ISBN: 0-89791-794-4. DOI: `10.1145/233269.233361`.

[85] Andreas Seibel, Stefan Neumann, and Holger Giese. “Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance”. In: *Software & Systems Modeling* 9.4 (2010), pp. 493–528. DOI: `10.1007/s10270-009-0146-z`.

[86] Oded Shmueli and Alon Itai. “Maintenance of Views”. In: *Proceedings of the International Conference on Management of Data.* SIGMOD ’84. ACM, 1984, pp. 240–255. ISBN: 0-89791-128-8. DOI: `10.1145/602259.602293`.

[87] 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.

[88] Srinath Srinivasa and M. Harjinder Singh. “Grace: A Graph Database System”. In: *Proceedings of the* 12*th International Conference on Management of Data.* Computer Society of India, Dec. 2005.

[89] Szabó Tamás. *Website: https://viatra.net/news/2014/11/mbeddr-meets-incquery-combining-best-features-two-modeling-worlds-eclipsecon-europe-201 (last access: 15 July 2017).* 2014.

[90] The Neo4j Team. *The Neo4j Manual v3.0.* Specification at `http://neo4j.com/docs/` (last access: July 8th 2016).

[91] Antoine Toulmé. "Presentation of EMF compare utility". In: *Eclipse Modeling Symposium.* 2006, pp. 1–8.

[92] Gergely Varró and Frederik Deckwerth. "A Rete Network Construction Algorithm for Incremental Pattern Matching". In: *Theory and Practice of Model Transformations.* Ed. by Keith Duddy and Gerti Kappel. Vol. 7909. LNCS. Springer, 2013, pp. 125–140. ISBN: 978-3-642-38882-8. DOI: `10.1007/978-3-642-38883-5_13`.

[93] Gergely Varró, Katalin Friedl, and Dániel Varró. "Graph Transformation in Relational Databases". In: *Electronic Notes in Theoretical Computer Science* 127.1 (2005), pp. 167–180. DOI: `10.1016/j.entcs.2004.12.034`.

[94] Gergely Varró and Dániel Varró. "Graph Transformation with Incremental Updates". In: *Electron. Notes Theor. Comput. Sci.* 109 (Dec. 2004), pp. 71–83. DOI: `10.1016/j.entcs.2004.02.057`.

[95] D.W. Williams, Jun Huan, and Wei Wang. "Graph Database Indexing Using Structured Graph Decomposition". In: *Proceedings of the* 23rd *International Conference on Data Engineering.* IEEE, Apr. 2007, pp. 976–985. DOI: `10.1109/ICDE.2007.368956`.

[96] Ian Wright and James A. R. Marshall. "The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case". In: *International Journal of Intelligent Games & Simulation* 2.1 (2003), pp. 36–48.

[97] 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. ACM, 2004, pp. 335–346. ISBN: 1-58113-859-8. DOI: `10.1145/1007568.1007607`.

[98] Xifeng Yan, Philip S. Yu, and Jiawei Han. "Graph Indexing Based on Discriminative Frequent Structure Analysis". In: *Transactions on Database Systems* 30.4 (Dec. 2005), pp. 960–993. DOI: `10.1145/1114244.1114248`.

[99] 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. ACM, 2005, pp. 766–777. ISBN: 1-59593-060-4. DOI: `10.1145/1066157.1066244`.

[100] Shijie Zhang, Meng Hu, and Jiong Yang. "TreePi: A Novel Graph Indexing Method". In: *Proceedings of the 23rd International Conference on Data Engineering.* IEEE, Apr. 2007, pp. 966–975. DOI: `10.1109/ICDE.2007.368955`.

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

[102] Albert Zündorf. "Graph pattern matching in PROGRES". In: *Graph Grammars and Their Application to Computer Science.* Ed. by Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg. Springer, 1996, pp. 454–468. ISBN: 978-3-540-68388-9. DOI: `10.1007/3-540-61228-9_105`.

Author's Contributions

Book Chapters

[C1] Thomas Beyhl, Gregor Berg, and Holger Giese. "Connecting Designing and Engineering Activities". In: *Design Thinking Research - Building Innovation Eco-Systems.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2013, pp. 153–182. ISBN: 978-3-319-01303-9. DOI: `10.1007/978-3-319-01303-9_11`.

[C2] Thomas Beyhl and Holger Giese. "Connecting Designing and Engineering Activities II". In: *Design Thinking Research - Building Innovators.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2015, pp. 211–239. ISBN: 978-3-319-06823-7. DOI: `10.1007/978-3-319-06823-7_12`.

[C3] Thomas Beyhl and Holger Giese. "Connecting Designing and Engineering Activities III". In: *Design Thinking Research - Making Design Thinking Foundational.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2016, pp. 265–290. ISBN: 978-3-319-19641-1. DOI: `10.1007/978-3-319-19641-1_16`.

[C4] Thomas Beyhl and Holger Giese. "The Design Thinking Methodology at Work: Capturing and Understanding the Interplay of Methods and Techniques". In: *Design Thinking Research - Taking Breakthrough Innovation Home.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2016. ISBN: 978-3-319-40381-6. DOI: `10.1007/978-3-319-40382-3_5`.

[C5] Gregor Gabrysiak, Holger Giese, and Thomas Beyhl. "Virtual Multi-User Software Prototypes III". In: *Design Thinking Research - Measuring Performance in Context.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Understanding Innovation. Springer, 2012, pp. 263–284. ISBN: 978-3-642-31990-7. DOI: `10.1007/978-3-642-31991-4_15`.

Conference and Workshop Proceedings

[P1] Thomas Beyhl, Gregor Berg, and Holger Giese. "Towards Documentation Support for Educational Design Thinking Projects". In: *International Conference on Engineering and Product Design Education.* Design Society, 2013, pp. 408–413. ISBN: 978-1-904670-42-1.

[P2] Thomas Beyhl, Gregor Berg, and Holger Giese. "Why Innovation Processes Need to Support Traceability". In: *Proceedings of 7^{th} International Workshop on Traceability in Emerging Forms of Software Engineering.* IEEE, 2013, pp. 1–4. DOI: `10.1109/TEFSE.2013.6620146`.

[P3] Thomas Beyhl, Dominique Blouin, Holger Giese, and Leen Lambers. “On the Operationalization of Graph Queries with Generalized Discrimination Networks”. In: *Proceedings of the 9th International Conference on Graph Transformations*. Ed. by Rachid Echahed and Mark Minas. Springer, 2016, pp. 170–186. ISBN: 978-3-319-40530-8. DOI: `10.1007/978-3-319-40530-8_11`.

[P4] Thomas Beyhl and Holger Giese. “Incremental View Maintenance for Deductive Graph Databases using Generalized Discrimination Networks”. In: *Proceedings of Graphs as Models Workshop 2016*. Ed. by Alexander Heußner, Aleks Kissinger, and Anton Wijs. Vol. 231. Electronic Proceedings in Theoretical Computer Science 5. Open Publishing Association, Dec. 2016, pp. 57–71. DOI: `10.4204/EPTCS.231.5`.

[P5] Thomas Beyhl and Holger Giese. “Traceability Recovery for Innovation Processes”. In: *Proceedings of the 8th International Symposium on Software and Systems Traceability*. IEEE, 2015, pp. 22–28. DOI: `10.1109/SST.2015.11`.

[P6] Thomas Beyhl, Regina Hebig, and Holger Giese. “A Model Management Framework for Maintaining Traceability Links”. In: *Software Engineering 2013 Workshopband*. Software Engineering 2013 - Workshopband, LNI 215, 2013, pp. 453–457. ISBN: 978-3-88579-609-1.

[P7] Axel Menning, Thomas Beyhl, Holger Giese, Ulrich Weinberg, and Claudia Nicolai. “Introducing the LogCal: Template-Based Documentation Support for Educational Design Thinking Projects”. In: *Proceedings of the 16th International Conference on Engineering and Product Design*. Design Society, 2014, pp. 68–73. ISBN: 978-1-904670-56-8.

Technical Reports

[R1] Thomas Beyhl and Holger Giese. *Efficient and Scalable Graph View Maintenance for Deductive Graph Databases based on Generalized Discrimination Networks*. Tech. rep. Technical Report No. 99. Hasso Plattner Institute at the University of Potsdam, 2015. URL: `http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-79535`.

List of Figures

List of Tables

List of Abbreviations

API	application programming interface
ASG	abstract syntax graph
ATL	ATLAS Transformation Language
CRUD	create, read, update, and delete
DAG	directed acyclic graph
DPO	double-pushout
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
JaMoPP	Java Model Parser and Printer
MDE	Model-Driven Engineering
MPS	Meta Programming System
NAC	negative application condition
OCL	Object Constraint Language
OLAP	online analytical processing
OLTP	online transactional processing
OQL	Object Query Language
PAC	positive application condition
QR-SQL	Object Relational SQL
SLD	Selective Linear Definite clause
SPO	single-pushout
TGG	Triple Graph Grammar
UML	Unified Modeling Language
XMI	XML Metadata Interchange

Appendix

A. Preliminaries

The following definitions are adapted from [16, 38, 41, 66].

Definition 18 describes the notion of E-graphs. E-graphs extend graphs by attributes for graph nodes and edges. These attributes store additional data values of entities and relationships. E-graphs represent attributes as data nodes that are connected to graph nodes and edges by means of node attribute and edge attribute edges, respectively. Note that graph nodes and edges can have multiple attributes.

Definition 18 (E-Graph)
An E-Graph $G = (N_G, N_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j\in\{G,NA,EA\}})$ consists of the sets

- *N_G and N_D called graph nodes and data nodes, respectively, and*
- *E_G, E_{NA}, E_{EA} called graph edges, node attribute edges, and edge attribute edges,*

and the source and target functions

- *$s_G : E_G \to N_G$ and $t_G : E_G \to N_G$ for graph edges,*
- *$s_{NA} : E_{NA} \to N_G$ and $t_{NA} : E_{NA} \to N_D$ for node attribute edges, and*
- *$s_{EA} : E_{EA} \to E_G$ and $t_{EA} : E_{EA} \to N_D$ for edge attribute edges.*

Accordingly, Definition 19 extends the notion of graph morphism in a manner that a source graph is embedded in a target graph, if the structure-preserving mapping also preserves the structure of data nodes, node attributes, and edge attributes as well.

Definition 19 (E-Graph Morphism)
Given two E-graphs

- *$G = (N_G^G, N_D^G, E_G^G, E_{NA}^G, E_{EA}^G, (s_j^G, t_j^G)_{j\in\{G,NA,EA\}})$ and*
- *$H = (N_G^H, N_D^H, E_G^H, E_{NA}^H, E_{EA}^H, (s_j^H, t_j^H)_{j\in\{G,NA,EA\}})$,*

an E-graph morphism $f : G \to H$ is a tuple $(f_{N_G}, f_{N_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with

- *$f_{N_i} : N_i^G \to N_i^H$ for $i \in \{G, D\}$ and*
- *$f_{E_j} : E_j^G \to E_j^H$ for $j \in \{G, NA, EA\}$*

such that f commutes with all source and target functions.

An attributed graph is an E-graph that additionally consists of an algebra over a data signature, which is defined by attribute value sorts and data operations for these sorts.

Definition 20 (Attributed Graph)
Given a data signature $DSIG = (S_D, OP_D)$ with attribute value sorts $S_D^{'} \subseteq S_D$, an attributed graph $AG = (G, D)$ consists of an E-graph G together with a $DSIG$-algebra D such that $\uplus_{s\in S_D^{'}} D_s = N_D$.

Definition 21 extends the notion of E-graph morphism for attributed graphs in a manner that also the mapping of the source and target algebra preserves their algebraic structure.

Definition 21 (Attributed Graph Morphism)
Given two attributed graphs $AG^G = (G^G, D^G)$ and $AG^H = (G^H, D^H)$, an attributed graph morphism $f : AG^G \to AG^H$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^G \to G^H$ and an algebra homomorphism $f_D : D^G \to D^H$ such that $f_{G,N_D}(x) = f_{D,s}(x)$ for all $x \in D_{G,s}, s \in S'_D$.

According to Definition 22, an attributed type graph is an attributed graph with an algebra. Attributed type graphs are reference graphs that describe kinds of graph nodes and edges as well as kinds of attributes that can be owned by these kinds of graph nodes and edges.

Definition 22 (Attributed Type Graph)
Given a data signature DSIG, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final DSIG-algebra.

According to Definition 23, typed attributed graphs over attributed type graphs are instance graphs that instantiate the kinds of graph nodes and edges as well as kinds of attributes that are defined by these attributed type graphs. Then, graph nodes and edges of typed attributed graphs have concrete types and their attributes have concrete data values.

Definition 23 (Typed Attributed Graph)
A typed attributed graph (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.

Definition 24 extends the notion of attributed graph morphism in a manner that also the types of graph nodes and edges are considered by the structure-preserving mapping between the source graph and target graph.

Definition 24 (Typed Attributed Graph Morphism)
A typed attributed graph morphism $f : (AG^G, t^G) \to (AG^H, t^H)$ is an attributed graph morphism $f : AG^G \to AG^H$ such that $t^H \circ f = t^G$.

Definition 25 extends attributed type graphs by an inheritance graph that describes a type hierarchy. This type hierarchy is used to express polymorphy of graph nodes by means of inheritance relations between graph nodes of the inheritance graph.

Definition 25 (Attributed Type Graph with Inheritance)
An attributed type graph with inheritance $ATGI = (ATG, I, A)$ consists of an attributed type graph ATG, an inheritance graph $I = (N_I, E_I, s_I, t_I)$ with $N_I = ATG_{N_G}$, and a set $A \subseteq N_I$ of abstract nodes. Moreover, I must be acyclic and the property $\forall e, e' \in E_I : s_I(e) = s_I(e') \Leftrightarrow e = e'$ must hold.

According to Definition 26, the inheritance clan of a node contains all nodes that inherit the node attributes and graph edges of the node.

Definition 26 (Inheritance Clan)
Given an attributed type graph with inheritance $ATGI = (ATG, I, A)$ with an inheritance graph $I = (N_I, E_I, s_I, t_I)$, an inheritance clan $clan_I(n) = \{m | m \in N_I \wedge \exists \text{ path } m \xrightarrow{*} n \in I\} \subseteq N_I$ with $n \in clan_I(n)$.

B. View Graph Transformation Language

The default view graph transformation language enables developers of the framework to model the left-hand side and right-hand side of view graph transformation rules. A view graph transformation rule is a transformation rule (cf. Definition 10) that manipulates view graphs to store graph pattern matches in terms of annotations and maintain these annotations. The left-hand side of a view graph transformation rule describes a graph pattern for which the framework has to maintain graph pattern matches. The right-hand side of a view graph transformation rule describes the side-effect that is applied to a view graph when a) a match is found for the graph pattern and must be stored by view graphs, b) a match is dissatisfied and must be deleted from view graphs, and c) a match must be revised to check whether it still satisfies the graph pattern. Note that view graph transformation rules apply side-effects to view graphs only. View graph transformation rules do not manipulate base graphs.

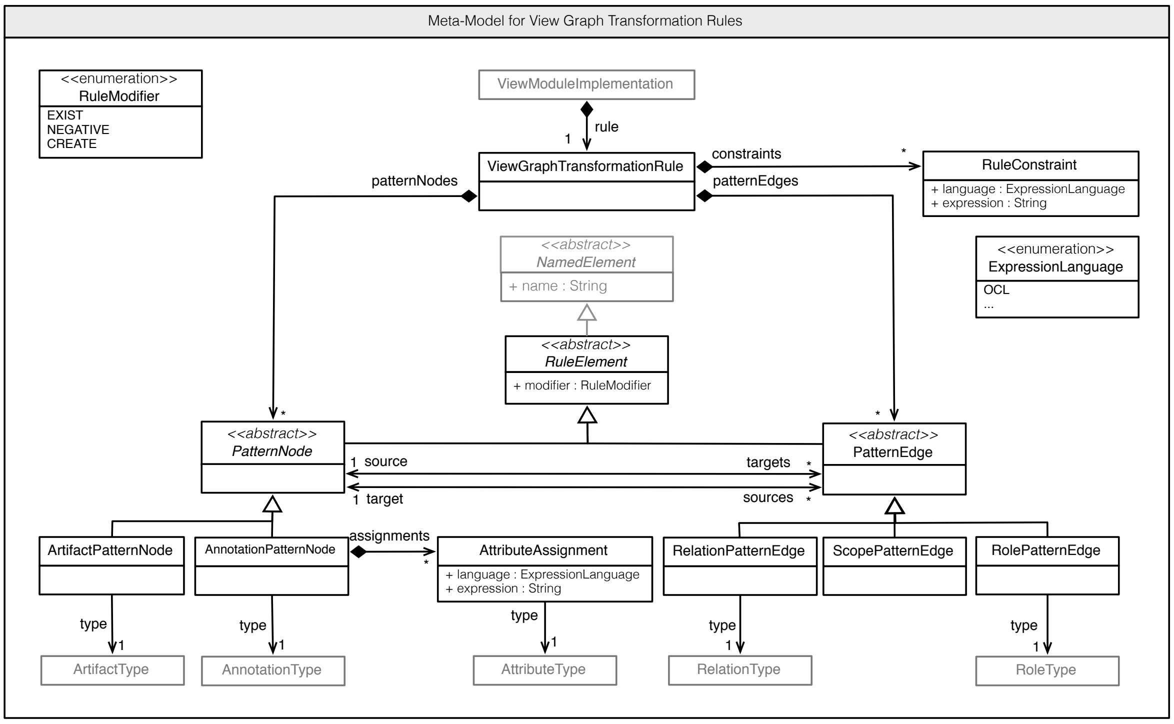


Figure B.1.: Metamodel for view graph transformations

Metamodel

Figure B.1 depicts the metamodel of the view graph transformation language. The metamodel depicts parts of the metamodel that are already introduced in gray color. The metamodel shows the ViewModuleImplementation class that owns a rule composition that targets at the ViewGraphTransformationRule class to describe that each kind of view module implementation consists of a view graph transformation rule. That means, a CreateImplementation, DeleteImplementation, and UpdateImplementation consists of a view graph transformation rule. The ViewGraphTransformationRule class owns the patternNodes and patternEdges compositions that target at the abstract PatternNode and PatternEdge class, respectively. Therefore, view graph transformation rules consist of pattern nodes and edges that together describe the left-hand side and right-hand side of view graph transformation rules. The PatternNode and PatternEdge class represents pattern nodes and edges (cf. Definition 7), respectively.

The PatternNode and PatternEdge classes are specializations of the abstract RuleElement class. The abstract RuleElement class is a specialization of the NamedElement class and, therefore, inherits the name attribute. Additionally, the RuleElement class consists of the modifier attribute with RuleModifier type and the binding attribute with RuleBinding type.

The RuleModifier enumeration lists all kinds of modifiers, i. e., Exist, Negative, and Create. The Exist modifier describes that a rule element belongs to the left-hand side of the view graph transformation rule and a graph node or edge that matches the rule element must exist to satisfy the left-hand side of the view graph transformation rule. The Negative modifier describes that a rule element belongs to the left-hand side of the view graph transformation rule and a graph node or edge that matches the rule element must *not* exist to satisfy the left-hand side of the view graph transformation rule. The Create modifier describes that a rule element belongs to the right-hand side of the view graph transformation rule and the view graph transformation rule creates graph nodes or edges in a view graph, when the left-hand side of the view graph transformation rule is satisfied.

The metamodel distinguishes the ArtifactPatternNode class and AnnotationPatternNode class as specializations of the PatternNode class. The ArtifactPatternNode class represents graph pattern nodes that match certain kinds of artifacts of base graphs. The AnnotationPatternNode class represents graph pattern nodes that match certain kinds of annotations of view graphs. The ArtifactPatternNode and AnnotationPatternNode class consist of type references that target at the ArtifactType and AnnotationType class to describe the kinds of these artifacts and annotations, respectively.

The PatternEdge class owns source and target associations that target at the PatternNode class. The source and target associations describe the direction of pattern edges. Furthermore, the metamodel distinguishes the RelationPatternEdge class, RolePatternEdge and ScopePatternEdge classes as specializations of the PatternEdge class. The RelationPatternEdge class represents graph pattern edges that match certain kinds of relations in base graphs. The RolePatternEdge class represents graph pattern edges that match certain kinds of roles in view graphs. The ScopePatternEdge class represent graph pattern edges that match scopes in view graphs. The RelationPatternEdge and RolePatternEdge class consist of type references that target at the RelationType and RoleType class, respectively, to describe the kinds of these relations and roles, respectively. Since all scopes consist of the same default edge type, no explicit edge types are considered for scope pattern edges.

Furthermore, the ViewGraphTransformationRule class owns the constraints composition that targets at the RuleConstraint class. The RuleConstraint class represents additional constraints of the left-hand side of view graph transformation rules, e. g., attribute constraints. For that purpose, the RuleConstraint class owns a language attribute with ExpressionLanguage type and an expression attribute with String type. The language and expression attributes enable developers to express rule constraints in arbitrary expression languages, e. g. OCL.

Furthermore, the AnnotationPatternNode class owns an assignments composition that targets at the AttributeAssignment class. The AttributeAssignment class represents assignments of values to attributes of annotations. For that purpose, the AttributeAssignment class owns a language attribute with ExpressionLanguage type and an expression attribute with String type. The language attribute describes the expression language that is used to specify the value of the expression attribute. Furthermore, the AttributeAssignment class owns a type association that targets at the AttributeType class to describe for which kind of attribute the attribute assignment describes attribute values of annotations.

Concrete Syntax

This thesis adapts the notation of UML object models to depict view graph transformation rules. This thesis depicts pattern nodes as rectangles labeled with a name and type separated by a colon. This thesis depicts artifact pattern nodes as solid rectangles and annotation pattern nodes as dashed rectangles. Section 7.1 describes an example.

This thesis depicts pattern edges as directed lines labeled with a name and type separated by a colon. This thesis depicts relation pattern edges as solid directed lines, role pattern edges as dashed directed lines, and scope pattern edges as dotted directed lines. Scope pattern edges do not consist of a name and employ a default edge type that is omitted in view graph transformation rules.

Rule modifiers denote which rule elements belong to the left-hand side and right-hand side of the view graph transformation rules. View graph transformation rules do not specifically label rule elements that must exist. View graph transformation rules cross out rule elements that must not exist. View graph transformation rules label rule elements that are created by the right-hand side of the view graph transformation rule with a Create modifier that is depicted as "++".

View graph transformation rules depict rule constraints as strings that consist of the name of the expression language and the expression itself separated by a colon. Names used in the expression refer to artifacts and annotations that match the pattern nodes with equal names.

View graph transformation rules depict attribute assignments as strings that consist of the name of the expression language and the assignment of the attribute value separated by a colon. The assignment of the attribute value consists of the attribute name and an expression that describes the attribute value separated by the assignment operator ":=".

B.1. Generated Story Diagrams

The following sections describe story diagrams that implement the CREATE, DELETE, and UPDATE execution modes of view modules.

Creation of Annotations

Figure B.2 shows the implementation of the CREATE execution mode of view modules. The *Bind Annotation Types* story action node binds required annotation types and roles types. The *Match Pattern* story action node implements the Generalization pattern. If a match of the Generalization pattern exists, the *Does Annotation Exist* story action node checks whether the match is already marked by an annotation. If the match is not already marked, the *Create Annotation* story action node creates an annotation including roles and scopes to mark the found graph pattern match. Otherwise, the story diagram continues to search for graph pattern matches.

Deletion of Annotations

Figure B.3 shows the implementation of the DELETE execution mode of view modules. The *Match Dangling Roles and Scopes* story action node searches for annotation with dangling roles or scopes. For each annotation with dangling roles or scopes, the *Add Dependent Annotations to Output* story action node adds all dependent annotations to the output of the story diagram. Then, the *Remove Roles and Scopes* story action nodes removes all roles and scopes from the annotation. Afterwards, the *Remove Annotation* story action nodes removes the annotation.

Update of Annotations

Figure B.4 shows the implementation of the UPDATE execution mode of view modules. The *Recover Marked Match* story action nodes binds all artifacts and annotations that are marked by an anntoation. Afterwards, the *Check for Match* story action node checks whether these artifacts and annotations are still a match of the Generalization pattern. If yes, the *Add Dependent Annotations to Output* story action node adds all dependent annotations to the output of the story diagram and the *Update Attributes* story action node updates the attribute values of the annotation. If no, the *Mark Annotation for Deletion* story action node detaches artifacts and annotations from the roles of the annotation to mark the annotation for deletion.

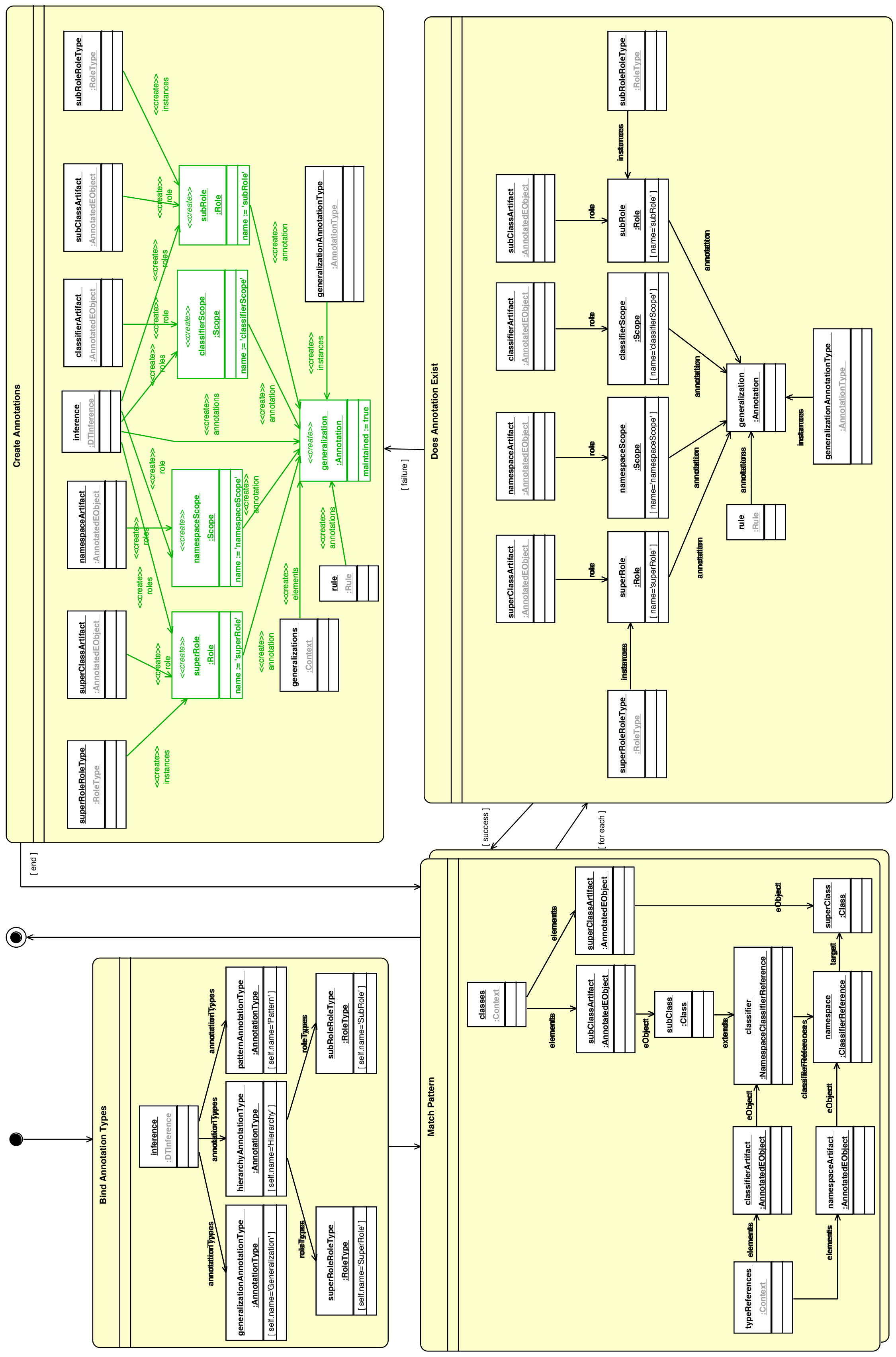

Bind Annotation Types
inference :DTInference
annotationTypes
generalizationAnnotationType :AnnotationType [self.name='Generalization']
hierarchyAnnotationType :AnnotationType [self.name='Hierarchy']
patternAnnotationType :AnnotationType [self.name='Pattern']
roleTypes
superRoleRoleType :RoleType [self.name='SuperRole']
subRoleRoleType :RoleType [self.name='SubRole']
Match Pattern
classes :Context
elements
subClassArtifact :AnnotatedEObject
superClassArtifact :AnnotatedEObject
eObject
subClass :Class
superClass :Class
extends
classifier :NamespaceClassifierReference
classifierReferences
namespace :ClassifierReference
target
classifierArtifact :AnnotatedEObject
namespaceArtifact :AnnotatedEObject
typeReferences :Context
[end]
[success]
[for each]
[failure]
Does Annotation Exist
subRoleRoleType :RoleType
instances
subClassArtifact :AnnotatedEObject
role
subRole :Role [name='subRole']
classifierArtifact :AnnotatedEObject
classifierScope :Scope [name='classifierScope']
namespaceArtifact :AnnotatedEObject
namespaceScope :Scope [name='namespaceScope']
superClassArtifact :AnnotatedEObject
superRole :Role [name='superRole']
superRoleRoleType :RoleType
annotation
annotations
generalization :Annotation
rule :Rule
generalizationAnnotationType :AnnotationType
Create Annotations
subRoleRoleType :RoleType
subClassArtifact :AnnotatedEObject
classifierArtifact :AnnotatedEObject
inference :DTInference
namespaceArtifact :AnnotatedEObject
superClassArtifact :AnnotatedEObject
superRoleRoleType :RoleType
<<create>> instances
<<create>> role
<<create>> roles
<<create>> subRole :Role name := 'subRole'
<<create>> classifierScope :Scope name := 'classifierScope'
<<create>> namespaceScope :Scope name := 'namespaceScope'
<<create>> superRole :Role name := 'superRole'
<<create>> annotation
<<create>> annotations
<<create>> elements
generalizationAnnotationType :AnnotationType
<<create>> generalization :Annotation maintained := true
generalizations :Context
rule :Rule

Figure B.2.: Create implementation of the Generalization pattern

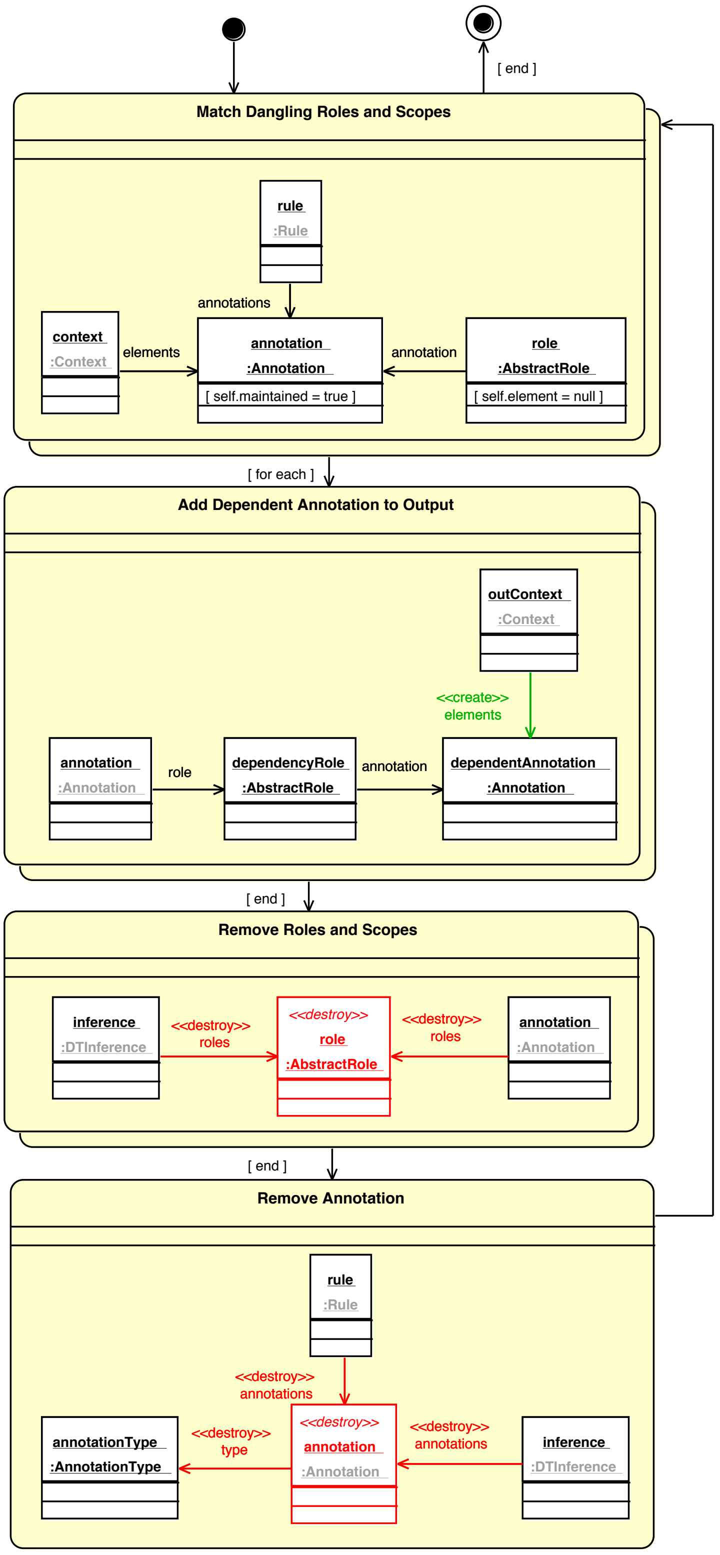
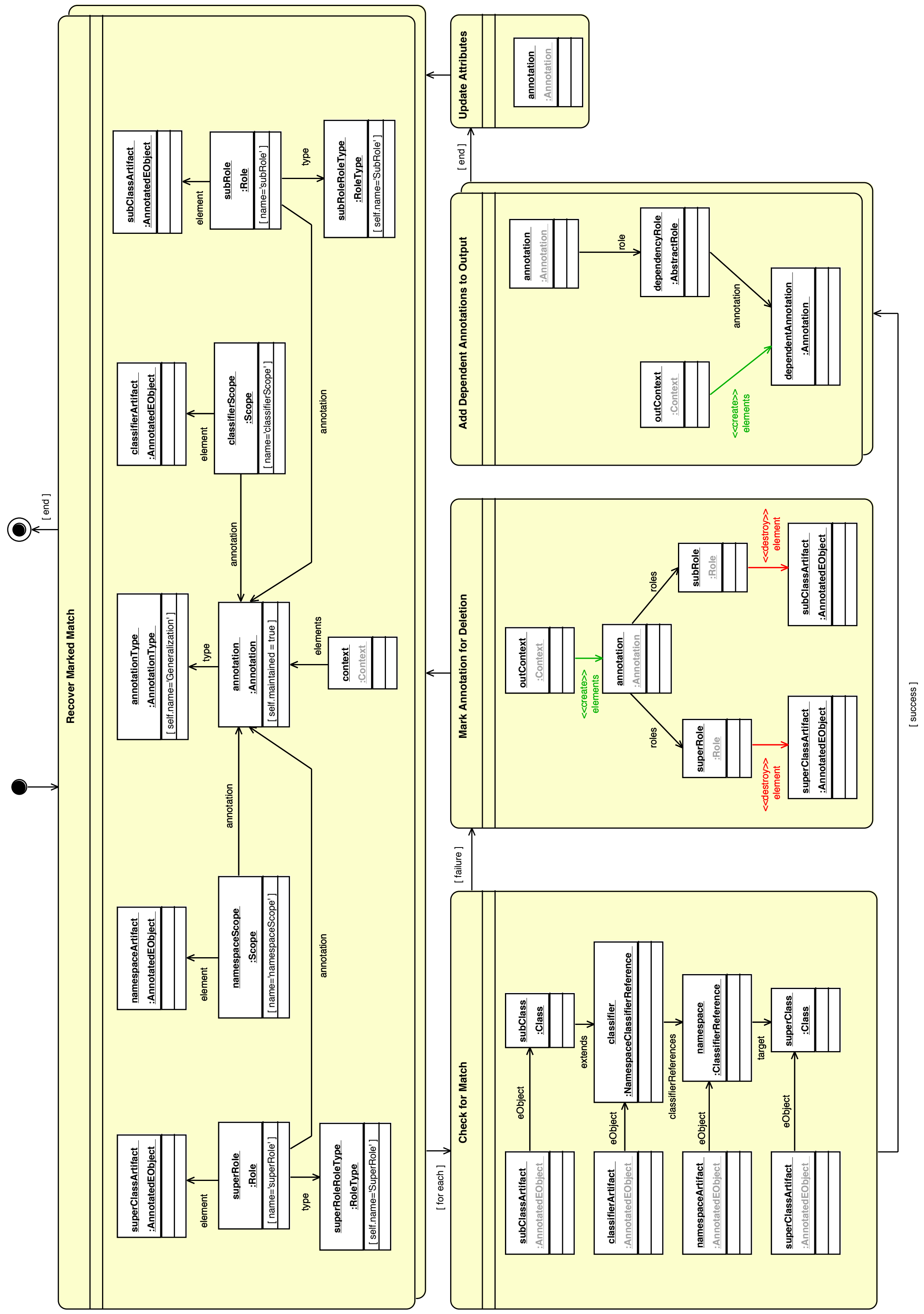


Figure B.3.: Delete implementation of the Generalization pattern



Recover Marked Match
[end]
superClassArtifact :AnnotatedEObject
element
superRole :Role
[name='superRole']
type
superRoleRoleType :RoleType
[self.name='SuperRole']
namespaceArtifact :AnnotatedEObject
namespaceScope :Scope
[name='namespaceScope']
annotation
annotationType :AnnotationType
[self.name='Generalization']
annotation :Annotation
[self.maintained = true]
elements
context :Context
classifierArtifact :AnnotatedEObject
classifierScope :Scope
[name='classifierScope']
subClassArtifact :AnnotatedEObject
subRole :Role
[name='subRole']
subRoleRoleType :RoleType
[self.name='SubRole']
[for each]
Check for Match
subClassArtifact :AnnotatedEObject
eObject
subClass :Class
extends
classifierArtifact :AnnotatedEObject
classifier :NamespaceClassifierReference
classifierReferences
namespaceArtifact :AnnotatedEObject
namespace :ClassifierReference
target
superClassArtifact :AnnotatedEObject
superClass :Class
[failure]
[success]
Mark Annotation for Deletion
outContext :Context
<<create>> elements
annotation :Annotation
roles
superRole :Role
subRole :Role
<<destroy>> element
superClassArtifact :AnnotatedEObject
subClassArtifact :AnnotatedEObject
Add Dependent Annotations to Output
annotation :Annotation
role
dependencyRole :AbstractRole
outContext :Context
dependentAnnotation :Annotation
Update Attributes
annotation :Annotation

Figure B.4.: Update implementation of the Generalization pattern

C. Mapping Graph Conditions

The following sections describe the mapping of graph conditions to view modules.

C.1. Atomic Graph Condition

Figure C.5 depicts a schematic view module that consists of three artifact input connectors that receive artifacts with X, Y, and Z artifact type. Furthermore, the view module consists of one annotation output connector with C annotation type that provides annotations, which mark graph pattern matches that satisfy the depicted graph pattern. When the graph pattern is satisfied, the view graph transformation rule creates an annotation with annotation type C and roles / scopes that mark all artifacts of the graph pattern match, when these artifacts are *not* already marked by an annotation with annotation type C (see NAC in Section 7.1.1).

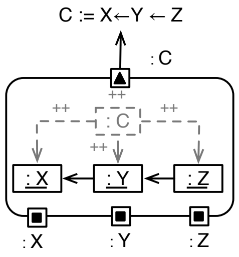


Figure C.5.: Atomic graph condition

C.2. Conjunction

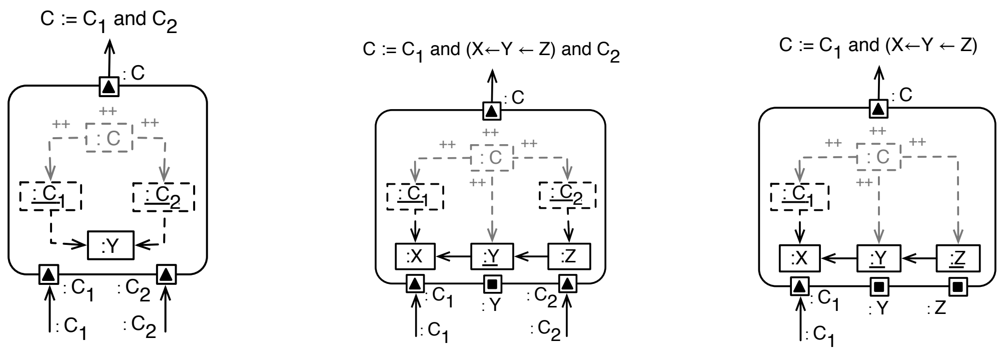
The following sections distinguish overlapping, disjoint, and extended graph patterns.

Overlapping Graph Patterns

Figure C.6(a) shows a schematic view module that implements a conjunction with overlapping graph patterns. The view module consists of two annotation input connectors and receives annotations with annotation type C_1 and C_2, respectively. The view graph transformation rule shows that the graph pattern match marked by the annotation with annotation type C_1 and the graph pattern match marked by the annotation with annotation type C_2 must have an artifact (or annotation) with artifact type (or annotation type) Y in common that has a certain role in one graph pattern match and a certain role in another graph pattern match. When the view graph transformation rule finds a match, the view graph transformation rule creates an annotation with annotation type C that marks both annotations with the help of roles or scopes, if the match is not already marked by an annotation with annotation type C (see NAC in Section 7.1.1).

Disjoint Graph Patterns

Figure C.6(b) shows a schematic view module that implements a conjunction with disjoint graph patterns. The view module consists of two annotation input connectors that receive annotations with annotation type C_1 and C_2, respectively. Additionally, the view module consists of an artifact input connector that receives artifacts with artifact type Y. The view graph transformation rule shows that the artifact with artifact type X must be part of the graph pattern match marked by the annotation with annotation type C_1 and the artifact



(a) Overlapping patterns (b) Disjoint patterns (c) Extended patterns

Figure C.6.: Conjunctions of graph patterns

with artifact type Z must be part of the graph pattern match marked by the annotation with annotation type C_2. Furthermore, the view graph transformation rule shows that the artifact with artifact type Y must connect the artifacts with artifact type X and Z. The view graph transformation rule describes that an annotation with annotation type C is created in the view graph that marks both reused annotations and the artifact with artifact type Y, if the annotations and artifacts of the graph pattern match are *not* already marked by an annotation with annotation type C (see NAC in Section 7.1.1).

Extended Graph Patterns

Figure C.6(c) shows a view module that owns one annotation input connector with annotation type C_1 and the artifact input connectors with artifact type Y and Z. The encapsulated view graph transformation rule depicts that the artifact pattern node with artifact type X of the graph pattern is extended by the artifact pattern nodes with artifact type Y and Z. Furthermore, the artifact that matches the artifact pattern node with artifact type X must be part of a graph pattern match marked by the annotation that matches the annotation pattern node with C_1 annotation type. The right-hand side of the view graph transformation rule describes that an annotation with annotation type C is created that marks the annotation that matches the annotation pattern node with annotation type C_1, the artifact that matches the artifact pattern node with artifact type Y, and the artifact that matches the artifact pattern node with artifact type Z, if these artifacts and annotations are *not* already marked by an annotation with annotation type C (see NAC in Section 7.1.1).

C.3. Disjunction

Figure C.7 shows a schematic view module that consists of one annotation input connector with annotation type C_{Super}. The annotation type C_{Super} is the annotation supertype of the annotation types C_1 and C_2. The view module produces annotations with annotation type C, which describe that artifacts that match the artifact pattern node X either satisfy graph patterns represented by annotations with annotation type C_1 or C_2. That means, the graph pattern exploits the polymorphism of the received annotations to map disjunctions.

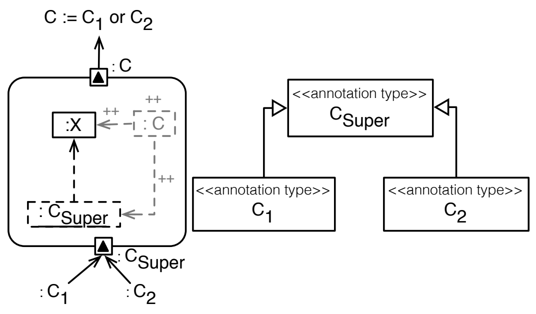


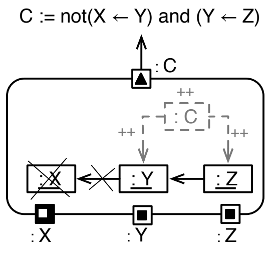
Figure C.7.: Disjunction of graph conditions

C.4. Negation

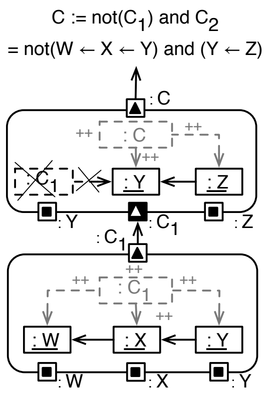
The following sections distinguish simple and complex negation (cf. Definition 8).

Simple Negation

Figure C.8(a) shows a schematic view module, which implements a simple negation. The view module receives artifacts with artifact type X, Y, and Z. The encapsulated view graph transformation rule describes that artifacts with artifact type Y must not be connected to artifacts with artifact type X. When the left-hand side of the transformation rule is satisfied, the right-hand side of the transformation rule creates an annotation with annotation type C that marks all existing graph nodes that satisfy the graph pattern, if these graph nodes are not already marked by an annotation with annotation type C (see NAC in Section 7.1.1).



(a) Simple negation



(b) Complex negation

Figure C.8.: Negation of graph conditions

Complex Negation

Figure C.8(b) shows two schematic view modules, which implement a complex negation. The non-negated and negated part of the graph pattern overlap in the artifact pattern node with artifact type Y. The view module at the bottom searches for graph pattern matches that dissatisfy the complex negation. That means, instead of searching for matches that satisfy graph pattern $\neg C_1$, the view module at the bottom searches for matches that satisfy graph pattern C_1. The view module marks each match for the graph pattern C_1. Afterwards, the view module at the bottom forwards all created annotations with annotation type C_1 to the successor view module on top. The view module on top checks whether no annotation with annotation type C_1 is connected to artifacts that match the artifact pattern node with artifact type Y. If yes, no graph nodes exist that dissatisfy the complex negation. Then, the view module on top creates an annotation that marks all graph nodes that match the non-negated part of the graph pattern, if these graph nodes are *not* already marked by an annotation with annotation type C (see NAC in Section 7.1.1).

C.5. Recursion

Figure C.9 shows two view modules, which implement a recursive graph condition. The view module at the bottom describes the recursion start. The view module on top describes the recursion step, because the output connector of the view module is connected to the input connector of the view module. The view module at the bottom receives artifacts with artifact type X and creates an annotation with annotation type C_{Start} for each pair of connected artifacts with artifact type X. The annotation marks the artifact that is the source of the connecting edge by a role with role type Start. The annotation marks the artifact that is the target of the connecting edge by a role with role type End. The view module on top receives artifacts with artifact type X and annotations with annotation type C_{Start} created by the view module at the bottom. The view module on top creates annotations with annotation type C_{Step}. Note that annotation type C_{Step} is an annotation subtype of annotation type C_{Start}. The view module on top takes annotations with annotation type C_{Start} and searches for artifacts with artifact type X that are connected to the artifact that acts in the End role of annotations with annotation type C_{Start}. If the left-hand side of the view graph transformation rule is satisfied, the right-hand side of the view graph transformation rule creates an annotation with annotation type C_{Step} that marks the annotation with annotation type C_{Start} as previous Step of the recursion and the artifact with artifact type X that is connected to the artifact that acts in the End role of the previous recursion step as artifact with End role of the current recursion step, if the annotations and artifacts are *not* already marked by an annotation with annotation type C_{Step} (see NAC in Section 7.1.1). Afterwards, created annotations with annotation type C_{Step} are forwarded to the input connector of the same view module. Since the annotation type C_{Step} is an annotation subtype of the annotation type C_{Start}, the annotation pattern node with annotation type C_{Start} matches annotations with annotation type C_{Step}. That means, the graph pattern exploits the polymorphism of annotations, which describe matches of recursion starts and steps, to implement recursion. The view module an top is executed until it does not create annotations anymore.

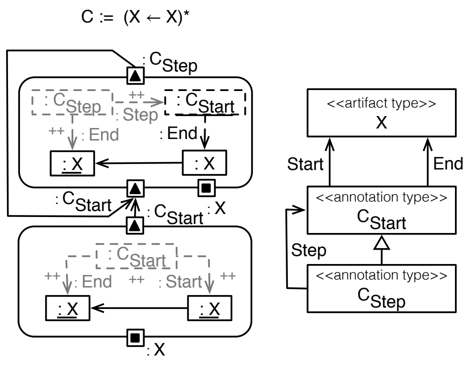


Figure C.9.: Schematic view module implementation

D. View Graph Maintenance

D.1. Reachability Test for Complex Negations

Algorithm D.1 describes a reachability test that collects all annotations that become suspicious due to created annotations. This algorithm searches for annotations that mark matches of graph patterns, which implement complex NACs, and become suspicious due to created annotations. In summary, the algorithm searches for annotations that have the same annotation types as the output connectors of the view modules that depend on the module that created an annotation. For that purpose, the algorithm traverses all artifacts and annotations that are relevant to these dependent modules and checks whether annotations are reachable that have the same annotation type as the annotations that are created by these dependent modules.

Algorithm D.1 Find suspicious annotations due to created annotations

```
Input: Created annotations
Output: Suspicious annotations that are maintained by successor view modules
 1: procedure REACHABILITY_SUSPICIOUS(annotations)
 2:   suspiciousAnnotations := $\emptyset$
 3:   for each annotation in annotations do
 4:     dependentModules := modules that dependent on the module that created the annotation
 5:     for each module in dependentModules do
 6:       if module is connected via negative connector to module that created annotation then
 7:         outputConnector := output connector of module
 8:         scope := artifacts and annotations that are marked by annotation
 9:         repeat
10:           added := $\emptyset$
11:           for each node in scope do
12:             for each connector of module do
13:               if connector is artifact or annotation input connector then
14:                 for each adjacentNode of node do
15:                   if adjacentNode.type $\in$ clan(connector.type) then
16:                     added := added $\cup$ {adjacentNode}
17:             if node is annotation then
18:               if node.type = outputConnector.type then
19:                 suspiciousAnnotations := suspiciousAnnotations $\cup$ {node}
20:           scope := added
21:         until added = $\emptyset$
22:   return suspiciousAnnotations
```

Algorithm D.1 shows the Reachability_Suspicious procedure that receives a set of created annotations. First, the procedure initializes an empty set of suspicious annotations. Then, the procedure iterates the set of received annotations. For each annotation, the procedure determines all modules that dependent on the module that created the annotation. For each of these dependent modules, the procedure checks whether the module is connected via a negative input connector to the module that created the annotation. If yes, the procedure retrieves the output connector of the module and initializes a scope with the artifacts and annotations that are marked by the received annotation. Then, the procedure checks for each node in the scope whether its adjacent nodes have a (sub-)type that is processed by the module. If yes, the procedure adds the adjacent node to a set of added nodes. If a node in the scope is an annotation, the procedure checks whether the annotation has the same type as the output connector of the dependent module. If yes, the procedure adds the node as suspicious annotation to the set of suspicious annotations. When the procedure processed all nodes in the scope, the procedure replaces the scope with the adjacent nodes that are stored by the added variable. The procedure repeats the lookup of adjacent nodes until it finds no new adjacent nodes anymore. Finally, the procedure returns all collected suspicious annotations.

E. Optimized View Graph Maintenance

E.1. Naive Duplicate Handling

The following algorithms describe the lookup of annotations with equal attribute values. Algorithm E.2 searches for annotations in a set of annotations that have the same attribute values as the annotation that would result from a graph pattern match and returns true, if such annotations exist. Otherwise, the procedure returns false. Algorithm E.3 searches for annotations in a set of annotations that have the same attribute values as a given annotation and returns true, if such annotations exist. Otherwise, the procedure returns false. Algorithm E.4 searches for an annotation in a set of annotations that has the same attribute values as the annotation that would result from a graph pattern match and returns this annotation. Otherwise, the procedure returns null.

Algorithm E.2 Checks whether match results in equal values of annotation attributes

```
Input: Graph pattern match and existing annotations for this match
Output: True, if annotation with equal attribute values exists. Otherwise, false.
 1: procedure EQUAL_ATTRIBUTE_VALUES(match, annotations)
 2:    for each annotation in annotations do
 3:       equalAttributes := 0
 4:       for each attribute in annotation.attributes do
 5:          assignment := corresponding attribute assignment of view transformation
 6:          if attribute.value = evaluation result of assignment expression for match then
 7:             equalAttributes++
 8:       if equalAttributes = length(annotation.attributes) then      //all attribute values are equal
 9:          return true
10:    return false
```

Algorithm E.3 Checks whether annotation has equal attribute values as existing annotation

```
Input: Preserved annotation and potential annotation duplicates
Output: True, if annotation with equal attribute values exists. Otherwise, false.
 1: procedure EQUAL_ATTRIBUTE_VALUE(annotation, existingAnnotations)
 2:    for each existingAnnotation in existingAnnotations do
 3:       equalAttributes := 0
 4:       for each existingAttribute in existingAnnotation.attributes do
 5:          for each attribute in annotation.attributes do
 6:             if existingAttribute.name = attribute.name then
 7:                if existingAttribute.value = attribute.value then
 8:                   equalAttributes++
 9:       if equalAttributes = length(annotation.attributes) then      //all attribute values are equal
10:          return true
11:    return false
```

Algorithm E.4 Lookup annotation that has equal attribute values

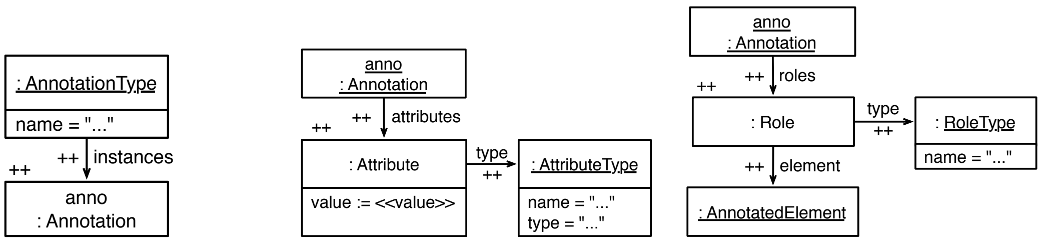
```
Input: Graph pattern match and existing annotations for this match
Output: Annotation with equal attribute values. Otherwise, null.
 1: procedure RETRIEVE_ANNOTATION_DUPLICATE(match, annotations)
 2:    for each annotation in annotations do
 3:       equalAttributes := 0
 4:       for each attribute in annotation.attributes do
 5:          assignment := corresponding attribute assignment of view transformation
 6:          if attribute.value = evaluation result of assignment expression for match then
 7:             equalAttributes++
 8:       if equalAttributes = length(annotation.attributes) then      //all attribute values are equal
 9:          return annotation
10:    return null
```

E.2. Annotations with Aggregations

Algorithm E.5 shows the Create_Annotation_Aggregate procedure. This procedure creates annotations that employ scope aggregations to mark graph nodes of matches that do not have certain roles in matches. The procedure receives a pattern match for which the procedure has to create an annotation. The procedure creates an annotation, adds attributes to the annotation, sets the attribute values and creates roles for each role rule link of the view graph transformation with Create modifier. If the view transformation consists of scope rule links, the procedure calls the Add_Scope_Aggregation procedure and passes the found pattern match and the created annotation to add all graph nodes that are not marked by roles to a scope aggregation. Finally, the procedure returns the created annotation.

Algorithm E.5 Create annotations with scope aggregations

```
Input: Graph pattern match that has to be marked
Output: Annotation that marks the match
 1: procedure CREATE_ANNOTATION_AGGREGATE(match)
 2:     annotation := create annotation for match                          //cf. Fig. E.10(a)
 3:     for each attribute assignment of view transformation do
 4:         create attribute                                                //cf. Fig. E.10(b)
 5:         evaluate expression of attribute assignment                     //cf. Fig. E.10(b)
 6:     for each role rule link of view transformation rule do
 7:         if role rule link has CREATE modifier then
 8:             create role                                                 //cf. Fig. E.10(c)
 9:     if view transformationrule has scope rule links then
10:         ADD_SCOPE_AGGREGATION(match, annotation)                        //cf. Algorithm E.6
11:     return annotation
```



(a) Create annotation (b) Create attributes (c) Create roles

Figure E.10.: Graph patterns and graph transformations of Create implementation with scope aggregations

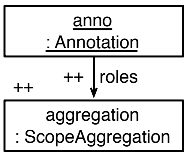
Algorithm E.6 shows the Add_Scope_Aggregation procedure. The procedure receives the pattern match for which the procedure has to add a scope aggregation to the received annotation. First, the procedure adds a scope aggregation to the annotation. Then, the procedure checks for each scope rule link of the view transformation rule, whether the scope rule link has a Create modifier. If yes, the procedure adds the graph node that matches the target of the scope rule link to a scope and adds the scope to the scope aggregation.

Algorithm E.7 shows the Delete_Scope_Aggregations procedure. The procedure deletes scope aggregations of an annotation. For that purpose, the procedure receives the scope aggregation that has to be deleted. First, the procedure deletes dangling and non-dangling scopes of the scope aggregation. Afterwards, the procedure deletes the scope aggregation.

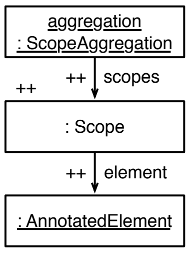
Algorithm E.8 shows the Obsolete_Scope_Aggregation procedure. The procedure receives a

Algorithm E.6 Create scope aggregation

```
Input: Graph pattern match and the annotation that has to mark this match
1: procedure ADD_SCOPE_AGGREGATION(match, annotation)
2:     scopeAggregation := add scope aggregation to annotation          //cf. Fig. E.11(a)
3:     for each scope rule link of view transformation rule do
4:         if scope rule link has CREATE modifier then
5:             add node that matches target of scope rule link to scope aggregation   //cf. Fig. E.11(b)
```



(a) Create scopes

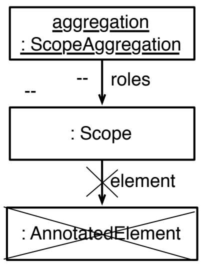


(b) Set scope target

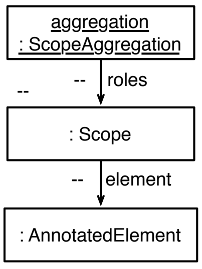
Figure E.11.: Graph patterns and graph transformations of Create implementation with scope aggregations

Algorithm E.7 Delete scope aggregation

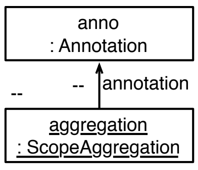
```
Input: Obsolete scope aggregation
1: procedure DELETE_SCOPE_AGGREGATIONS(aggregation)
2:     remove dangling scopes from aggregation          //cf. Fig. E.12(a)
3:     remove non-dangling scopes from aggregation      //cf. Fig. E.12(b)
4:     remove aggregation                               //cf. Fig. E.12(c)
```



(a) Dangling scope



(b) Remove scope



(c) Remove scope aggregation

Figure E.12.: Graph patterns and graph transformations of Delete implementation with scope aggregations

scope `aggregation` that has to be set obsolete. Then, the procedure detaches the graph nodes from all scopes that are owned by the passed scope `aggregation`.

Algorithm E.8 Set scope aggregation obsolete

```
Input: Scope aggregation
Output: Obsolete annotations and extracted annotations
1: procedure OBSOLETE_SCOPE_AGGREGATION(aggregation)
2:     for each scope in aggregation.scopes do
3:         detach marked graph node from scope          //cf. Fig. E.13
```

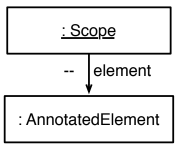


Figure E.13.: Detach scope

Algorithm E.8 shows the Obsolete_Scope_Aggregation procedure that sets scope aggregations obsolete. The algorithm receives a scope aggregation that has to be set obsolete. For each scope of the scope aggregation, the algorithm detaches the marked graph node.

Expanding Annotations with Aggregations

The Algorithm E.9 transforms annotations with scope aggregations into annotations without scope aggregations by creating a set of scopes for each scope aggregation. The Expand_Matches procedure receives annotations with scope aggregations as input. First, the procedure initializes an empty set of transformed annotations that stores all transformed annotations without scope aggregations. For each received annotation, the algorithm checks whether the annotation consists of scope aggregations. If yes, the algorithm transforms the annotation that owns the scope aggregation into an annotation without scope aggregation by iterating over all roles of the annotation. Then, the algorithm checks whether the role is a scope aggregation. For each scope aggregation, the algorithm creates a new transformed annotation. Then, the algorithm adds the scopes of the scope aggregation to the new transformed annotation. Afterwards, the algorithm adds each role of the annotation to the transformed annotation.

If the received annotation does not consist of scope aggregations, the algorithm adds the received annotation to the set of transformed annotations, because the received annotation only consists of roles that do not have to be transformed.

When the algorithm transformed all annotations, the algorithm returns all transformed annotations.

Algorithm E.9 Expand annotations with scope aggregations

```
Input: Annotations with scope aggregations
Output: Expanded annotations with original scopes
 1: procedure EXPAND_MATCHES(annotations)
 2:   transformed := ∅
 3:   for each annotation in annotations do
 4:     if annotation has scope aggregations then    //transform annotation into annotations without aggregations
 5:       for each role in annotation.roles do
 6:         if role is scope aggregation then
 7:           transformedAnnotation := create new annotation
 8:           for each scope in role.scopes do
 9:             transformedAnnotation.roles := transformedAnnotation.roles ∪ {scope}
10:           for each role2 in annotation.roles do
11:             if role2 is role then
12:               transformedAnnotation.roles := transformedAnnotation.roles ∪ {role2}
13:           transformed := transformed ∪ {transformedAnnotation}
14:     else                                          //no annotation duplicates exist
15:       transformed := transformed ∪ {annotation}
16:   return transformed
```

E.3. Space and Time Complexity

This section describes the space complexity and time complexity of the view graph maintenance, when view modules employ a) no duplicate handling, b) naive duplicate handling, and c) duplicate handling with aggregation. This section describes and discusses worst-case scenarios for each kind of duplicate handling.

Space Complexity

The space complexity describes the number of graph nodes and edges that are required by view graphs to mark matches. For that purpose, this section describes the space complexity as a function of the number of annotation duplicates that must be stored by view graphs.

Table E.1 describes the space complexity for storing annotation duplicates. The variable d stands for the number of annotation duplicates. The variable r stands for the number of roles per annotation. The variable s stands for the number of scopes per annotation. The variable p stands for the number of pattern nodes that are encapsulated by the module.

The following sections describe the space complexity for each kind of duplicate handling and compare the space complexities for one single view module and a sequence of view modules.

No Duplicate Handling

When a view module does not employ duplicate handling, each annotation duplicate must be marked by a separate annotation. For each annotation duplicate, the view graph stores r roles. The number of roles that must be stored is equal to the number of role types that are owned by the annotation type of the annotation duplicate. For each annotation duplicate, the view graph stores s scopes. The view graph must store one scope for each graph node of a match that is not marked by a role. Thus, the number of scopes is $s = (p - r)$ for p pattern nodes and r roles. In summary, the view graph stores $f(d) = d + r \cdot d + s \cdot d = d \cdot (1 + r + s)$ graph nodes and edges in the view graph to handle d annotation duplicates.

Naive Duplicate Handling

When a view module employs a naive duplicate handling, the view graph stores one annotation for a set of annotation duplicates. For this single annotation, the view graph stores r roles and s scopes. The view graph does not store roles and scopes for annotation duplicates. The number of roles is equal to the number of role types that are owned by the annotation type of the annotation. The view graph must store one scope for each graph node of the match that is not marked by a role. Thus, the number of scopes is $s = (p - r)$ for p pattern nodes and r roles. In summary, the view graph stores $f(d) = 1 + r + s$ graph nodes and graph edges in the view graph to handle d annotation duplicates. Thus, the required number of graph nodes and edges is independent from the number of annotation duplicates.

Duplicate Handling with Aggregation

When a view module employs a duplicate handling with aggregation, the view graph stores one annotation for a set of annotation duplicates. Furthermore, the annotation employs additional scope aggregations to keep track of these annotation duplicates. For this single annotation, the view graph stores r roles. The number of roles is equal to the number of role types that are owned by the annotation type of the annotation. The view graph stores these roles only once, because the annotation duplicates have these roles in common. For each annotation duplicate, the view graph stores one scope aggregation and s scopes. The view graph must store one scope for each graph node of the match that is not marked by a role. Thus, the number of scopes is $s = (p - r)$ for p pattern nodes and r roles. The view graph must aggregate these scopes for each annotation duplicate in terms of one scope aggregation. In summary, the view graph stores $f(d) = 1 + r + d + s \cdot d$ graph nodes and edges in the view graph to handle d annotation duplicates.

Table E.1.: Space complexity of duplicate handling

	Space Complexity	
	Single View Module	Sequence of View Modules
No Handling	$f(d) = d \cdot (1 + r + s) \in O(d)$	$f(d_n) = \sum_{i=1}^{n}(d_{i-1} \cdot c_i) \cdot (1 + r_i + s_i) \in O(d)$
Naive	$f(d) = 1 + r + s \in O(1)$	$f(d_n) = \sum_{i=1}^{n} 1 + r_i + s_i \in O(1)$
Aggregation	$f(d) = 1 + r + d + s \cdot d \in O(d)$	$f(d_n) = \sum_{i=1}^{n} 1 + r_i + d_i + s_i \cdot d_i \in O(d)$

No. of role (r) and scopes (s) per annotation, pattern nodes (p), annotation duplicates (d)

Discussion

In general, the space complexity of view modules is $O(n)$, because each graph pattern match is marked by one annotation. The following paragraphs describe the space complexity of view modules as function of the number of annotation duplicates.

The number of graph nodes and edges that must be stored by the view graph is proportional to the number of annotation duplicates, when the view module employs no duplicate handling and duplicate handling with aggregation. The space complexity for storing annotation duplicates without duplicate handling is $O(n)$, because the number of graph nodes and edges doubles, when the number of annotation duplicates doubles. The space complexity for storing annotation duplicates with aggregation is $O(n)$, because the number of graph nodes and edges that are required for storing scope aggregations and scopes doubles, when the number of annotation duplicates doubles. However, the number of graph nodes and graph edges increases slower for duplicate handling with aggregation than for no duplicate handling, because the view modules that employ no duplicate handling store additional roles for each annotation duplicate in contrast to duplicate handling with aggregation. The space complexity for storing annotation duplicates in a naive manner is $O(1)$, because the view graph always stores one annotation with a fixed number of roles and scopes independent from the number of annotation duplicates. But, the naive duplicate handling loses the capability to enumerate all graph pattern matches instantly.

When searching for matches, view modules consider annotations provided by predecessor view modules. In the worst case, each of these annotations participates in a match. View modules that employ different kinds of duplication handling provide different numbers of annotations to successor view modules. This number of provided annotations has an impact on the space complexity of successor view modules.

View modules that employ naive duplicate handling and duplicate handling with aggregation create only one annotation for a set of annotations. Therefore, successor view modules have to consider only one annotation for a set of annotation duplicates during their graph pattern matching. Consequently, the number of graph nodes and graph edges stored by the view graph adds up for a sequence of view modules.

For naive duplicate handling, the view graph stores $f(d_n) = \sum_{i=1}^{n}(1 + r_i + s_i)$ graph nodes and graph edges, when the view module dependency graph consists of n dependent view modules. Thus, the space complexity for naive duplicate handling is $O(1)$, because the number of stored annotations, roles, and scopes is independent from the number of annotation duplicates.

For duplicate handling with aggregation, the view graph stores $f(d_n) = \sum_{i=1}^{n}(1 + r_i + d_i + s_i \cdot d_i)$ graph nodes and graph edges, when the view module dependency graph consists of n dependent view modules. Thus, the space complexity for duplicate handling with aggregation is $O(n)$, because the number of stored scope aggregations and scopes dependents on the number of annotation duplicates.

View modules that employ no duplicate handling create one annotation for each annotation duplicate. Therefore, successor view modules have to consider all annotation duplicates during their graph pattern matching as well. Consequently, the number of graph nodes and edges stored by the view graph adds up for a sequence of view modules and the number of annotations created by successor view modules depends on the number of annotations duplicates created by predecessor view modules. Thus, the view graph stores $f(d_n) = \sum_{i=1}^{n}(d_{i-1} \cdot c_i) \cdot (1 + r_i + s_i)$ graph nodes and edges, when the view module dependency graph consists of n dependent view modules. The variable d_i stands for the number of annotation duplicates that are created by the predecessor view module. The variable c_i stands for the number of annotation duplicates that are created by the successor view module for each received annotation duplicate. Consequently, the space complexity is $O(n)$, because for each annotation duplicate created by a predecessor view module, the successor view module finds an equivalent match. Therefore, the number of annotations increases faster than for the duplicate handling with aggregation.

Time Complexity

The time complexity describes the number of possible mappings between the graph nodes of the pattern and the graphs nodes received by a view module for pattern matching. For that purpose, this section describes the time complexity as a function of the number of annotation duplicates that must be considered by a view module during graph pattern matching.

The time complexity for generating each possible mapping between p graph nodes of a graph pattern to m graph nodes a graph is $O(m^p)$ [102]. Artifacts of base graphs and annotations in view graphs belong to this graph. Thus, the number of annotation duplicates that have to be considered during graph pattern matching has an impact on the execution time of view modules. When a view module receives annotations, the graph pattern that is encapsulated by the view module employs pattern nodes that are bound to the received annotations. Thus, the mapping from the graph pattern node to the received annotations is fixed and does not have to be matched again.

Table E.2 describes the time complexity for executing view modules, i. e., searching for matches of the encapsulated graph pattern. The variable m stands for the number of artifacts and annotations received by a view module. The variable p stands for the number of pattern nodes in the encapsulated pattern. The variable d stands for the number of annotation duplicates that are received by a view module.

The following sections describe the time complexity for each kind of duplicate handling and compare the time complexity for one single module and a sequence of view modules.

No Duplicate Handling

When view modules do not employ duplicate handling, successor view modules receive one annotation for each annotation duplicate. In the view module, at least one graph pattern node is bound to the received annotations and, therefore, the remaining pattern nodes have to be considered for each received annotation, when generating each possible mapping between the graph pattern and the received graph nodes of base graphs and view graphs. Thus, the execution time of a view module dependents on the number of received annotation duplicates.

For each received annotation, m^p mappings between the pattern and the received graph nodes have to be generated by the view module. Thus, the view module has to generate $f(d) = d \cdot m^p$ mappings in total.

Naive Duplicate Handling

When view modules employ naive duplicate handling, successor view modules receive only one annotation for a set of annotation duplicates. Thus, only one annotation is bound to the graph pattern node that represents the annotation in the pattern and must be considered by the graph pattern matching. Therefore, the execution time of the successor view module is not impacted by the number of annotation duplicates. But, the naive duplicate handling loses the capability to enumerate all graph pattern matches instantly.

When view modules employ naive duplicate handling, $f(d) = m^p$ mappings between the pattern and the received graph nodes of base graphs and view graphs have to be generated by the view module during pattern matching.

Duplicate Handling with Aggregation

When view modules employ duplicate handling with aggregation, successor view modules receive only one annotation for a set of annotation duplicates. Thus, only one annotation is bound to the pattern node that represents the annotation in the pattern and must be considered by the pattern matching. Therefore, the execution time of the successor view module is not impacted by the number of received annotation duplicates.

When view modules employ duplicate handling with aggregation, $f(d) = m^p$ mappings between the pattern and the graph nodes received by view modules have to be generated by the view module during pattern matching.

Table E.2.: Time complexity of duplicate handling

	Time Complexity	
	Single View Module	Sequence of View Modules
No Handling	$f(d) = d \cdot m^p \in O(d)$	$f(d) = \sum_{i=1}^{n} d_{i-1} \cdot m_i^{p_i} \in O(d)$
Naive	$f(d) = m^p \in O(1)$	$f(d) = \sum_{i=1}^{n} m_i^{p_i} \in O(1)$
Aggregation	$f(d) = m^p \in O(1)$	$f(d) = \sum_{i=1}^{n} m_i^{p_i} \in O(1)$

No. of annotation duplicates (d), pattern nodes (p), graph nodes (m) received by view module

Discussion

In general, the number of mappings between the nodes of the pattern and the graph nodes received by the view module is m^p [102]. The following paragraphs describe the time complexity of view modules as function of the number of annotation duplicates.

For the naive duplicate handling and the duplicate handling with aggregation, the number of annotation duplicates has no impact on the time complexity, because both approaches create only one annotation for a set of annotation duplicates. Thus, both approaches have a $O(1)$ time complexity.

When a view module employs no duplicate handling, the view module creates one annotation per annotation duplicate. Thus, this approach has an impact on the time complexity of view modules, because they have to consider each annotation duplicate, when generating each possible mapping between the graph pattern and the graph nodes received by the view modules. Thus, this approach has a $O(n)$ time complexity.

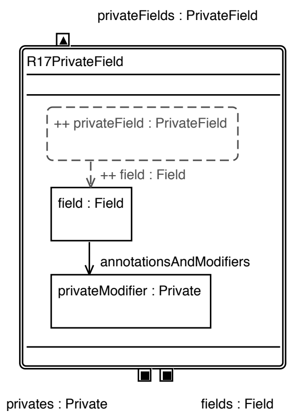
When view modules employ naive duplicate handling, view modules exchange one annotation for a set of annotation duplicates. Thus, the successor view modules have to consider at most one annotation for all annotation duplicates. Then, the remaining graph pattern nodes have to be mapped to the received graph nodes for this annotation only. Therefore, the time complexity for a sequence of view modules adds up.

For naive duplicate handling, the view module has to generate and check $f(d) = \sum_{i=1}^{n} m_i^{p_i}$ mappings between p_i graph nodes of the pattern and m_i graph nodes received by the view module. Thus, this approach has a $O(1)$ time complexity for a sequence of view modules.

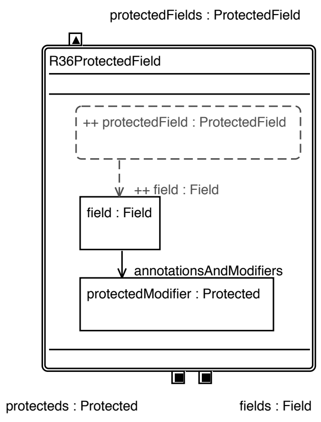
The same line of arguments holds for duplicate handling with aggregation, because successor view modules also receive only one annotation for a set of annotation duplicates. Thus, this approach has a $O(1)$ time complexity for a sequence of view modules.

When view modules employ no duplicate handling, view modules exchange all annotation duplicates. Therefore, successor view modules have to consider all annotation duplicates that are provided by predecessor view modules. Thus, a view module has to generate and check $f(d) = \sum_{i=1}^{n} d_{i-1} \cdot m_i^{p_i}$ mappings between p_i nodes of the graph pattern and m_i graph nodes received by the view module for d_{i-1} annotation duplicates provided by the predecessor module. Thus, this approach has a $O(n)$ time complexity for a sequence of view modules.

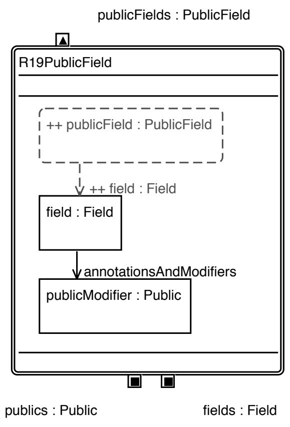
F. Case Study - Design Pattern Recovery



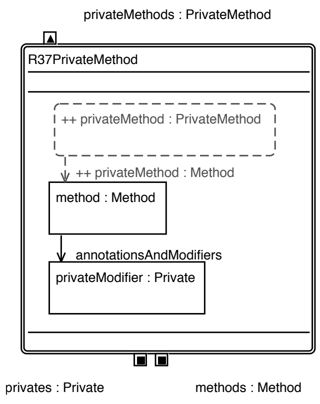
(a) Private field



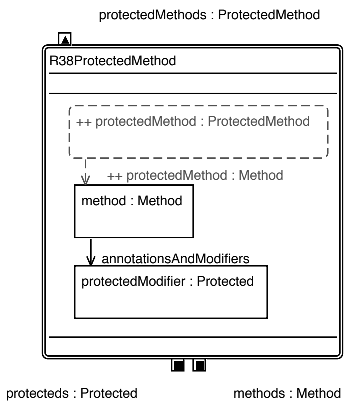
(b) Protected field



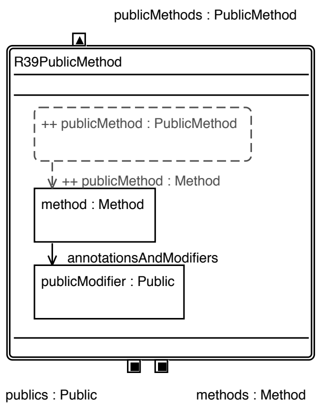
(c) Public field


privateMethods : PrivateMethod
R37PrivateMethod
++ privateMethod : PrivateMethod
++ privateMethod : Method
method : Method
annotationsAndModifiers
privateModifier : Private
privates : Private
methods : Method

(d) Private method

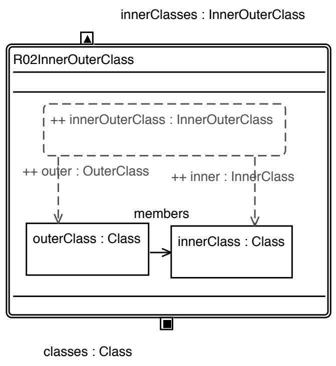

protectedMethods : ProtectedMethod
R38ProtectedMethod
++ protectedMethod : ProtectedMethod
++ protectedMethod : Method
method : Method
annotationsAndModifiers
protectedModifier : Protected
protecteds : Protected
methods : Method

(e) Protected method

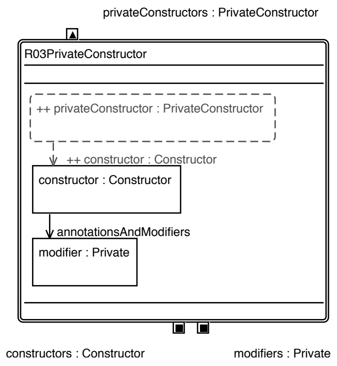


(f) Public method

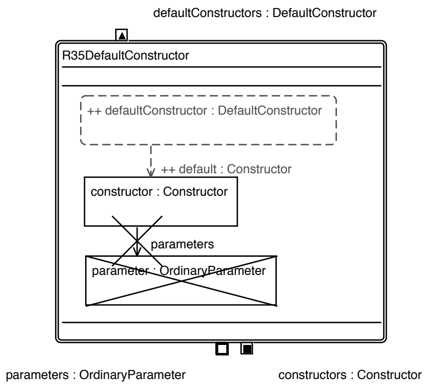
Figure F.14.: Visibility of Fields and Methods



(a) Inner and outer class

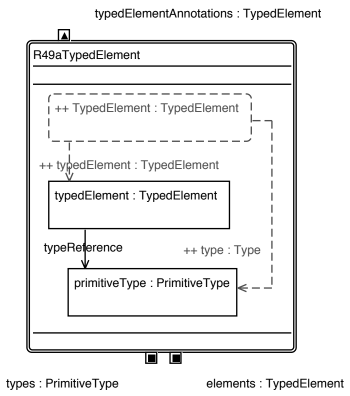

privateConstructors : PrivateConstructor
R03PrivateConstructor
++ privateConstructor : PrivateConstructor
++ constructor : Constructor
constructor : Constructor
annotationsAndModifiers
modifier : Private
constructors : Constructor
modifiers : Private

(b) Private constructor.

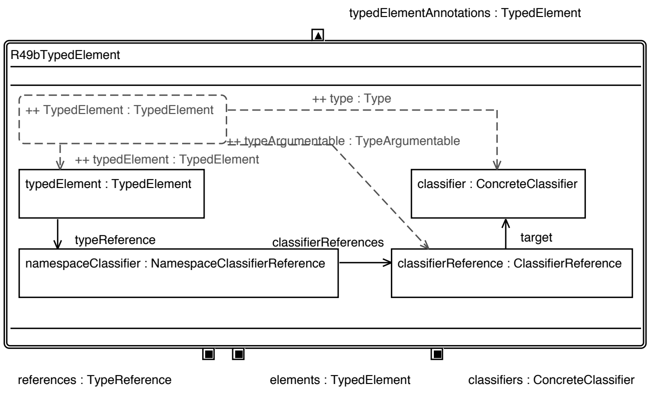

defaultConstructors : DefaultConstructor
R35DefaultConstructor
++ defaultConstructor : DefaultConstructor
++ default : Constructor
constructor : Constructor
parameters
parameter : OrdinaryParameter
parameters : OrdinaryParameter
constructors : Constructor

(c) Default constructor

Figure F.15.: Auxiliary Patterns


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

(a) Primitive typed element


typedElementAnnotations : TypedElement
R49bTypedElement
++ TypedElement : TypedElement
++ type : Type
++ typeArgumentable : TypeArgumentable
++ typedElement : TypedElement
typedElement : TypedElement
classifier : ConcreteClassifier
typeReference
classifierReferences
target
namespaceClassifier : NamespaceClassifierReference
classifierReference : ClassifierReference
references : TypeReference
elements : TypedElement
classifiers : ConcreteClassifier

(b) Complex typed element

Figure F.16.: Typed Element

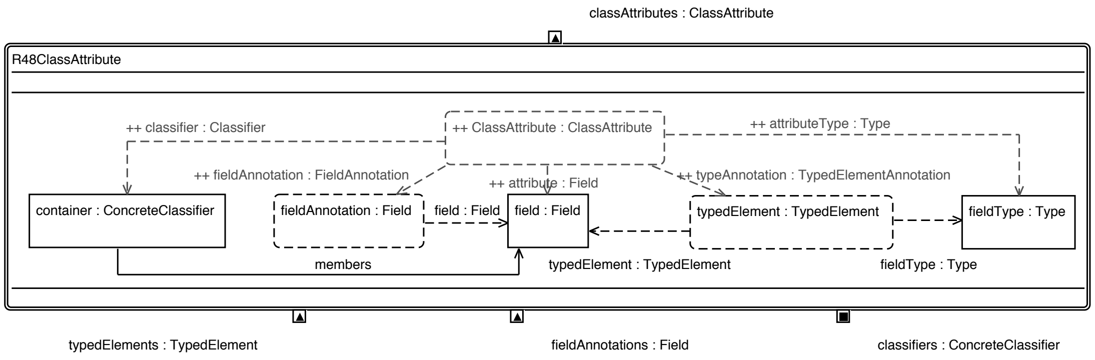
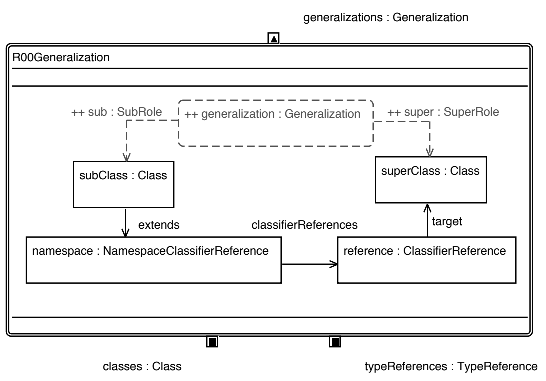
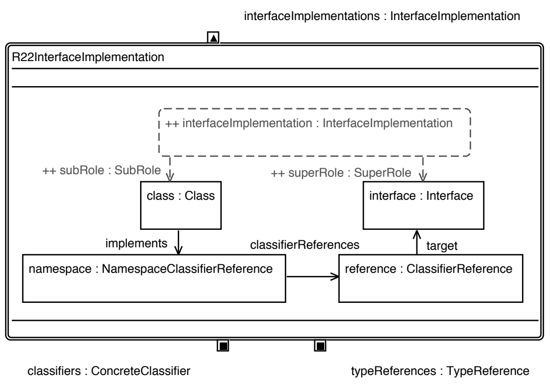

classAttributes : ClassAttribute
R48ClassAttribute
++ classifier : Classifier
++ ClassAttribute : ClassAttribute
++ attributeType : Type
++ fieldAnnotation : FieldAnnotation
++ attribute : Field
++ typeAnnotation : TypedElementAnnotation
container : ConcreteClassifier
fieldAnnotation : Field
field :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

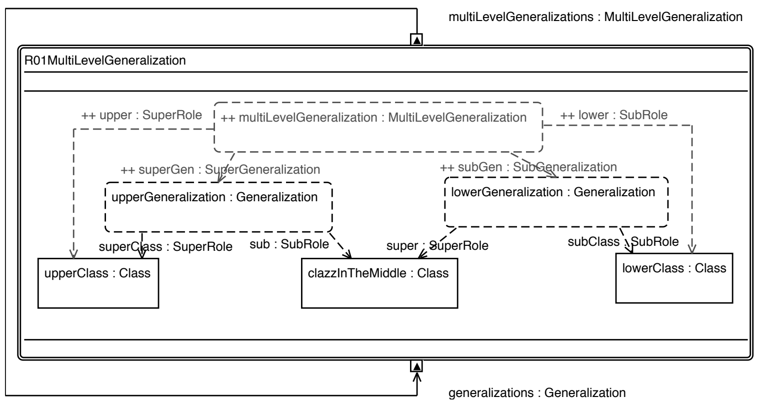
Figure F.17.: A class attribute is contained by a concrete classifier and has a primitive or complex type.


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

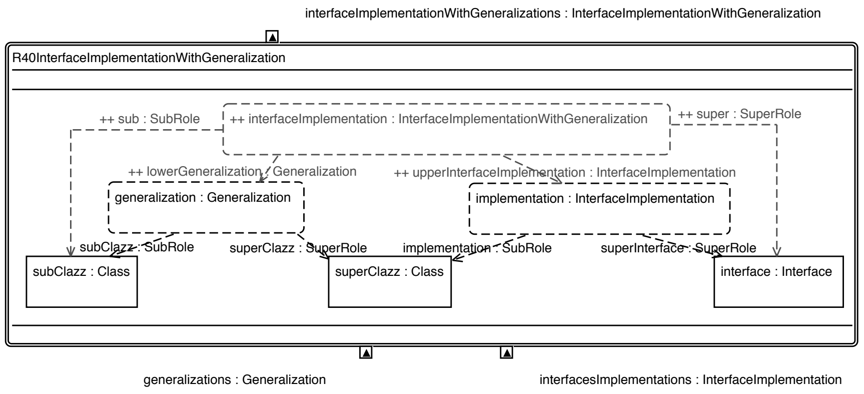
(a) A generalization between two classes exists, if the sub class points via a namespace classifier and classifier reference to its super class.


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

(b) A class implements an interface, if it points via a namespace classifier and classifier reference to an interface.

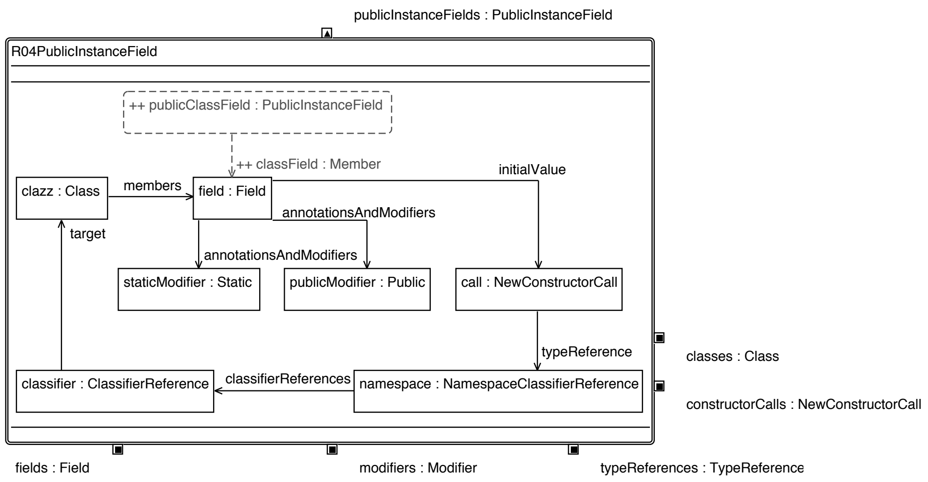

multiLevelGeneralizations : MultiLevelGeneralization
R01MultiLevelGeneralization
++ upper : SuperRole
++ multiLevelGeneralization : MultiLevelGeneralization
++ lower : SubRole
++ superGen : SuperGeneralization
++ subGen : SubGeneralization
upperGeneralization : Generalization
lowerGeneralization : Generalization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

(c) A multi-level generalization between two classes exists, if a super class is the sub class of another generalization.

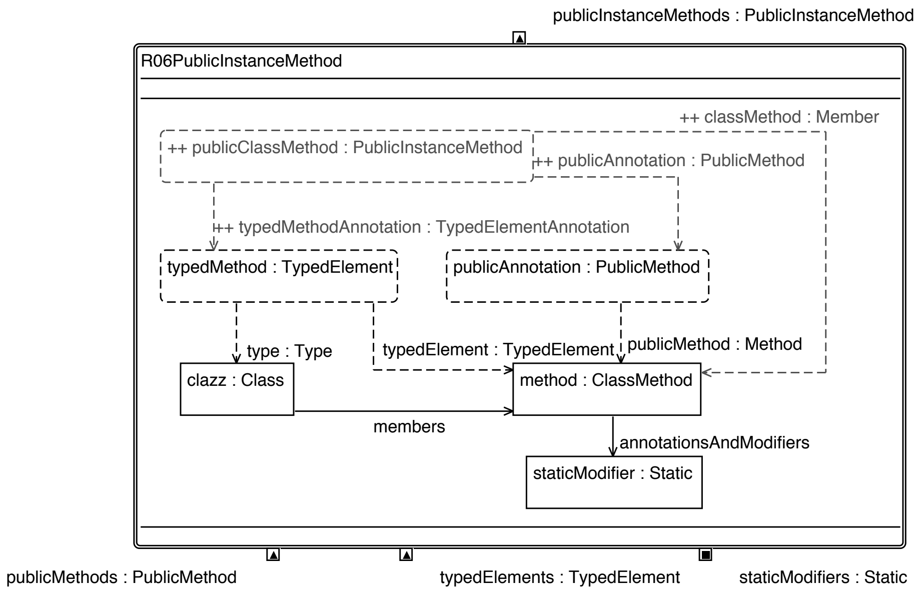

interfaceImplementationWithGeneralizations : InterfaceImplementationWithGeneralization
R40InterfaceImplementationWithGeneralization
++ sub : SubRole
++ interfaceImplementation : InterfaceImplementationWithGeneralization
++ super : SuperRole
++ lowerGeneralization : Generalization
++ upperInterfaceImplementation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

(d) A class implements an interface, if a super class implements an interface.

Figure F.18.: Hierarchy

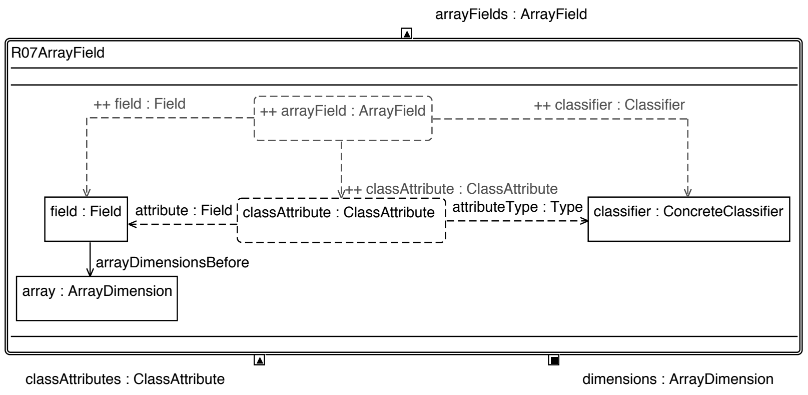


(a) A public instance field has a static and public modifier and the initial value of the field is an instance of the class that contains the field as a member. The initial value of the field is obtained from the call of the constructor of the containing class.

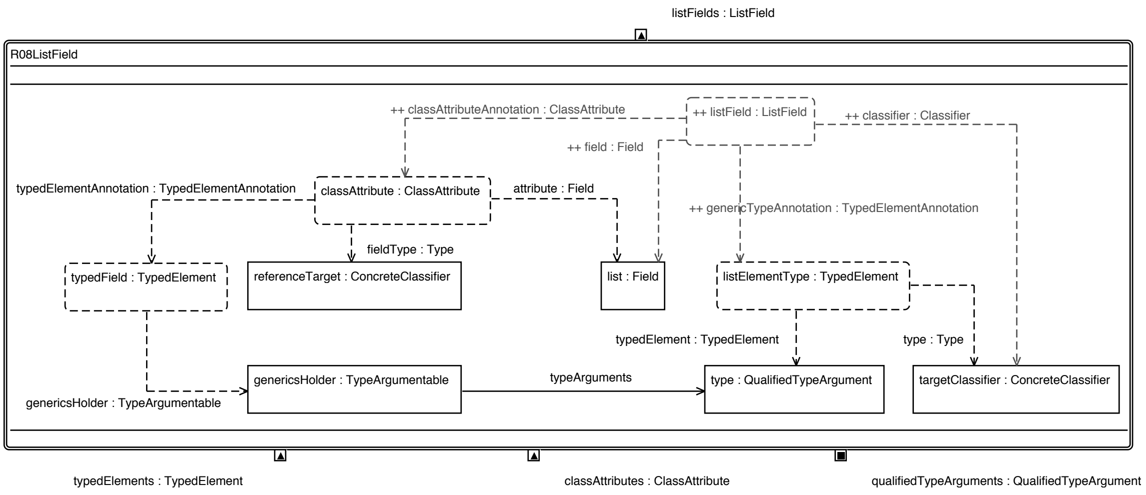

publicInstanceMethods : PublicInstanceMethod
R06PublicInstanceMethod
++ classMethod : Member
++ publicClassMethod : PublicInstanceMethod
++ publicAnnotation : PublicMethod
++ typedMethodAnnotation : TypedElementAnnotation
typedMethod : TypedElement
publicAnnotation : PublicMethod
type : Type
typedElement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

(b) A public instance method consists of a static and public modifier and returns an instance of the containing class.

Figure F.19.: Public Instance Member


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

(a) An array field is a field, which consists of an array dimension.


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

(b) A list field is a field, which defines the type of the elements contained by the list.

Figure F.20.: Association

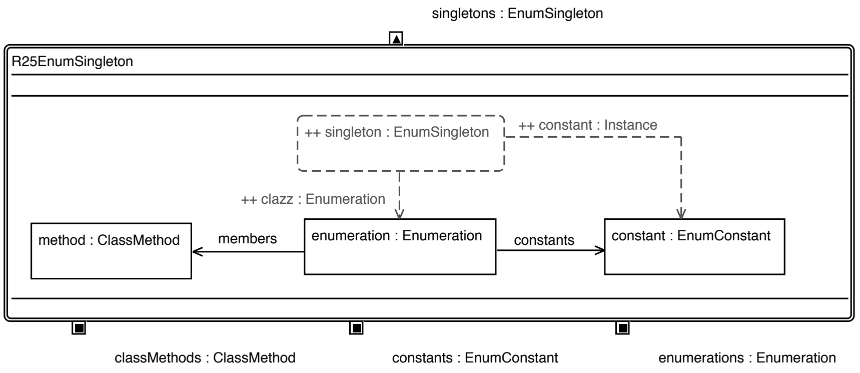


Figure F.21.: An enumeration with a method and a constant can be considered as Singleton design pattern.

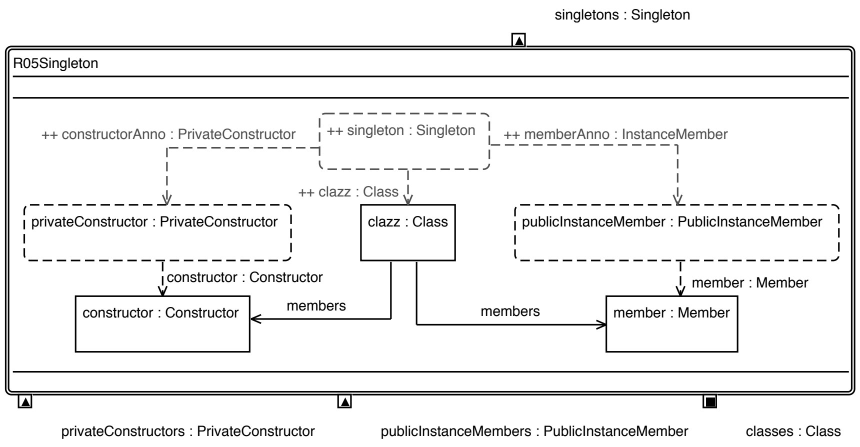

singletons : Singleton
R05Singleton
++ constructorAnno : PrivateConstructor
++ singleton : Singleton
++ memberAnno : InstanceMember
++ clazz : Class
privateConstructor : PrivateConstructor
clazz : Class
publicInstanceMember : PublicInstanceMember
constructor : Constructor
member : Member
constructor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

Figure F.22.: 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.

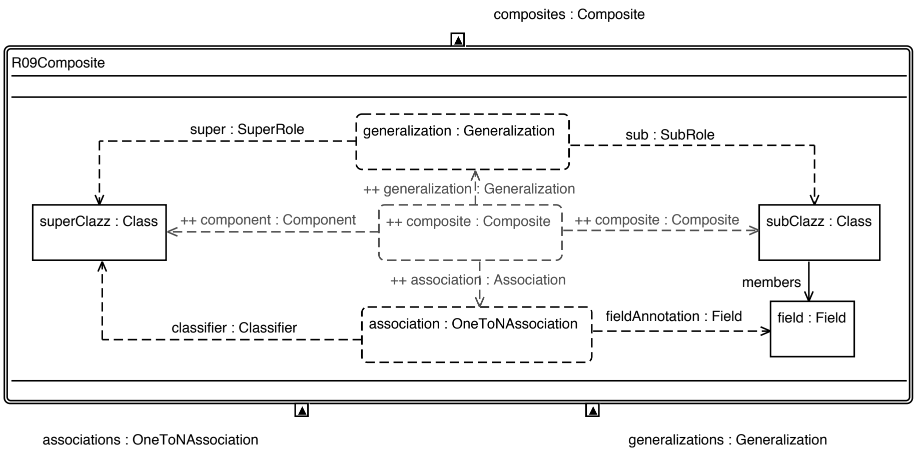


Figure F.23.: Two classes constitute a Composite design pattern, if a generalization (incl. multi-level generalization) between both classes exists and the sub class owns a to-many reference that has the super class as target classifier.

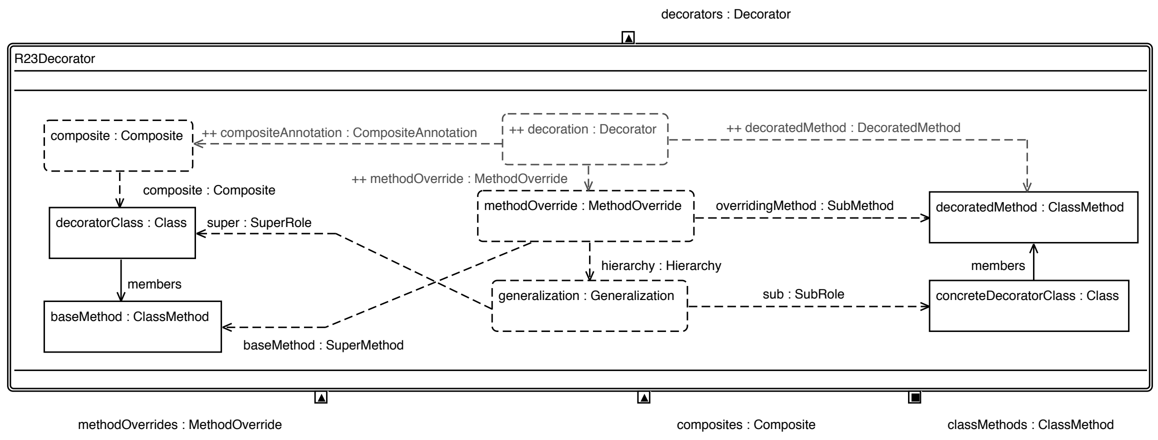

decorators : Decorator
R23Decorator
composite : Composite
++ compositeAnnotation : CompositeAnnotation
++ decoration : Decorator
++ decoratedMethod : DecoratedMethod
++ methodOverride : MethodOverride
composite : Composite
methodOverride : MethodOverride
overridingMethod : SubMethod
decoratedMethod :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

Figure F.24.: 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.

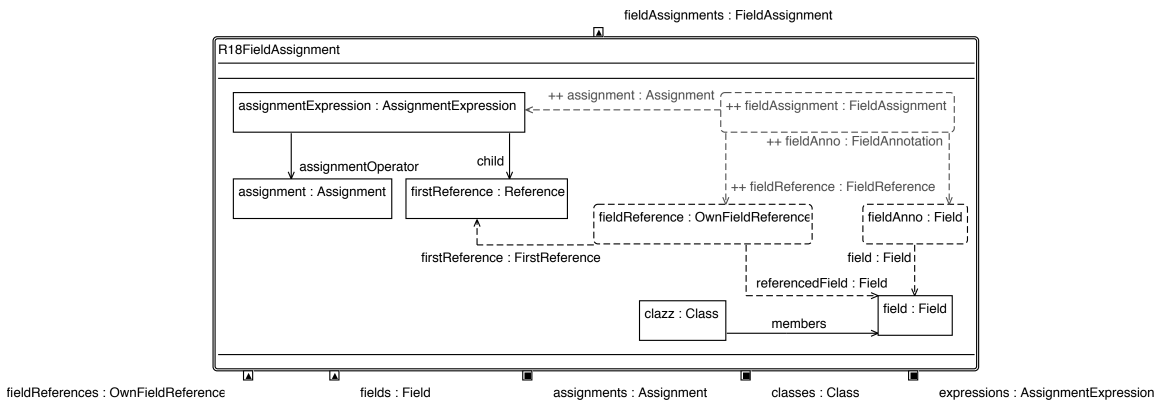
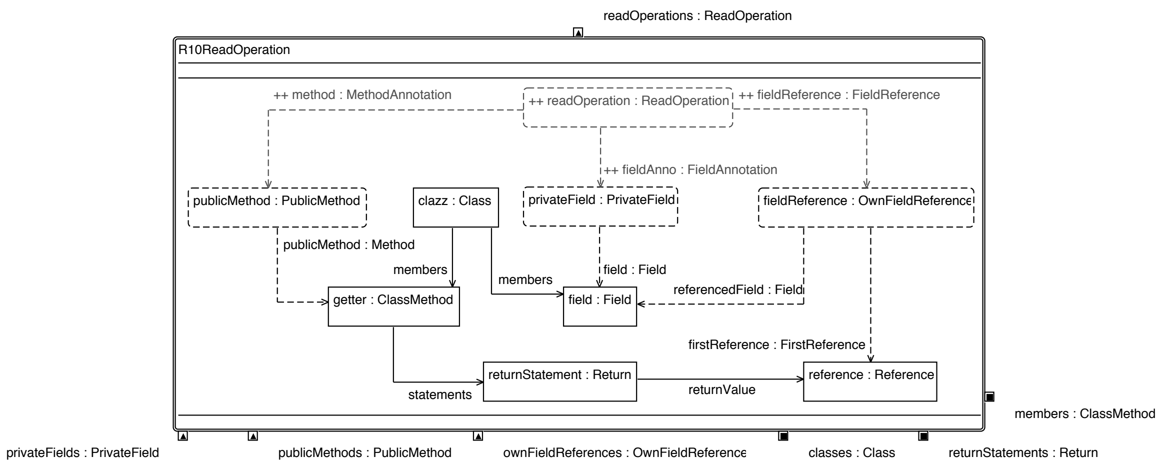
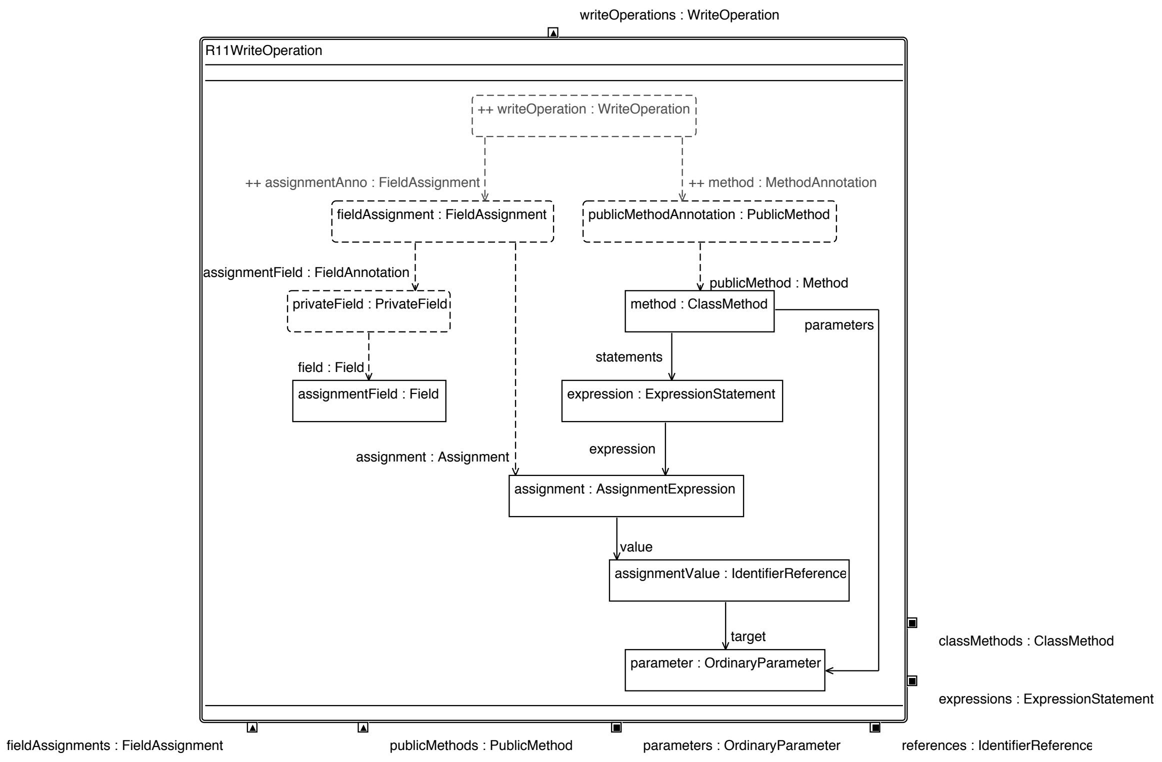

fieldAssignments : FieldAssignment
R18FieldAssignment
assignmentExpression : AssignmentExpression
++ assignment : Assignment
++ fieldAssignment : FieldAssignment
++ fieldAnno : FieldAnnotation
assignmentOperator
child
++ fieldReference : FieldReference
assignment : Assignment
firstReference : Reference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

Figure F.25.: A field assignment is present, if a field of a class is referenced in an expression that consists of an assignment.

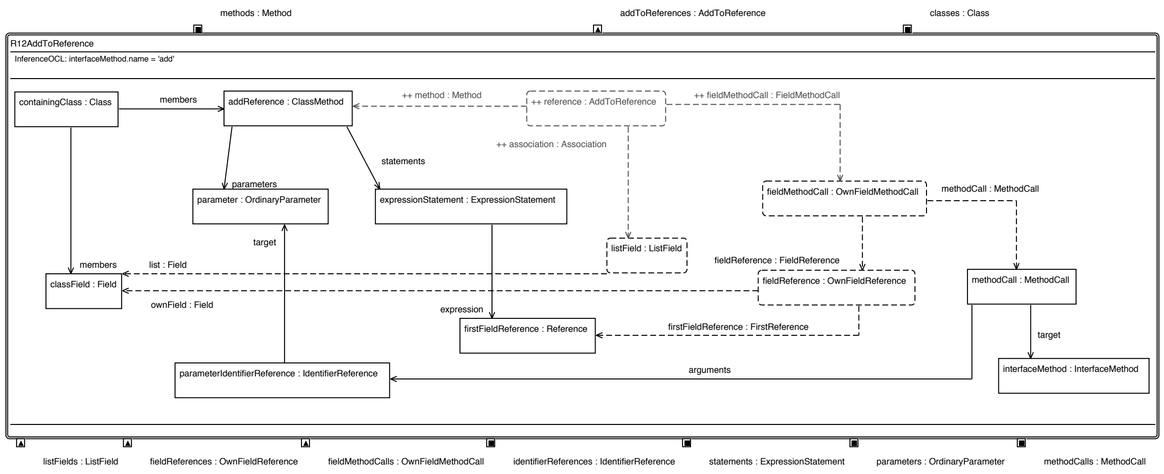

readOperations : ReadOperation
R10ReadOperation
++ method : MethodAnnotation
++ readOperation : ReadOperation
++ fieldReference : FieldReference
++ fieldAnno : FieldAnnotation
publicMethod : PublicMethod
clazz : Class
privateField : PrivateField
fieldReference : OwnFieldReference
publicMethod : Method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

(a) A read operation (getter method) of a class is a public method that returns the value of a private field owned by this class.

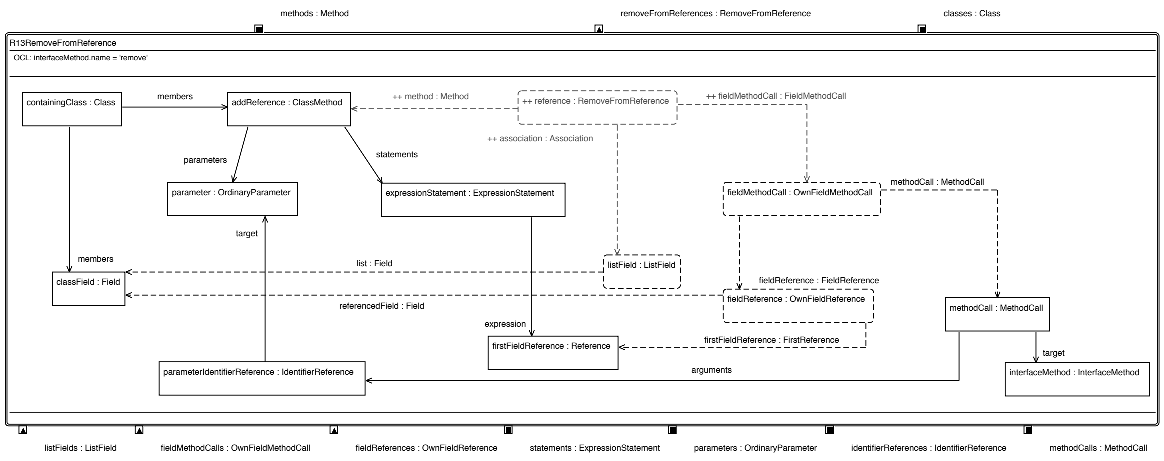

writeOperations : WriteOperation
R11WriteOperation
++ writeOperation : WriteOperation
++ assignmentAnno : FieldAssignment
++ method : MethodAnnotation
fieldAssignment : FieldAssignment
publicMethodAnnotation : PublicMethod
assignmentField : FieldAnnotation
publicMethod : Method
privateField : PrivateField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

(b) 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.

Figure F.26.: Getter and Setter

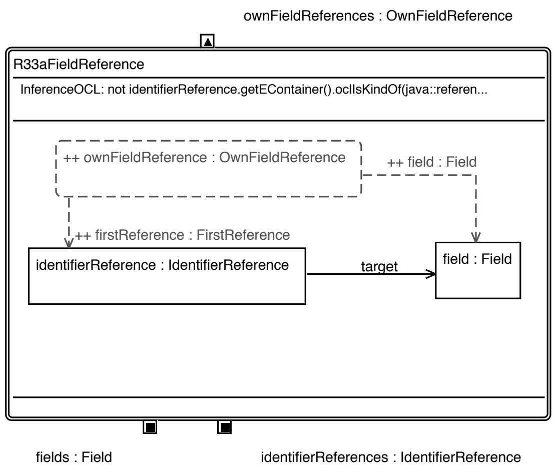

methods : Method
addToReferences : AddToReference
classes : Class
R12AddToReference
InferenceOCL: interfaceMethod.name = 'add'
containingClass : Class
members
addReference : ClassMethod
++ method : Method
++ reference : AddToReference
++ fieldMethodCall : FieldMethodCall
++ association : Association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

(a) 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.

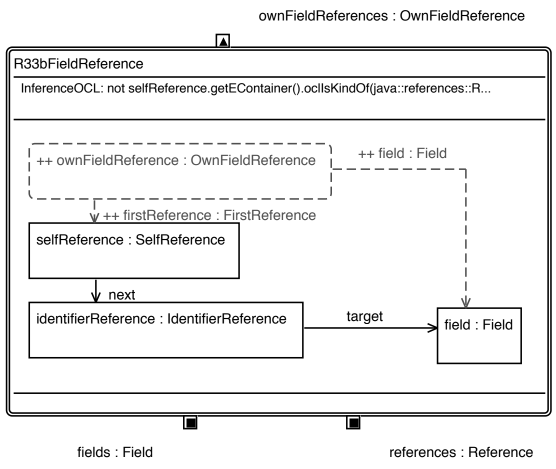

methods : Method
removeFromReferences : RemoveFromReference
classes : Class
R13RemoveFromReference
OCL: interfaceMethod.name = 'remove'
containingClass : Class
members
addReference : ClassMethod
++ method : Method
++ reference : RemoveFromReference
++ fieldMethodCall : FieldMethodCall
++ association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

(b) 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.

Figure F.27.: Add / Remove from Reference

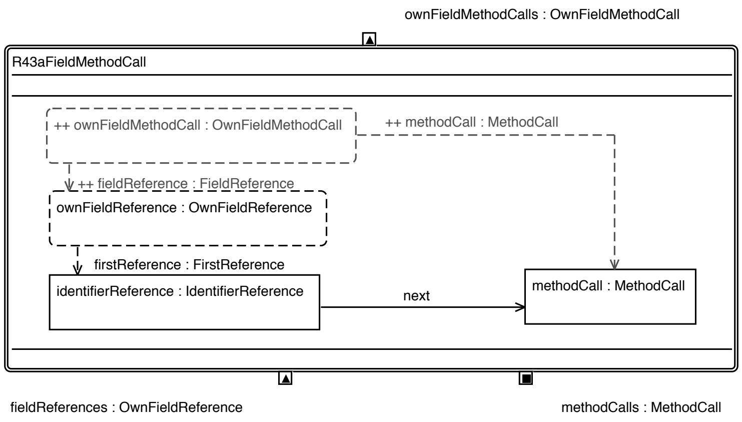

ownFieldReferences : OwnFieldReference
R33aFieldReference
InferenceOCL: not identifierReference.getEContainer().oclIsKindOf(java::referen...
++ ownFieldReference : OwnFieldReference
++ field : Field
++ firstReference : FirstReference
identifierReference : IdentifierReference
target
field : Field
fields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

(a) Field access without "this" keyword.

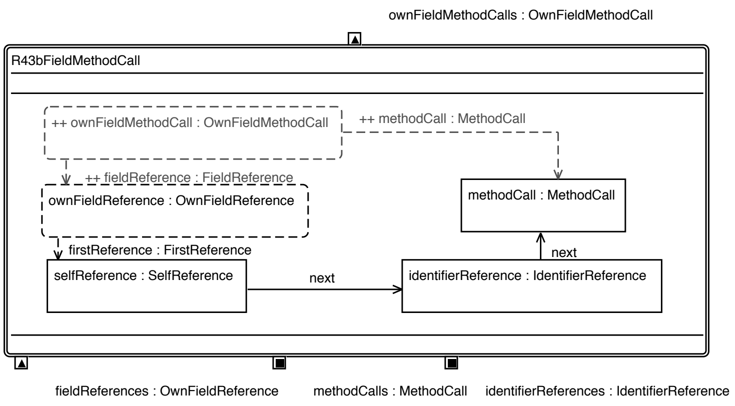

ownFieldReferences : OwnFieldReference
R33bFieldReference
InferenceOCL: not selfReference.getEContainer().oclIsKindOf(java::references::R...
++ ownFieldReference : OwnFieldReference
++ field : Field
++ firstReference : FirstReference
selfReference : SelfReference
next
identifierReference : IdentifierReference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

(b) Field access with "this" keyword.

Figure F.28.: Field Access


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

(a) Method call without "this" keyword.


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

(b) Method call with "this" keyword.

Figure F.29.: Method Calls

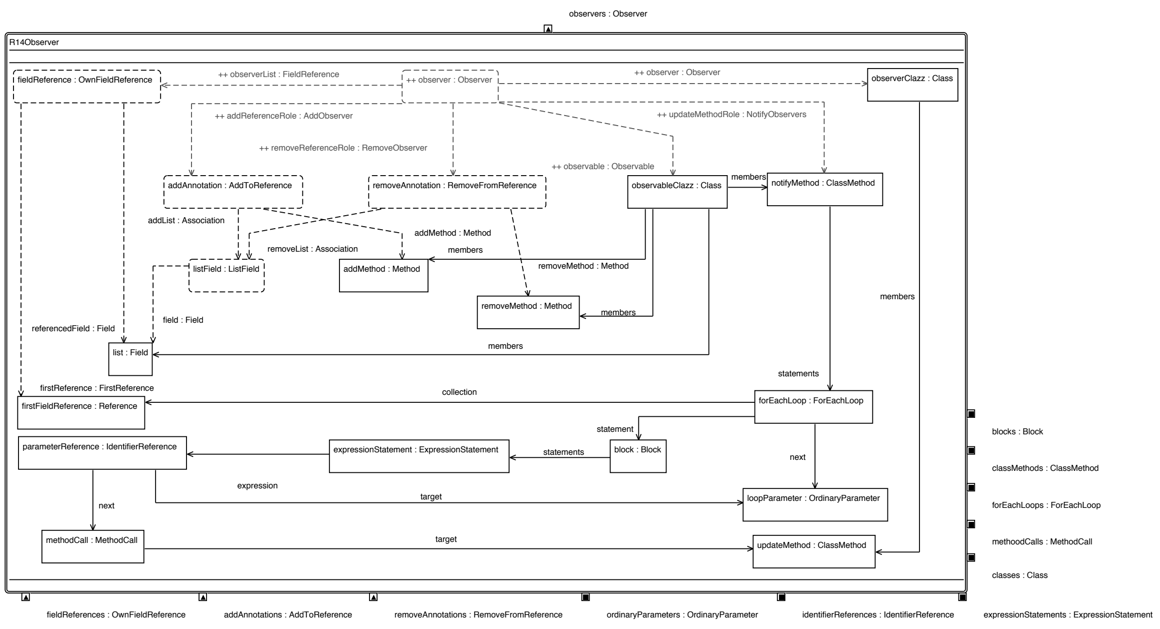

observers : Observer
R14Observer
fieldReference : OwnFieldReference
++ observerList : FieldReference
++ observer : Observer
++ observer : Observer
observerClazz : Class
++ addReferenceRole : AddObserver
++ updateMethodRole : NotifyObservers
++ removeReferenceRole : RemoveObserver
++ observable : Observable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

Figure F.30.: 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.

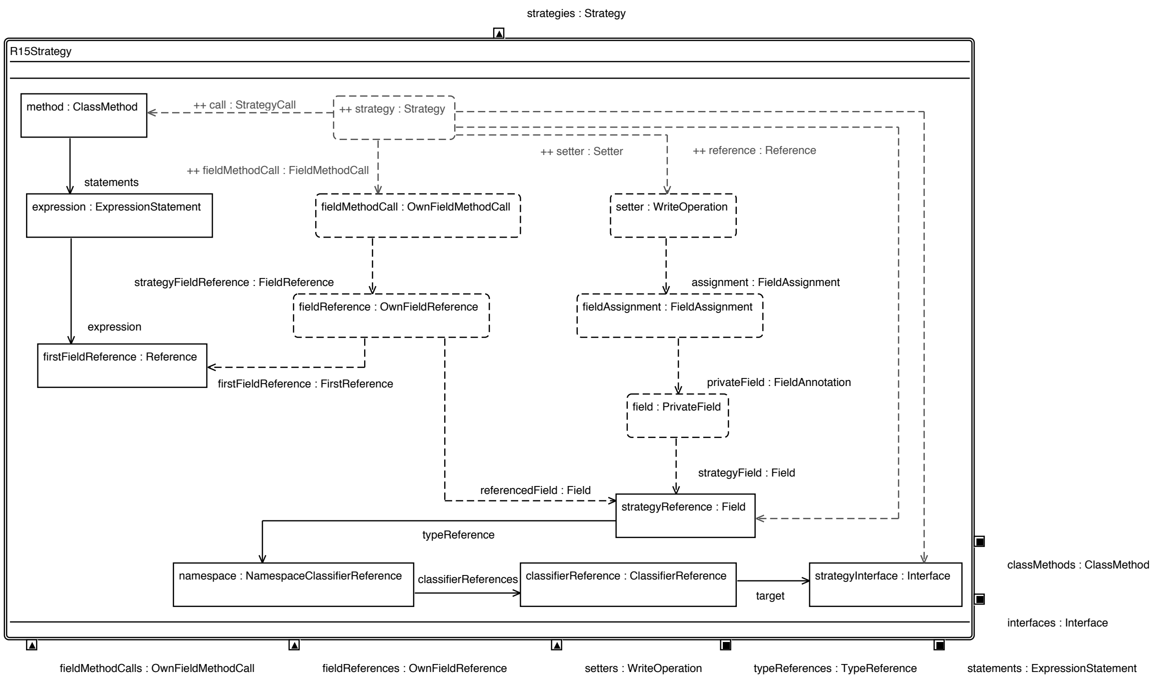
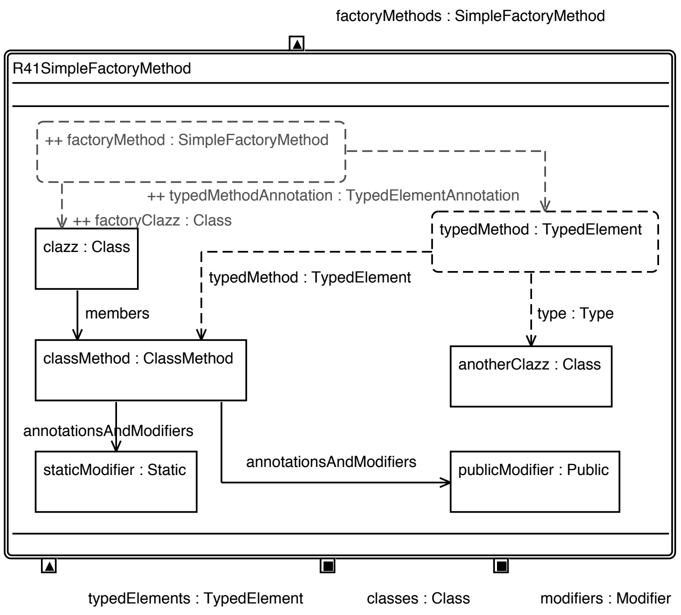
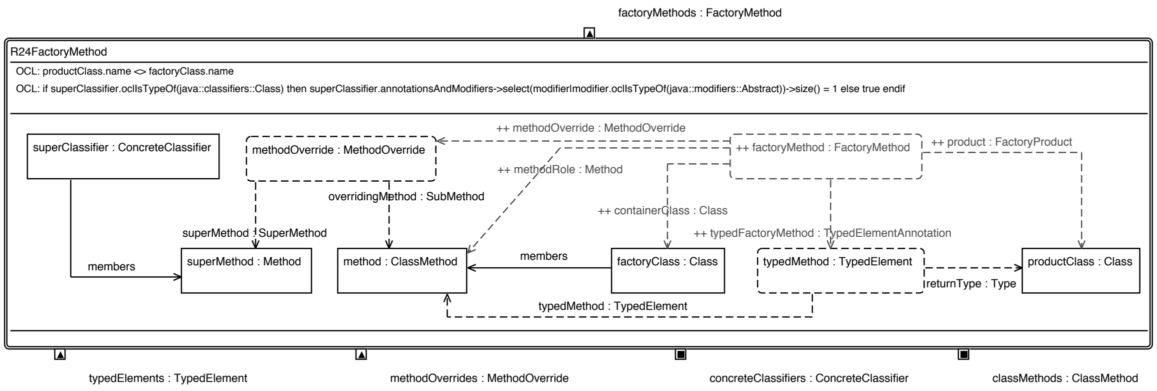

strategies : Strategy
R15Strategy
method : ClassMethod
++ call : StrategyCall
++ strategy : Strategy
++ setter : Setter
++ reference : Reference
statements
++ fieldMethodCall : FieldMethodCall
expression : ExpressionStatement
fieldMethodCall : OwnFieldMethodCall
setter : WriteOperation
strategyFieldReference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

Figure F.31.: A Strategy design pattern exists, if a write operation exists that enables to set the strategy and a method exists that executes the strategy.


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

(a) 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.


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

(b) 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.

Figure F.32.: Factory Method

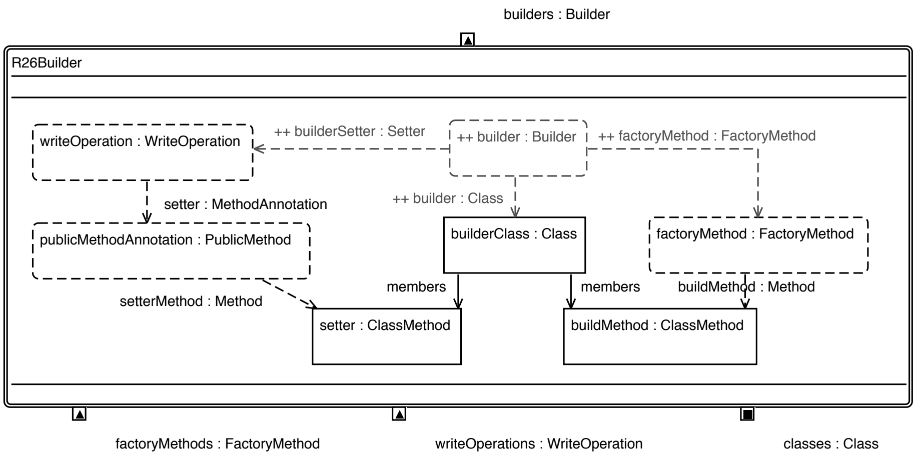

builders : Builder
R26Builder
writeOperation : WriteOperation
++ builderSetter : Setter
++ builder : Builder
++ factoryMethod : FactoryMethod
setter : MethodAnnotation
++ builder : Class
publicMethodAnnotation : PublicMethod
builderClass : Class
factoryMethod : FactoryMethod
members
members
buildMethod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

Figure F.33.: 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.

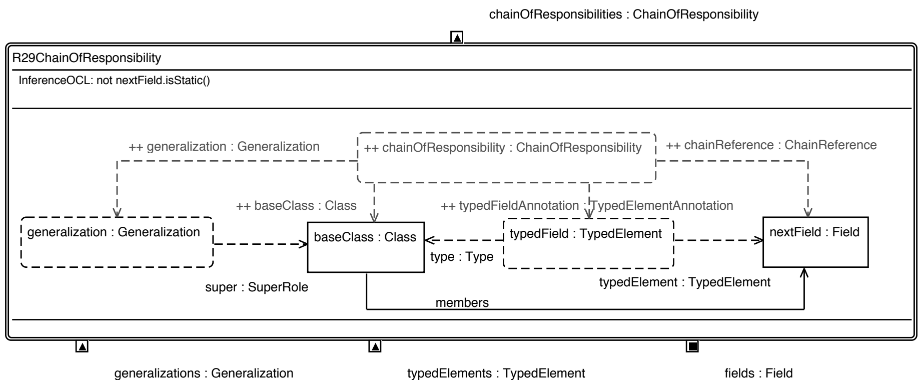

chainOfResponsibilities : ChainOfResponsibility
R29ChainOfResponsibility
InferenceOCL: not nextField.isStatic()
++ generalization : Generalization
++ chainOfResponsibility : ChainOfResponsibility
++ chainReference : ChainReference
++ baseClass : Class
++ typedFieldAnnotation : TypedElementAnnotation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

Figure F.34.: 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.

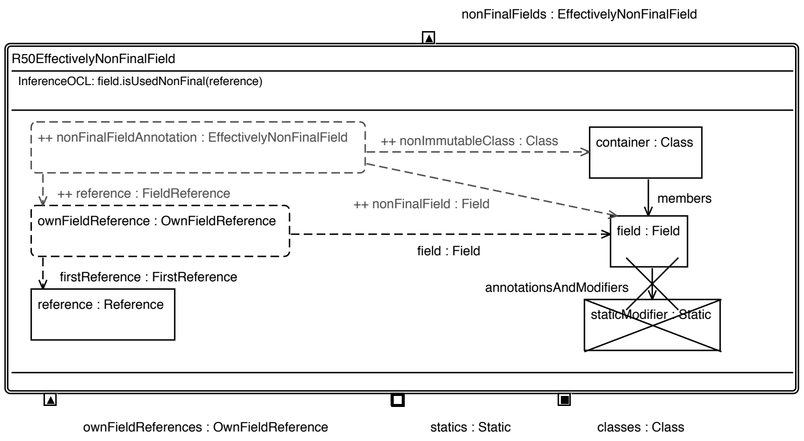

nonFinalFields : EffectivelyNonFinalField
R50EffectivelyNonFinalField
InferenceOCL: field.isUsedNonFinal(reference)
++ nonFinalFieldAnnotation : EffectivelyNonFinalField
++ nonImmutableClass : Class
container : Class
++ reference : FieldReference
members
++ nonFinalField : Field
ownFieldReference : OwnFieldReference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

Figure F.35.: 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.

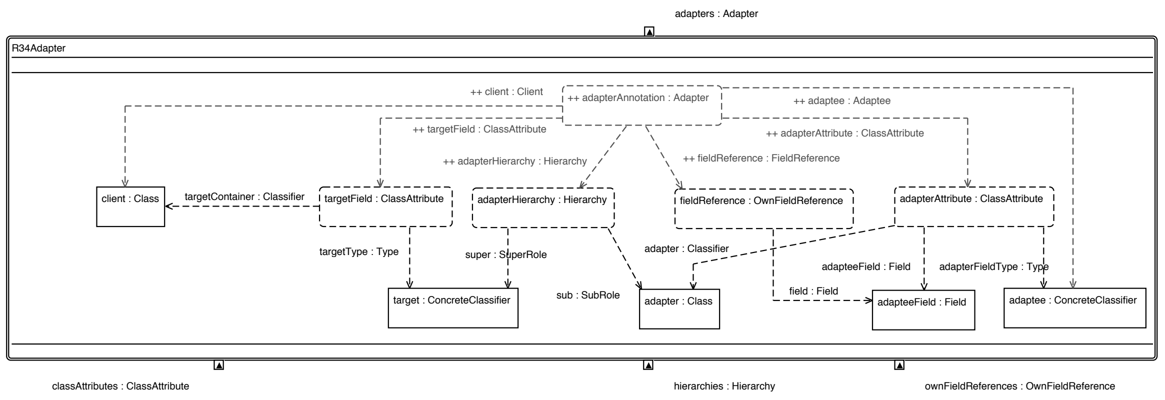

adapters : Adapter
R34Adapter
++ client : Client
++ adapterAnnotation : Adapter
++ adaptee : Adaptee
++ targetField : ClassAttribute
++ adapterAttribute : ClassAttribute
++ adapterHierarchy : Hierarchy
++ fieldReference : FieldReference
client : Class
targetContainer : Classifier
targetField : ClassAttribute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

Figure F.36.: An Adapter design pattern exists, if a hierarchy 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.

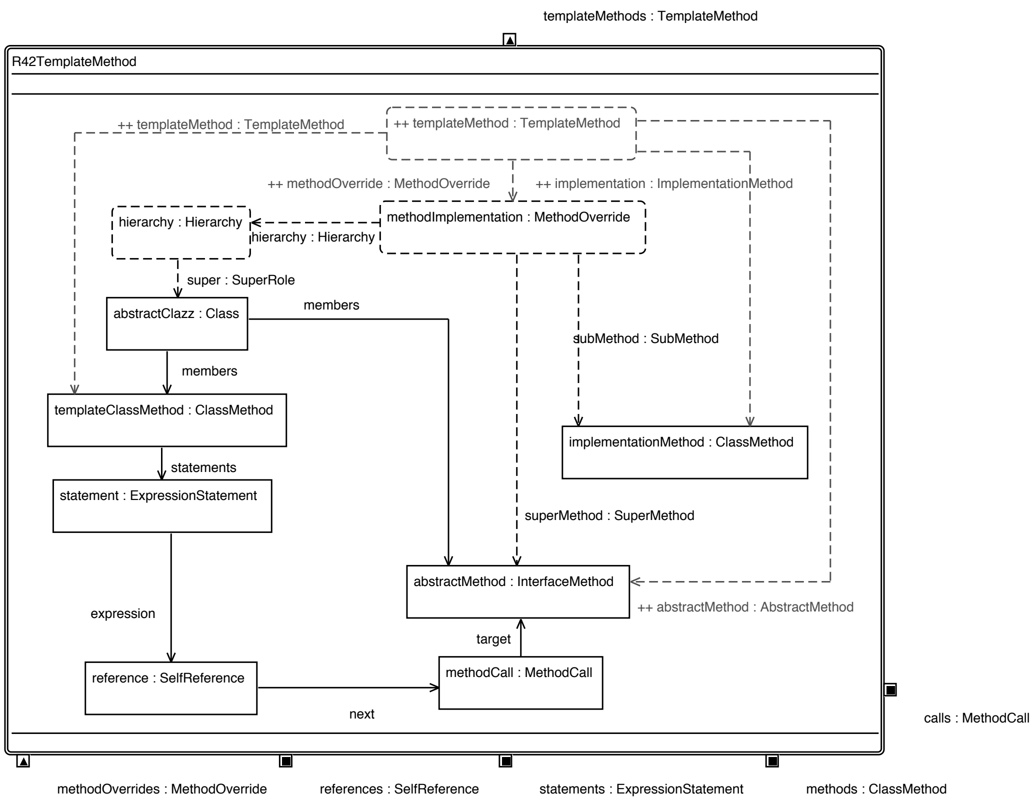

templateMethods : TemplateMethod
R42TemplateMethod
++ templateMethod : TemplateMethod
++ templateMethod : TemplateMethod
++ methodOverride : MethodOverride
++ implementation : ImplementationMethod
hierarchy : Hierarchy
hierarchy : Hierarchy
methodImplementation : MethodOverride
super : SuperRole
abstractClazz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

Figure F.37.: 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.

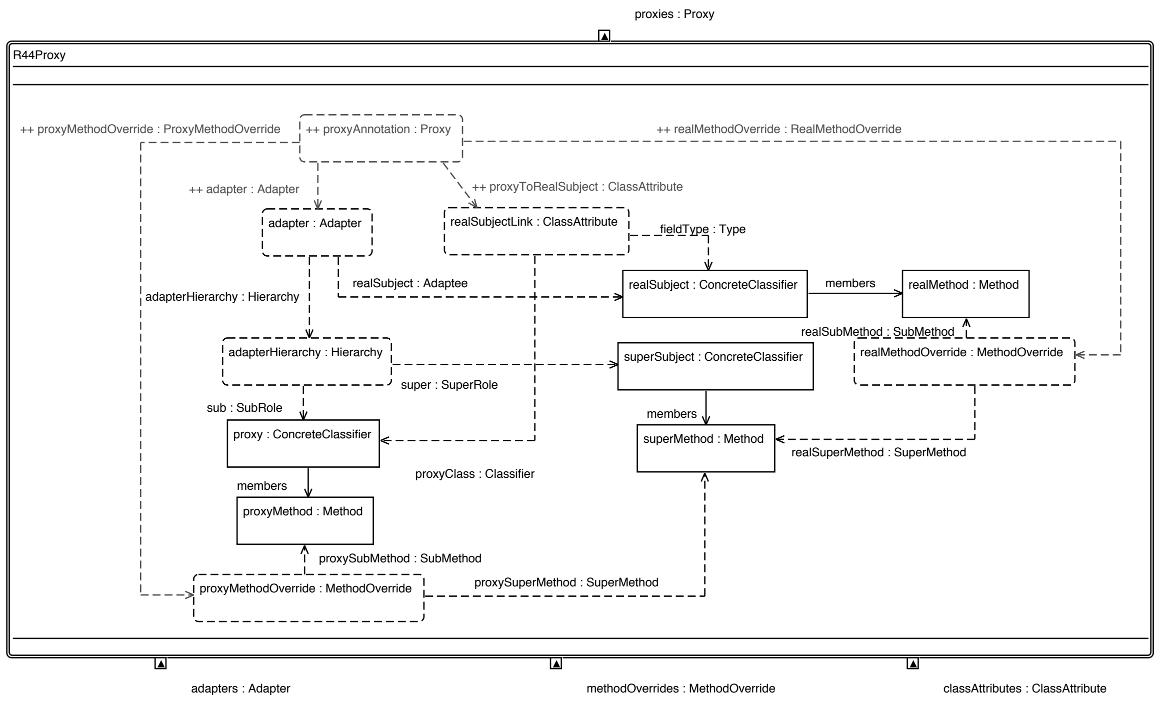

proxies : Proxy
R44Proxy
++ proxyMethodOverride : ProxyMethodOverride
++ proxyAnnotation : Proxy
++ realMethodOverride : RealMethodOverride
++ adapter : Adapter
++ proxyToRealSubject : ClassAttribute
adapter : Adapter
realSubjectLink : ClassAttribute
fieldType : Type
realSubject : Adaptee
realSubject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

Figure F.38.: 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.

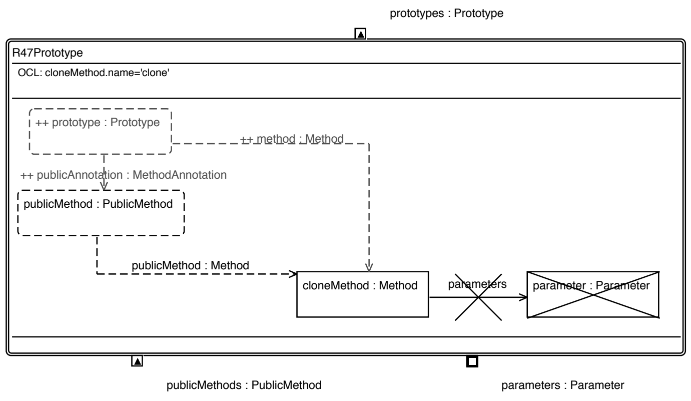


Figure F.39.: A Prototype design pattern exists, if a public method with name 'clone' exists and does not consist of parameters.

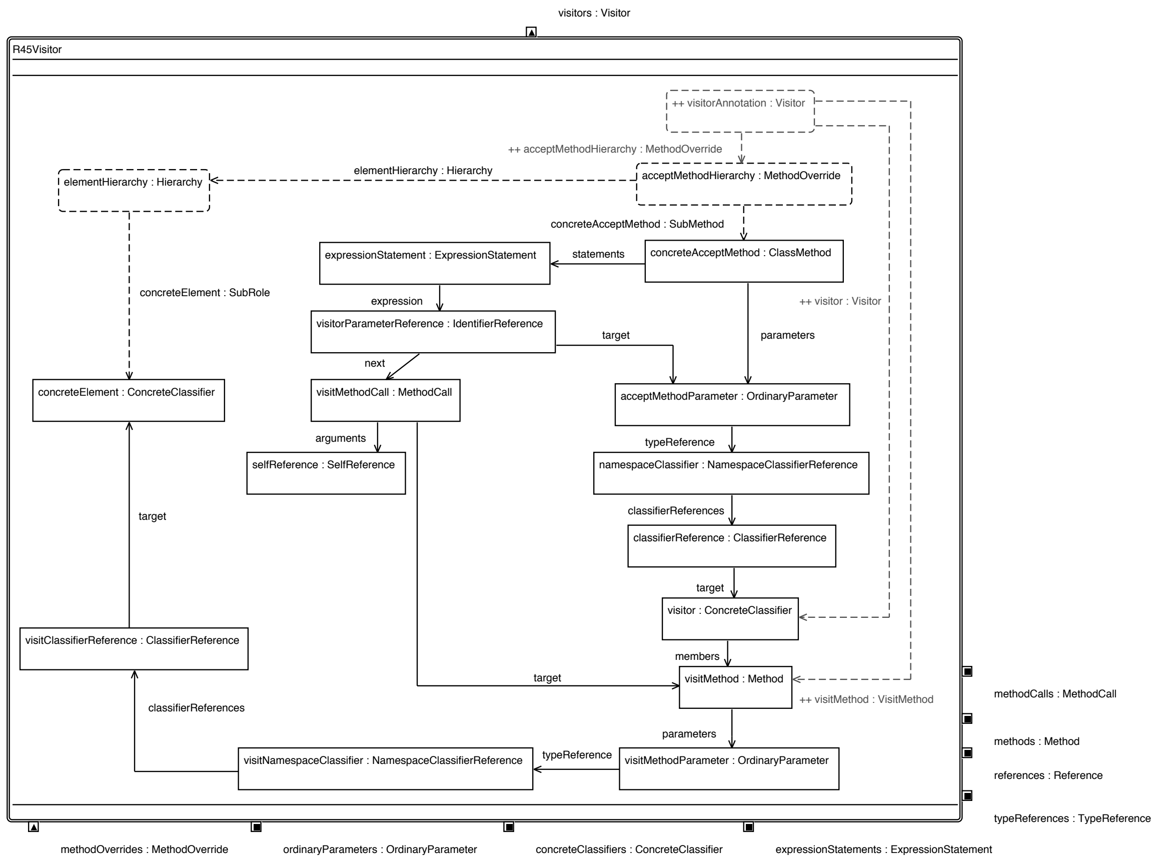

visitors : Visitor
R45Visitor
++ visitorAnnotation : Visitor
++ acceptMethodHierarchy : MethodOverride
elementHierarchy : Hierarchy
elementHierarchy : Hierarchy
acceptMethodHierarchy : MethodOverride
concreteAcceptMethod : SubMethod
expressionStatement : ExpressionStatement
statements
concreteAcceptMethod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

Figure F.40.: 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.

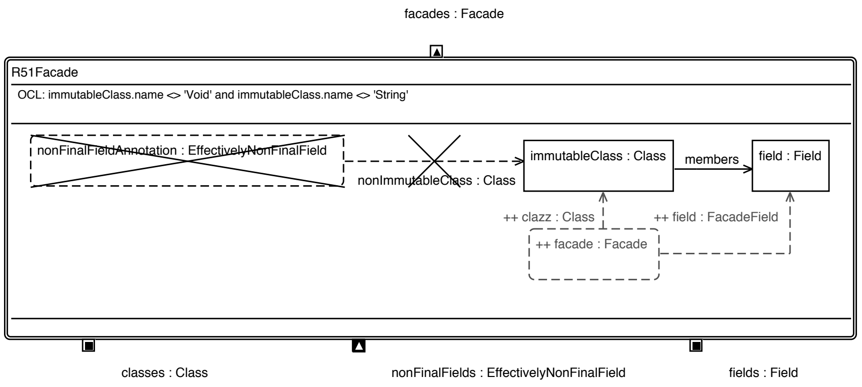

facades : Facade
R51Facade
OCL: immutableClass.name <> 'Void' and immutableClass.name <> 'String'
nonFinalFieldAnnotation : EffectivelyNonFinalField
nonImmutableClass : Class
immutableClass : Class
members
field : Field
++ clazz : Class
++ field : FacadeField
++ facade : Facade
classes : Class
nonFinalFields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

Figure F.41.: A class implements the Facade design pattern, if it does not consist of effectively non final fields.

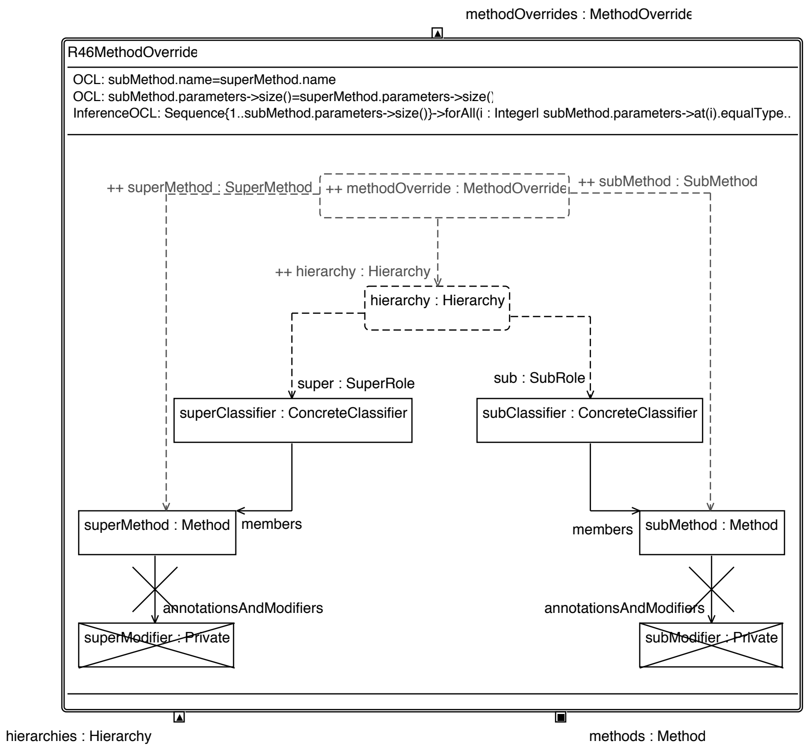

methodOverrides : MethodOverride
R46MethodOverride
OCL: subMethod.name=superMethod.name
OCL: subMethod.parameters->size()=superMethod.parameters->size()
InferenceOCL: Sequence{1..subMethod.parameters->size()}->forAll(i : Integerl subMethod.parameters->at(i).equalType..
++ superMethod : SuperMethod
++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

Figure F.42.: 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.

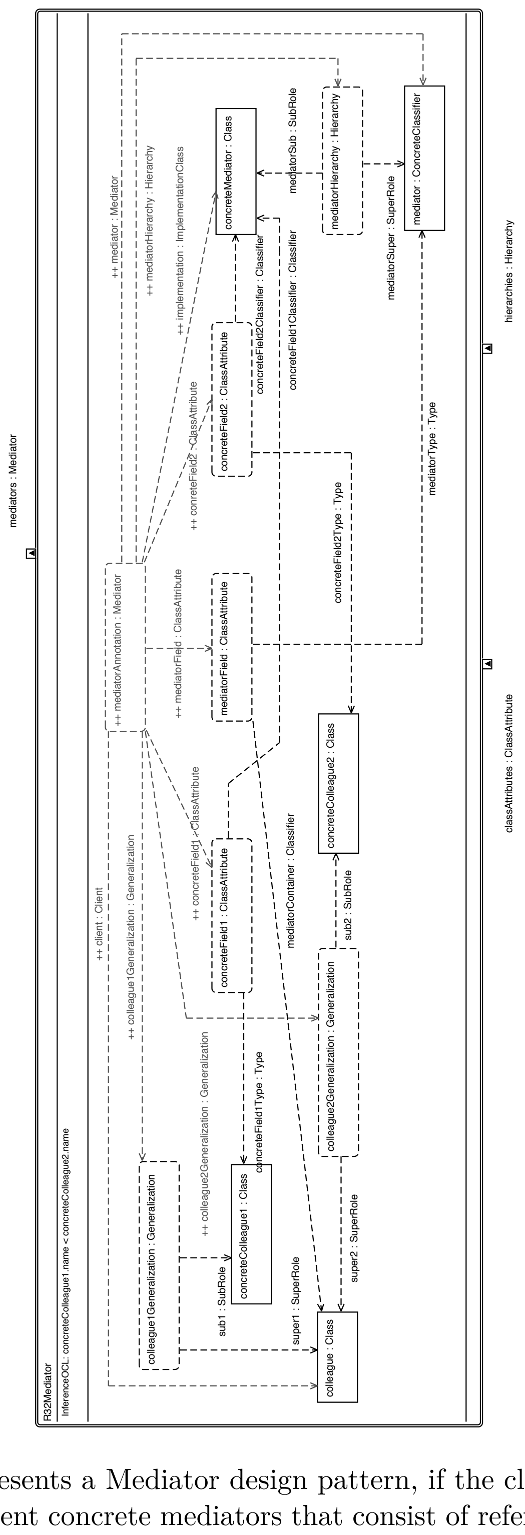
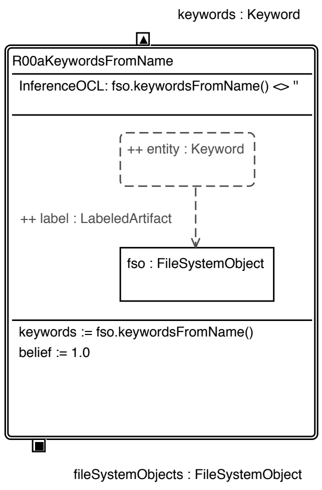

mediators : Mediator
R32Mediator
InferenceOCL: concreteColleague1.name < concreteColleague2.name
++ mediator : Mediator
++ mediatorHierarchy : Hierarchy
++ implementation : ImplementationClass
concreteMediator : Class
mediatorSub : SubRole
mediatorHierarchy : Hierarchy
mediator : ConcreteClassifier
mediatorSuper : SuperRole
hierarchies : Hierarchy
concreteField2Classifier : Classifier
concreteField1Classifier : Classifier
concreteField2 : ClassAttribute
++ conreteField2 : ClassAttribute
mediatorType : Type
concreteField2Type : Type
++ mediatorAnnotation : Mediator
++ mediatorField : ClassAttribute
mediatorField : ClassAttribute
concreteColleague2 : Class
classAttributes : ClassAttribute
++ concreteField1 : ClassAttribute
concreteField1 : ClassAttribute
mediatorContainer : Classifier
sub2 : SubRole
++ client : Client
++ colleague1Generalization : Generalization
colleague2Generalization : Generalization
++ colleague2Generalization : Generalization
concreteField1Type : Type
colleague1Generalization : Generalization
concreteColleague1 : Class
sub1 : SubRole
super1 : SuperRole
super2 : SuperRole
colleague : Class

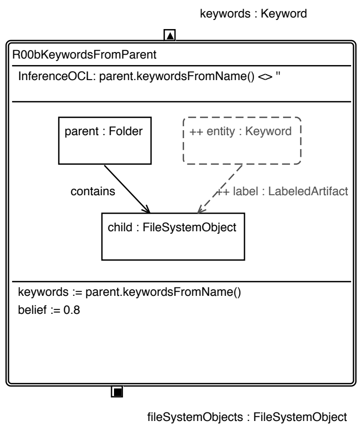
Figure F.43.: A class represents a Mediator design pattern, if the class consists of sub classes that implement concrete mediators that consist of references to colleague classes and each colleague class knows its mediator class.

G. Case Study - Tracing Innovation Projects

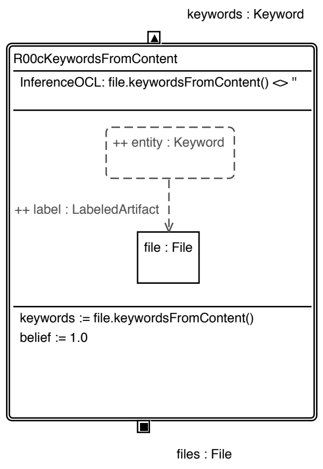
The following figures show view modules for tracing Design Thinking innovation projects.


keywords : Keyword
R00aKeywordsFromName
InferenceOCL: fso.keywordsFromName() <> ''
++ entity : Keyword
++ label : LabeledArtifact
fso : FileSystemObject
keywords := fso.keywordsFromName()
belief := 1.0
fileSystemObjects : FileSystemObject

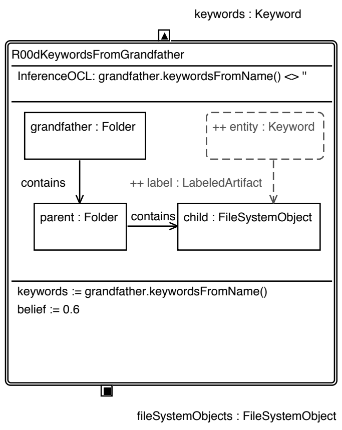
(a) Extracts keywords from file system object names.


keywords : Keyword
R00bKeywordsFromParent
InferenceOCL: parent.keywordsFromName() <> ''
parent : Folder
++ entity : Keyword
contains
++ label : LabeledArtifact
child : FileSystemObject
keywords := parent.keywordsFromName()
belief := 0.8
fileSystemObjects : FileSystemObject

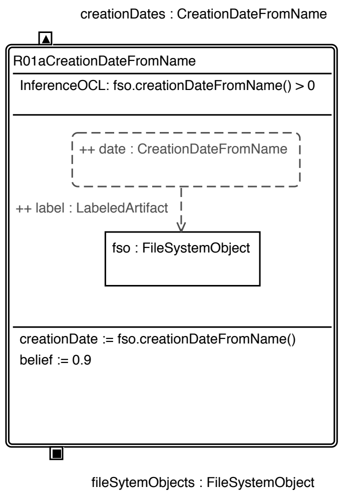
(b) Extracts keywords from the name of the parent-folder.


keywords : Keyword
R00cKeywordsFromContent
InferenceOCL: file.keywordsFromContent() <> ''
++ entity : Keyword
++ label : LabeledArtifact
file : File
keywords := file.keywordsFromContent()
belief := 1.0
files : File

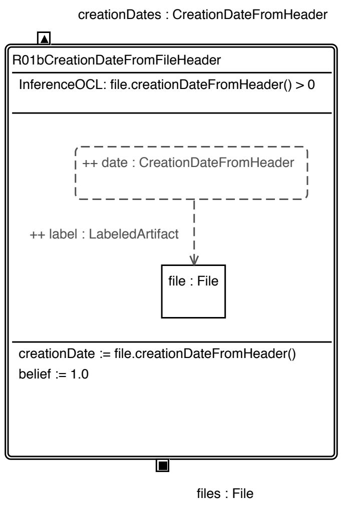
(c) Extracts keywords from the content of a file.


keywords : Keyword
R00dKeywordsFromGrandfather
InferenceOCL: grandfather.keywordsFromName() <> ''
grandfather : Folder
++ entity : Keyword
contains
++ label : LabeledArtifact
parent : Folder
contains
child : FileSystemObject
keywords := grandfather.keywordsFromName()
belief := 0.6
fileSystemObjects : FileSystemObject

(d) Extracts keywords from the grandparent-folder of a file system object.

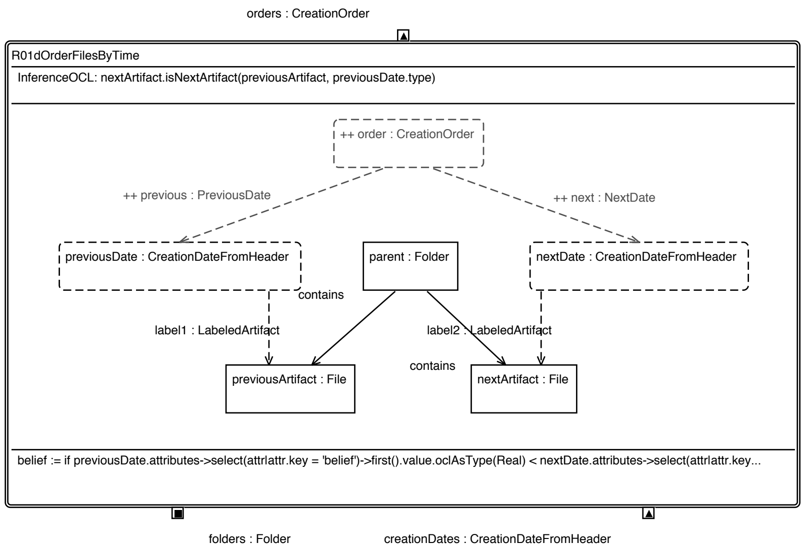

creationDates : CreationDateFromName
R01aCreationDateFromName
InferenceOCL: fso.creationDateFromName() > 0
++ date : CreationDateFromName
++ label : LabeledArtifact
fso : FileSystemObject
creationDate := fso.creationDateFromName()
belief := 0.9
fileSytemObjects : FileSystemObject

(e) Obtains the creation date of a file system object from its file name.

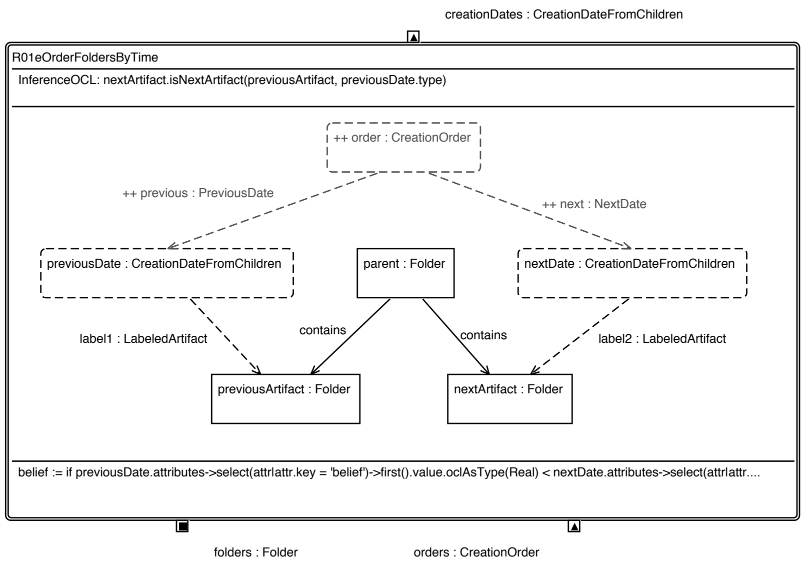

creationDates : CreationDateFromHeader
R01bCreationDateFromFileHeader
InferenceOCL: file.creationDateFromHeader() > 0
++ date : CreationDateFromHeader
++ label : LabeledArtifact
file : File
creationDate := file.creationDateFromHeader()
belief := 1.0
files : File

(f) Extracts the creation date of a file system object from its header.

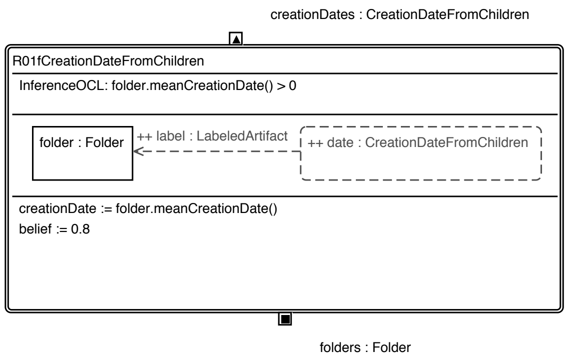
Figure G.44.: Meta data extraction


orders : CreationOrder
R01dOrderFilesByTime
InferenceOCL: nextArtifact.isNextArtifact(previousArtifact, previousDate.type)
++ order : CreationOrder
++ previous : PreviousDate
++ next : NextDate
previousDate : CreationDateFromHeader
parent : Folder
nextDate : CreationDateFromHeader
contains
label1 : LabeledArtifact
label2 : LabeledArtifact
contains
previousArtifact : File
nextArtifact : File
belief := if previousDate.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real) < nextDate.attributes->select(attr|attr.key...
folders : Folder
creationDates : CreationDateFromHeader

(a) Orders files by their creation dates.


creationDates : CreationDateFromChildren
R01eOrderFoldersByTime
InferenceOCL: nextArtifact.isNextArtifact(previousArtifact, previousDate.type)
++ order : CreationOrder
++ previous : PreviousDate
++ next : NextDate
previousDate : CreationDateFromChildren
parent : Folder
nextDate : CreationDateFromChildren
label1 : LabeledArtifact
contains
contains
label2 : LabeledArtifact
previousArtifact : Folder
nextArtifact : Folder
belief := if previousDate.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real) < nextDate.attributes->select(attr|attr....
folders : Folder
orders : CreationOrder

(b) Orders folders by their creation dates.


creationDates : CreationDateFromChildren
R01fCreationDateFromChildren
InferenceOCL: folder.meanCreationDate() > 0
folder : Folder
++ label : LabeledArtifact
++ date : CreationDateFromChildren
creationDate := folder.meanCreationDate()
belief := 0.8
folders : Folder

(c) Gives a folder a creation date by calculating the mean of creation dates from contained files.

Figure G.45.: Creation order

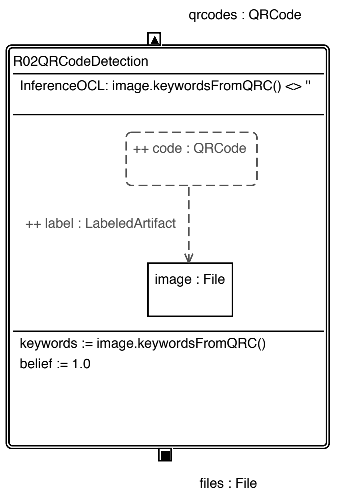
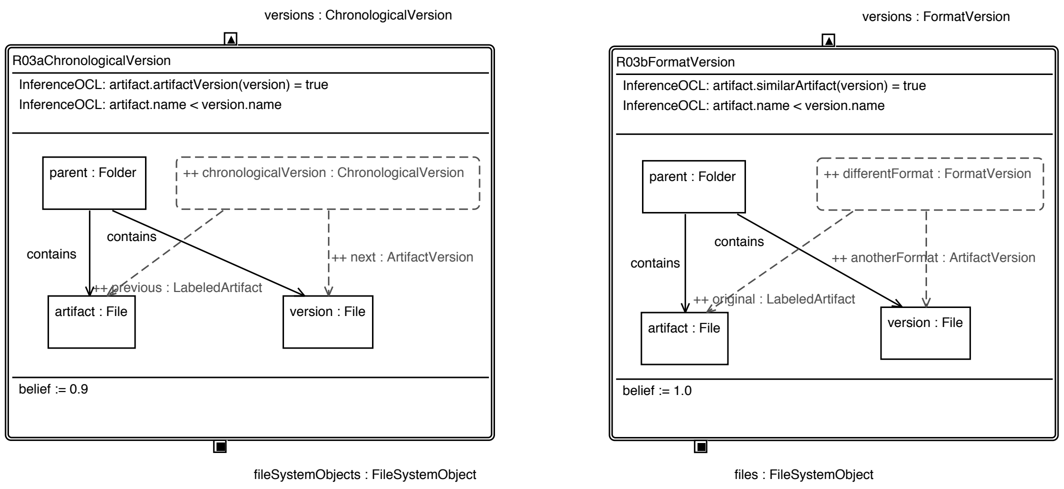

qrcodes : QRCode
R02QRCodeDetection
InferenceOCL: image.keywordsFromQRC() <> ''
++ code : QRCode
++ label : LabeledArtifact
image : File
keywords := image.keywordsFromQRC()
belief := 1.0
files : File

Figure G.46.: Detects a given QR-Code and extracts keywords from the decoded text.


versions : ChronologicalVersion
R03aChronologicalVersion
InferenceOCL: artifact.artifactVersion(version) = true
InferenceOCL: artifact.name < version.name
parent : Folder
++ chronologicalVersion : ChronologicalVersion
contains
contains
++ next : ArtifactVersion
++ previous : LabeledArtifact
artifact : File
version : File
belief := 0.9
fileSystemObjects : FileSystemObject
versions : FormatVersion
R03bFormatVersion
InferenceOCL: artifact.similarArtifact(version) = true
InferenceOCL: artifact.name < version.name
parent : Folder
++ differentFormat : FormatVersion
contains
contains
++ anotherFormat : ArtifactVersion
++ original : LabeledArtifact
artifact : File
version : File
belief := 1.0
files : FileSystemObject

(a) Determines chronological versions of a file in the same folder.

(b) Detects files with the same name, but different file extensions.

Figure G.47.: Versions of artifacts

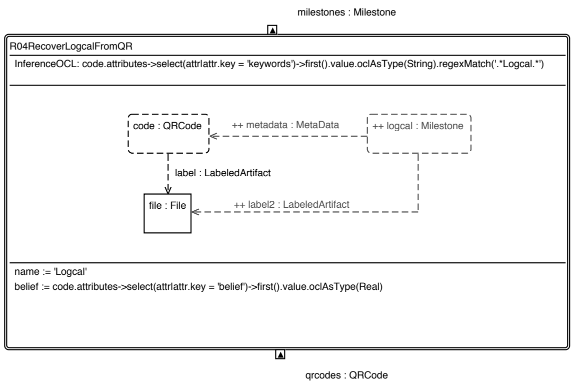
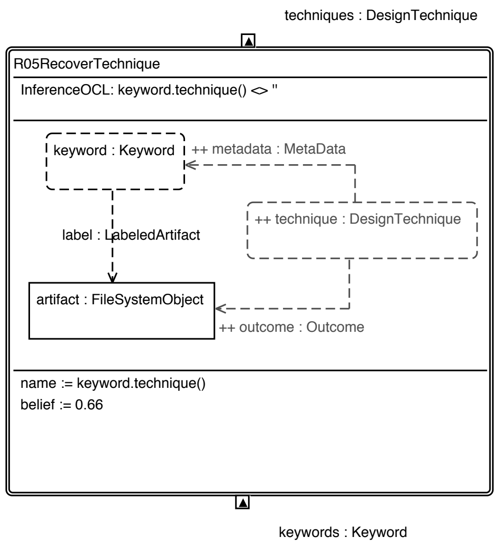
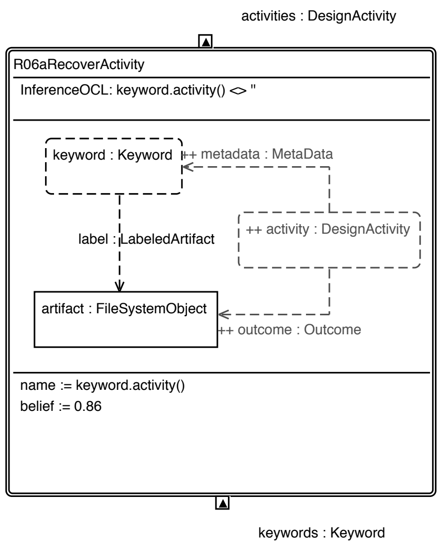

milestones : Milestone
R04RecoverLogcalFromQR
InferenceOCL: code.attributes->select(attr|attr.key = 'keywords')->first().value.oclAsType(String).regexMatch('.*Logcal.*')
code : QRCode
++ metadata : MetaData
++ logcal : Milestone
label : LabeledArtifact
file : File
++ label2 : LabeledArtifact
name := 'Logcal'
belief := code.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real)
qrcodes : QRCode

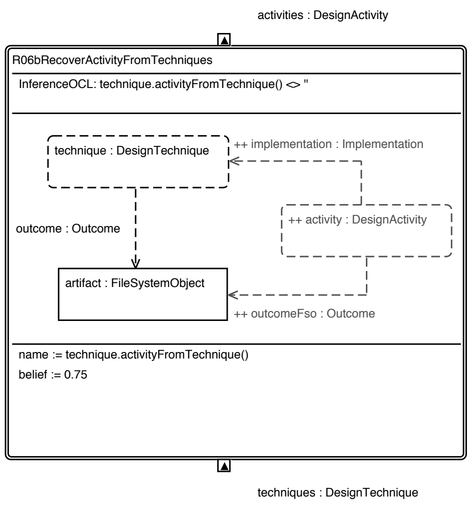
Figure G.48.: Detects the LogCal by processing found QR codes.


techniques : DesignTechnique
R05RecoverTechnique
InferenceOCL: keyword.technique() <> ''
keyword : Keyword
++ metadata : MetaData
++ technique : DesignTechnique
label : LabeledArtifact
artifact : FileSystemObject
++ outcome : Outcome
name := keyword.technique()
belief := 0.66
keywords : Keyword

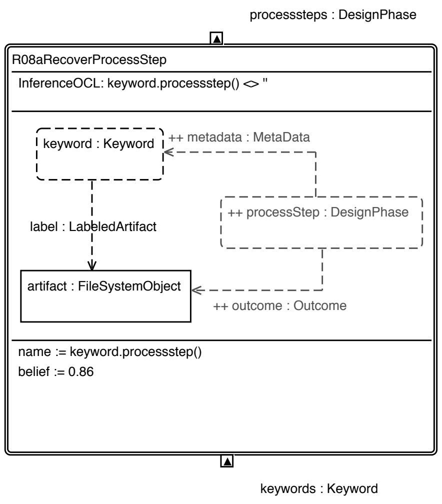
(a) Extracts the technique from keywords.


activities : DesignActivity
R06aRecoverActivity
InferenceOCL: keyword.activity() <> ''
keyword : Keyword
++ metadata : MetaData
++ activity : DesignActivity
label : LabeledArtifact
artifact : FileSystemObject
++ outcome : Outcome
name := keyword.activity()
belief := 0.86
keywords : Keyword

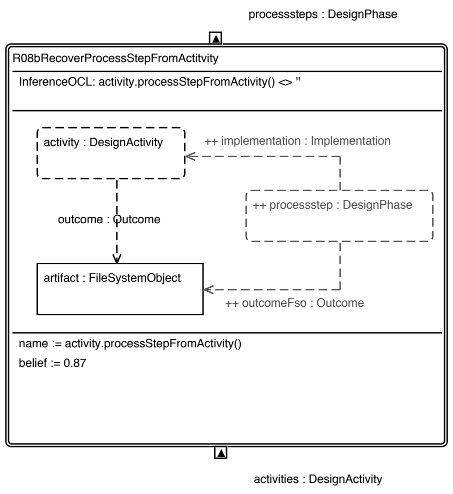
(b) Extracts the activity from keywords.


activities : DesignActivity
R06bRecoverActivityFromTechniques
InferenceOCL: technique.activityFromTechnique() <> ''
technique : DesignTechnique
++ implementation : Implementation
++ activity : DesignActivity
outcome : Outcome
artifact : FileSystemObject
++ outcomeFso : Outcome
name := technique.activityFromTechnique()
belief := 0.75
techniques : DesignTechnique

(c) Extracts the activity from a technique.

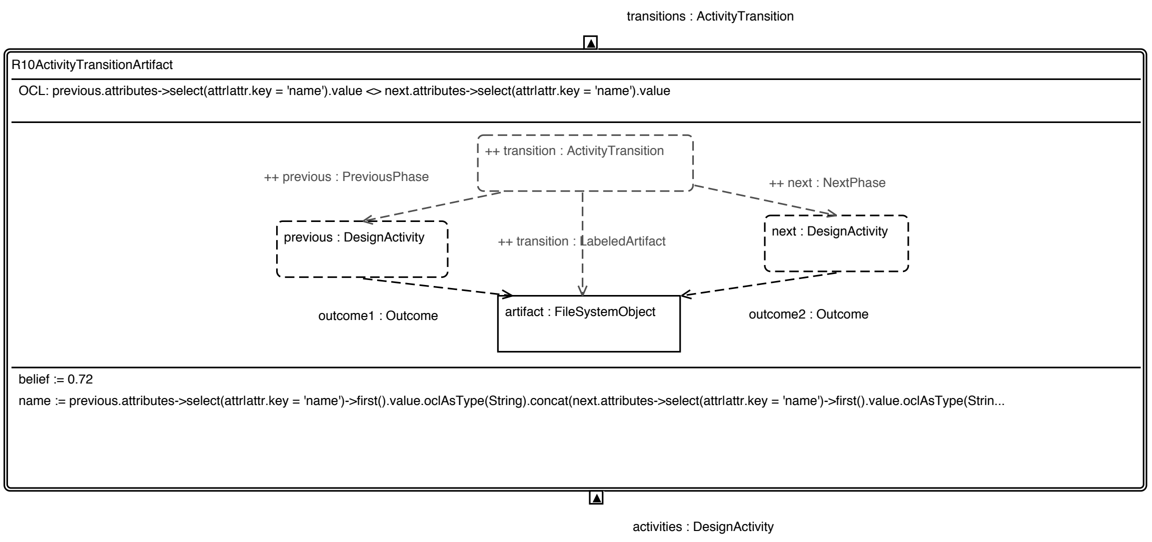

processsteps : DesignPhase
R08aRecoverProcessStep
InferenceOCL: keyword.processstep() <> ''
keyword : Keyword
++ metadata : MetaData
++ processStep : DesignPhase
label : LabeledArtifact
artifact : FileSystemObject
++ outcome : Outcome
name := keyword.processstep()
belief := 0.86
keywords : Keyword

(d) Extracts the phase from keywords.


processsteps : DesignPhase
R08bRecoverProcessStepFromActitvity
InferenceOCL: activity.processStepFromActivity() <> ''
activity : DesignActivity
++ implementation : Implementation
++ processstep : DesignPhase
outcome : Outcome
artifact : FileSystemObject
++ outcomeFso : Outcome
name := activity.processStepFromActivity()
belief := 0.87
activities : DesignActivity

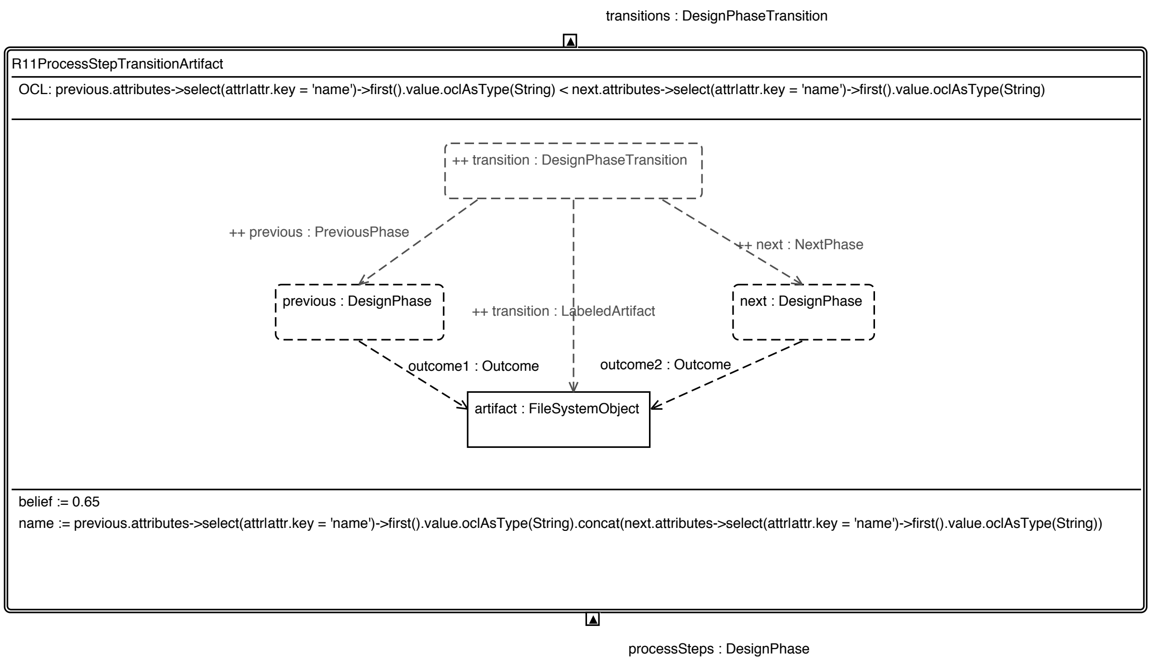
(e) Extracts the phase from an activity.

Figure G.49.: Design techniques, design activities, design phases


transitions : ActivityTransition
R10ActivityTransitionArtifact
++ transition : ActivityTransition
++ previous : PreviousPhase
++ next : NextPhase
previous : DesignActivity
++ transition : LabeledArtifact
next : DesignActivity
outcome1 : Outcome
artifact : FileSystemObject
outcome2 : Outcome
activities : DesignActivity

```
OCL: previous.attributes->select(attr|attr.key = 'name').value <> next.attributes->select(attr|attr.key = 'name').value
belief := 0.72
name := previous.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String).concat(next.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(Strin...
```

(a) Detects design artifacts that are assigned to two different design activities and, therefore, are transition artifact.


transitions : DesignPhaseTransition
R11ProcessStepTransitionArtifact
++ transition : DesignPhaseTransition
++ previous : PreviousPhase
++ next : NextPhase
previous : DesignPhase
++ transition : LabeledArtifact
next : DesignPhase
outcome1 : Outcome
outcome2 : Outcome
artifact : FileSystemObject
processSteps : DesignPhase

```
OCL: previous.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String) < next.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
belief := 0.65
name := previous.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String).concat(next.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String))
```

(b) Detects design artifacts that are assigned to two different design phases and, therefore, are transition artifacts.

Figure G.50.: Transition artifacts

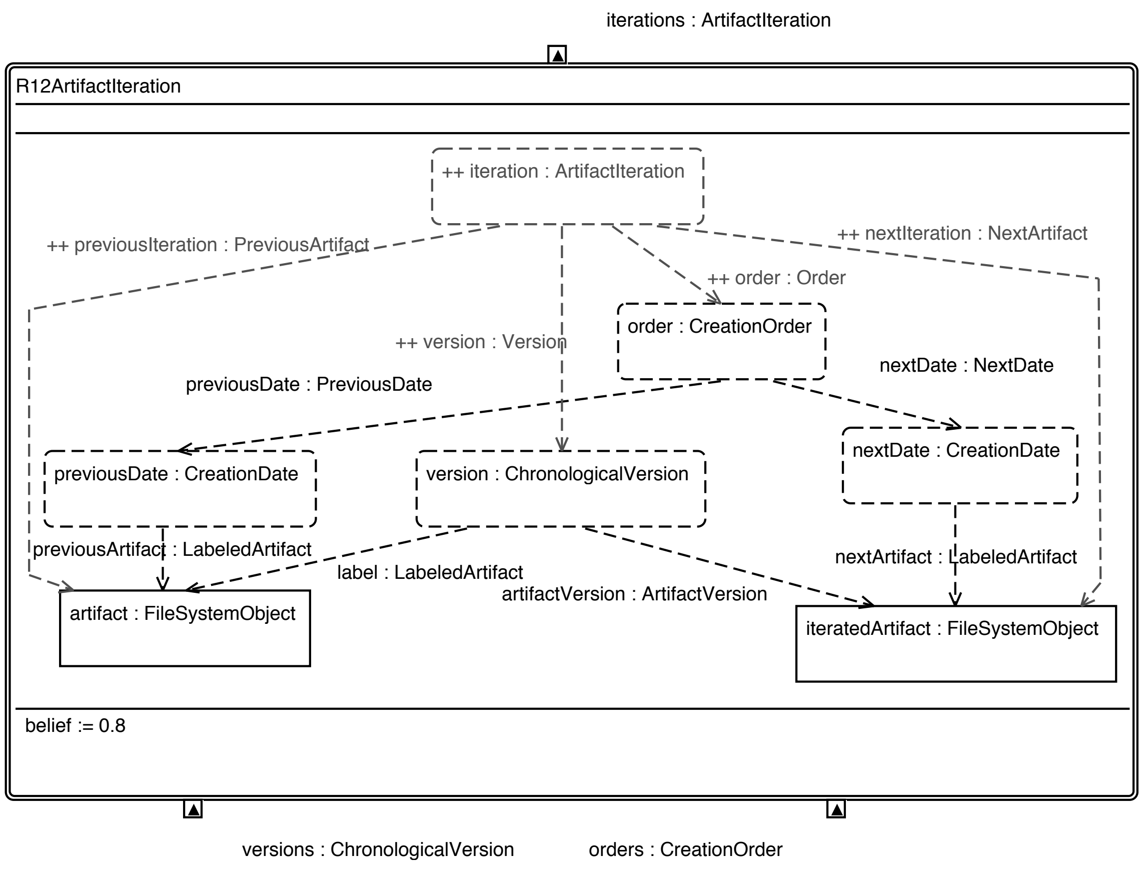
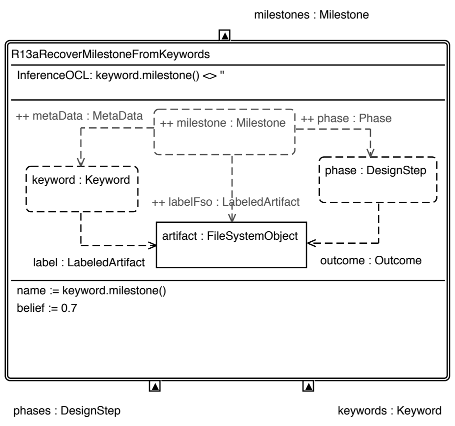
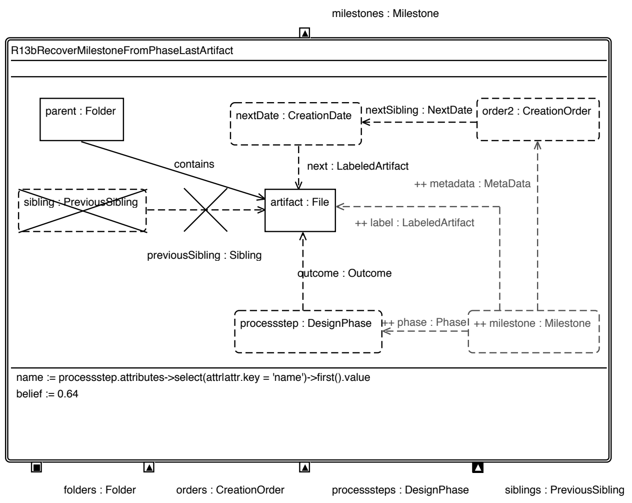

iterations : ArtifactIteration
R12ArtifactIteration
++ iteration : ArtifactIteration
++ previousIteration : PreviousArtifact
++ nextIteration : NextArtifact
++ order : Order
order : CreationOrder
++ version : Version
nextDate : NextDate
previousDate : PreviousDate
previousDate : CreationDate
version : ChronologicalVersion
nextDate : CreationDate
previousArtifact : LabeledArtifact
label : LabeledArtifact
nextArtifact : LabeledArtifact
artifactVersion : ArtifactVersion
artifact : FileSystemObject
iteratedArtifact : FileSystemObject
belief := 0.8
versions : ChronologicalVersion
orders : CreationOrder

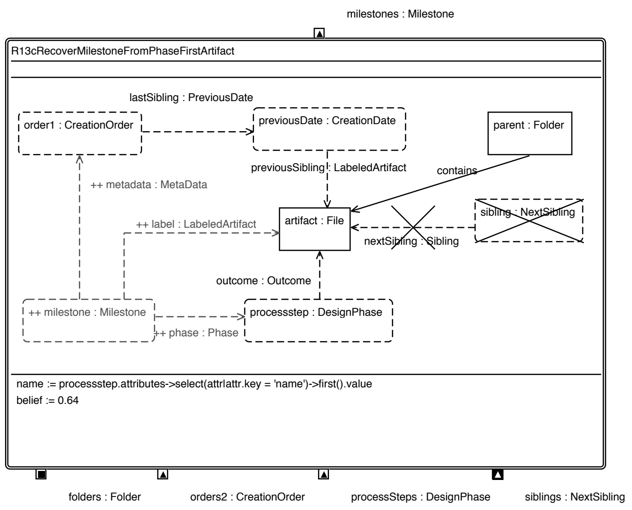
Figure G.51.: Detects iterations of artifacts, if two artifacts are in the same creation- and chronological order.



(a) Detects milestones by analyzing extracted keywords.

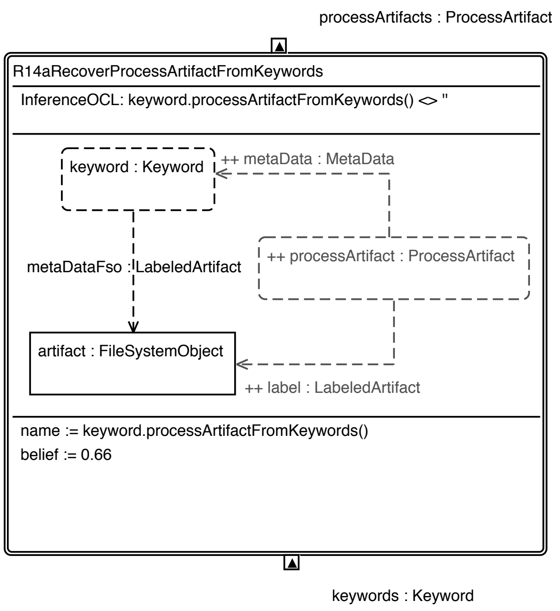


(b) Marks a file system object as milestone, if it is the last artifact in the current design phase.

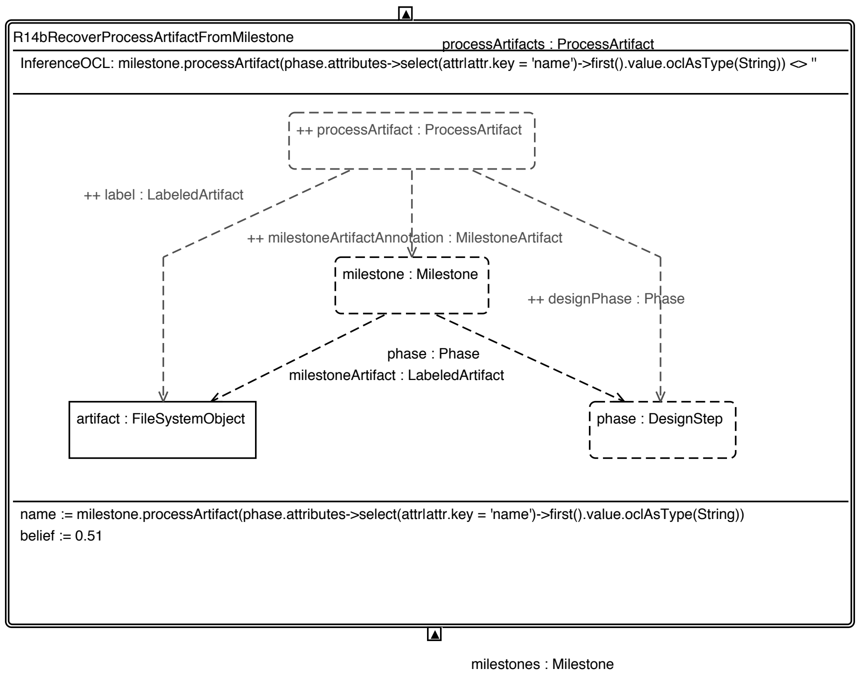

milestones : Milestone
R13cRecoverMilestoneFromPhaseFirstArtifact
lastSibling : PreviousDate
order1 : CreationOrder
previousDate : CreationDate
parent : Folder
previousSibling : LabeledArtifact
contains
++ metadata : MetaData
sibling : NextSibling
++ label : LabeledArtifact
artifact : File
nextSibling : Sibling
outcome : Outcome
++ milestone : Milestone
processstep : DesignPhase
++ phase : Phase
name := processstep.attributes->select(attr|attr.key = 'name')->first().value
belief := 0.64
folders : Folder
orders2 : CreationOrder
processSteps : DesignPhase
siblings : NextSibling

(c) Marks a file system object as milestone, if it is the first artifact of the current design phase.

Figure G.52.: Milestones

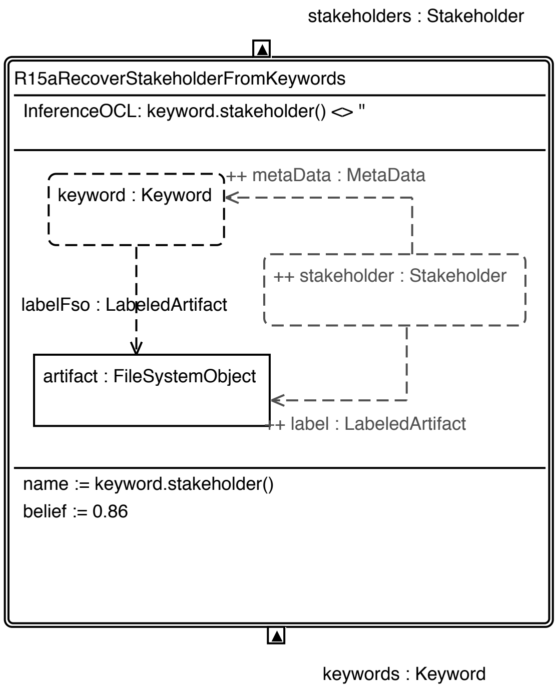

processArtifacts : ProcessArtifact
R14aRecoverProcessArtifactFromKeywords
InferenceOCL: keyword.processArtifactFromKeywords() <> ''
keyword : Keyword
++ metaData : MetaData
++ processArtifact : ProcessArtifact
metaDataFso : LabeledArtifact
artifact : FileSystemObject
++ label : LabeledArtifact
name := keyword.processArtifactFromKeywords()
belief := 0.66
keywords : Keyword

(a) Detects process artifacts from keywords.

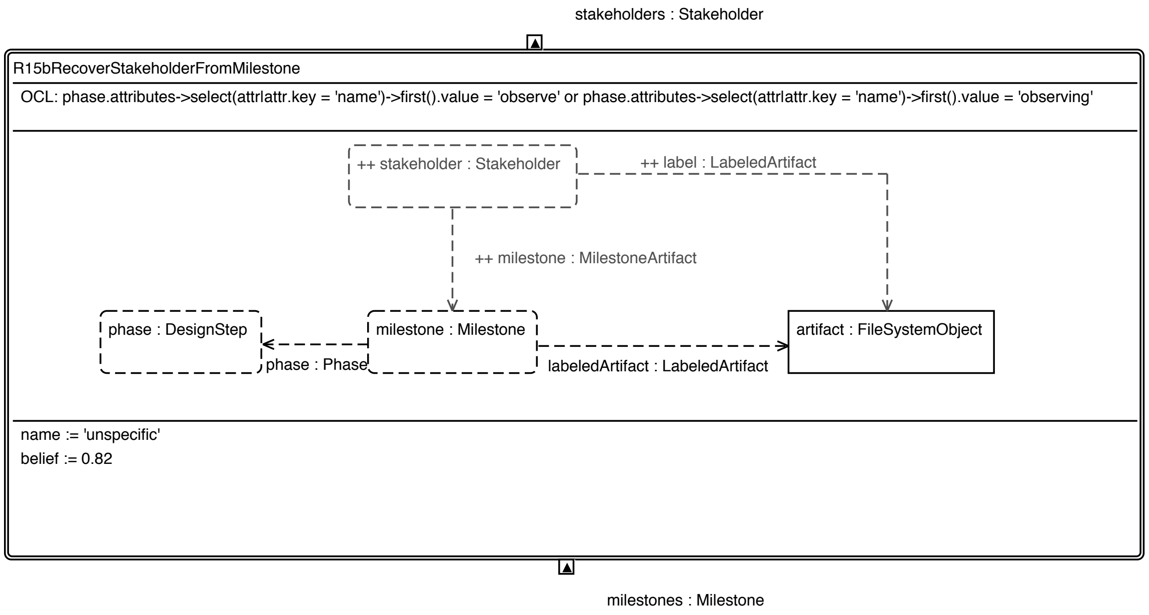

R14bRecoverProcessArtifactFromMilestone
processArtifacts : ProcessArtifact
InferenceOCL: milestone.processArtifact(phase.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)) <> ''
++ processArtifact : ProcessArtifact
++ label : LabeledArtifact
++ milestoneArtifactAnnotation : MilestoneArtifact
milestone : Milestone
++ designPhase : Phase
phase : Phase
milestoneArtifact : LabeledArtifact
artifact : FileSystemObject
phase : DesignStep
name := milestone.processArtifact(phase.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String))
belief := 0.51
milestones : Milestone

(b) Detects process artifacts, if the milestone indicates a process artifact.

Figure G.53.: Process artifacts

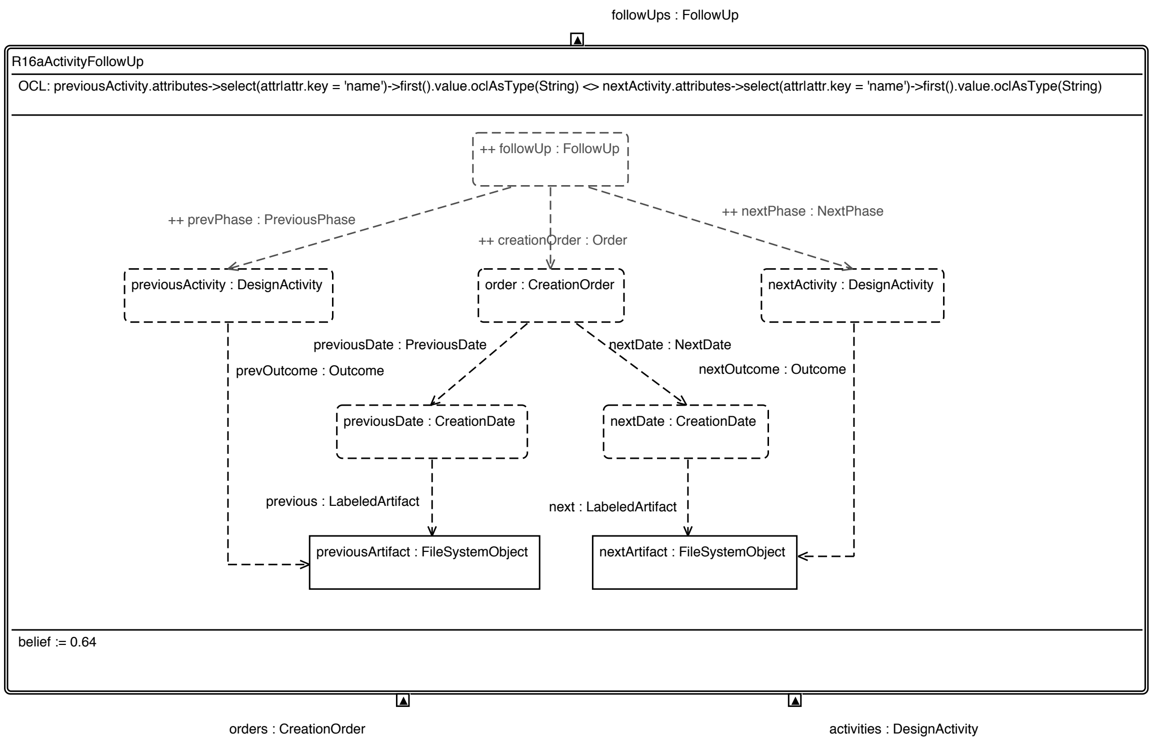

stakeholders : Stakeholder
R15aRecoverStakeholderFromKeywords
InferenceOCL: keyword.stakeholder() <> ''
keyword : Keyword
++ metaData : MetaData
++ stakeholder : Stakeholder
labelFso : LabeledArtifact
artifact : FileSystemObject
++ label : LabeledArtifact
name := keyword.stakeholder()
belief := 0.86
keywords : Keyword

(a) Recovers stakeholders from keywords.

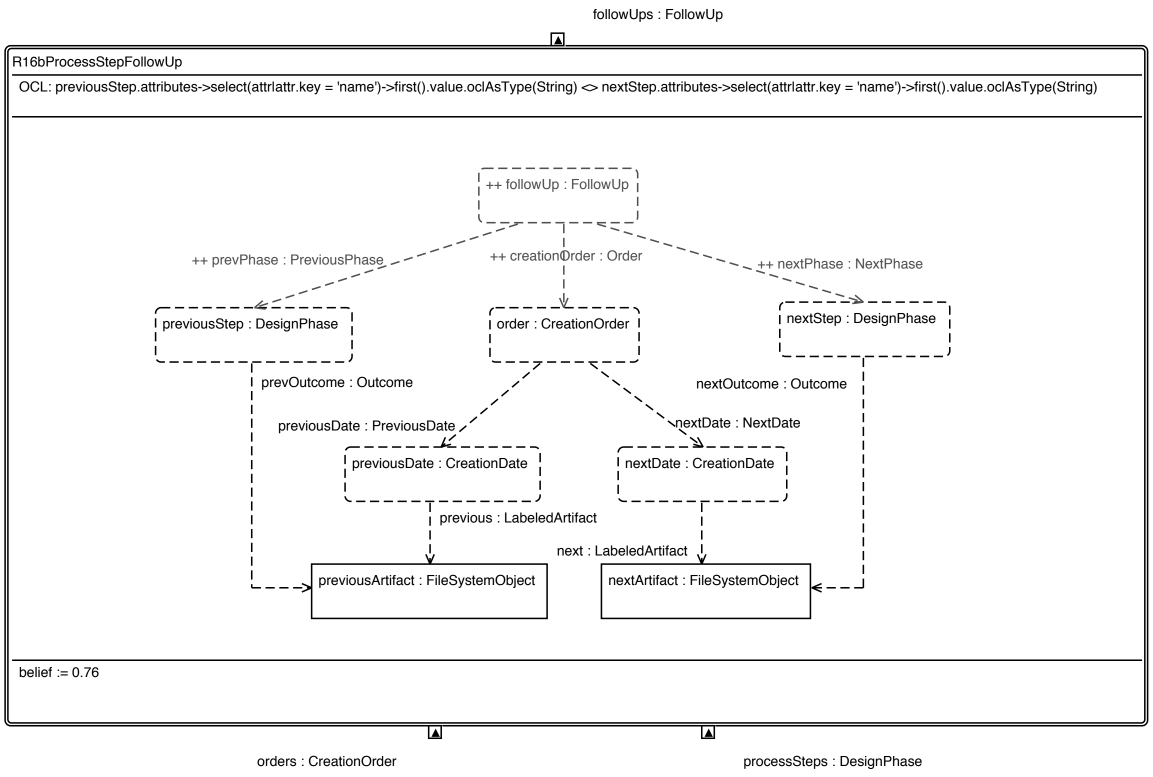

stakeholders : Stakeholder
R15bRecoverStakeholderFromMilestone
OCL: phase.attributes->select(attr|attr.key = 'name')->first().value = 'observe' or phase.attributes->select(attr|attr.key = 'name')->first().value = 'observing'
++ stakeholder : Stakeholder
++ label : LabeledArtifact
++ milestone : MilestoneArtifact
phase : DesignStep
milestone : Milestone
artifact : FileSystemObject
phase : Phase
labeledArtifact : LabeledArtifact
name := 'unspecific'
belief := 0.82
milestones : Milestone

(b) Recovers stakeholders from milestones.

Figure G.54.: Stakeholder

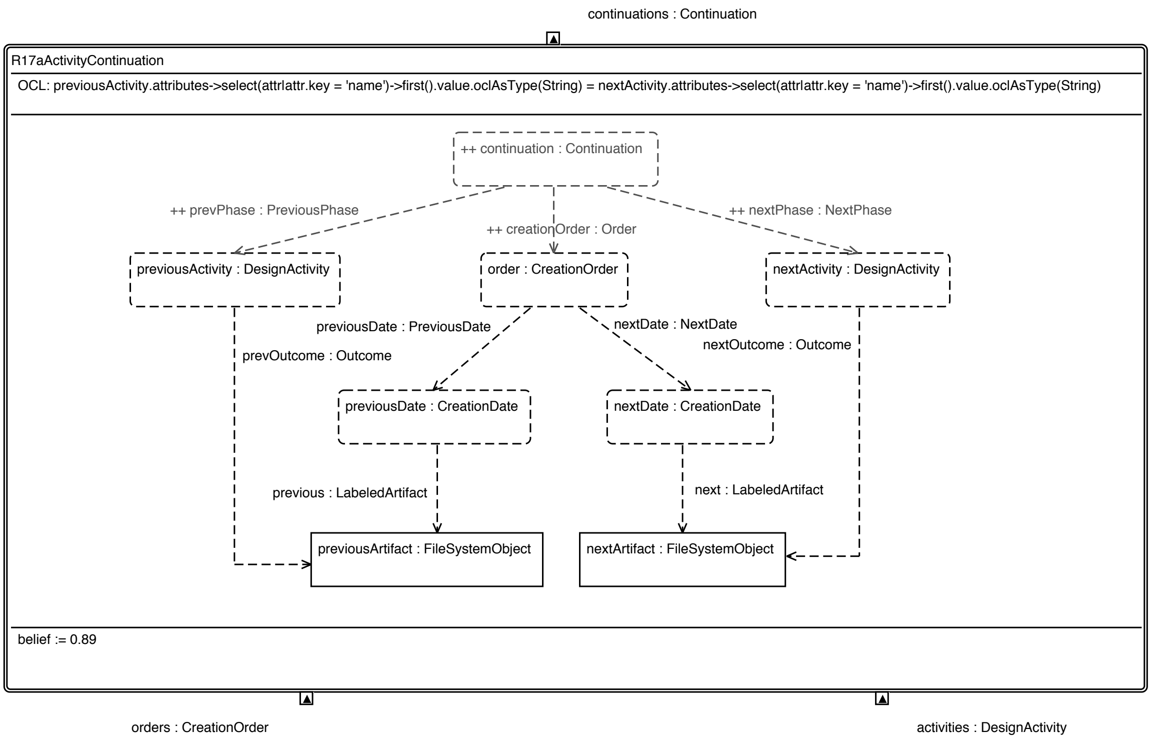

followUps : FollowUp
R16aActivityFollowUp
OCL: previousActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String) <> nextActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
++ followUp : FollowUp
++ prevPhase : PreviousPhase
++ creationOrder : Order
++ nextPhase : NextPhase
previousActivity : DesignActivity
order : CreationOrder
nextActivity : DesignActivity
previousDate : PreviousDate
nextDate : NextDate
prevOutcome : Outcome
nextOutcome : Outcome
previousDate : CreationDate
nextDate : CreationDate
previous : LabeledArtifact
next : LabeledArtifact
previousArtifact : FileSystemObject
nextArtifact : FileSystemObject
belief := 0.64
orders : CreationOrder
activities : DesignActivity

(a) Detects design activity follow-ups, if a design activity differs from the previous one.

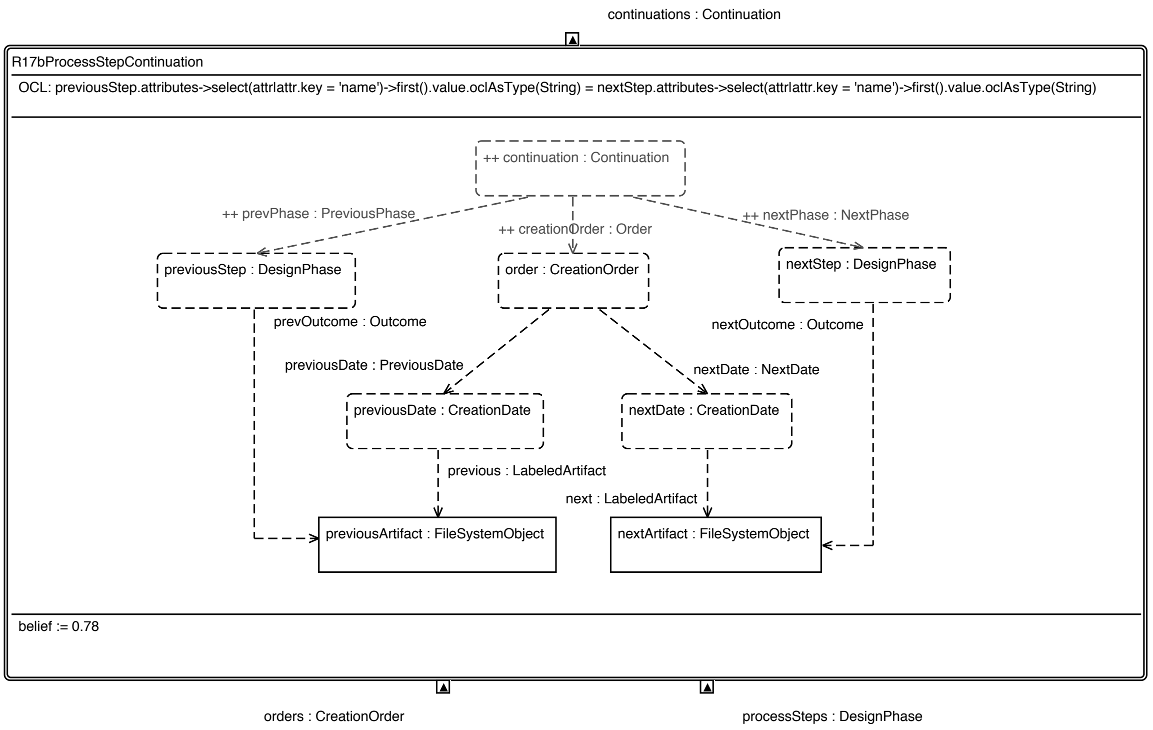

followUps : FollowUp
R16bProcessStepFollowUp
OCL: previousStep.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String) <> nextStep.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
++ followUp : FollowUp
++ prevPhase : PreviousPhase
++ creationOrder : Order
++ nextPhase : NextPhase
previousStep : DesignPhase
order : CreationOrder
nextStep : DesignPhase
prevOutcome : Outcome
nextOutcome : Outcome
previousDate : PreviousDate
nextDate : NextDate
previousDate : CreationDate
nextDate : CreationDate
previous : LabeledArtifact
next : LabeledArtifact
previousArtifact : FileSystemObject
nextArtifact : FileSystemObject
belief := 0.76
orders : CreationOrder
processSteps : DesignPhase

(b) Detects design phase follow-ups, if a design phase differs from the previous one.

Figure G.55.: Follow-ups

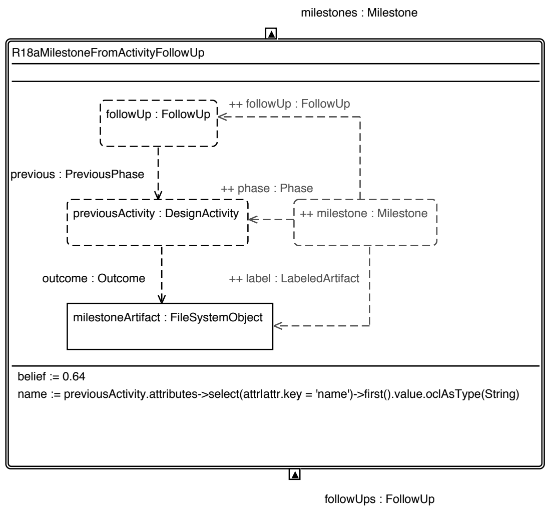

continuations : Continuation
R17aActivityContinuation
OCL: previousActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String) = nextActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
++ continuation : Continuation
++ prevPhase : PreviousPhase
++ creationOrder : Order
++ nextPhase : NextPhase
previousActivity : DesignActivity
order : CreationOrder
nextActivity : DesignActivity
previousDate : PreviousDate
nextDate : NextDate
prevOutcome : Outcome
nextOutcome : Outcome
previousDate : CreationDate
nextDate : CreationDate
previous : LabeledArtifact
next : LabeledArtifact
previousArtifact : FileSystemObject
nextArtifact : FileSystemObject
belief := 0.89
orders : CreationOrder
activities : DesignActivity

(a) Detects design activity continuations, if a design activity does not differ from the previous design activity.

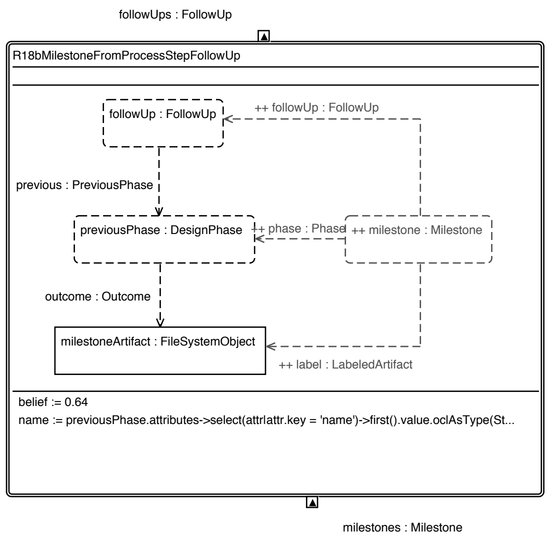

continuations : Continuation
R17bProcessStepContinuation
OCL: previousStep.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String) = nextStep.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
++ continuation : Continuation
++ prevPhase : PreviousPhase
++ creationOrder : Order
++ nextPhase : NextPhase
previousStep : DesignPhase
order : CreationOrder
nextStep : DesignPhase
prevOutcome : Outcome
nextOutcome : Outcome
previousDate : PreviousDate
nextDate : NextDate
previousDate : CreationDate
nextDate : CreationDate
previous : LabeledArtifact
next : LabeledArtifact
previousArtifact : FileSystemObject
nextArtifact : FileSystemObject
belief := 0.78
orders : CreationOrder
processSteps : DesignPhase

(b) Detects design phase continuations, if a design phase does not differ from the previous design phase.

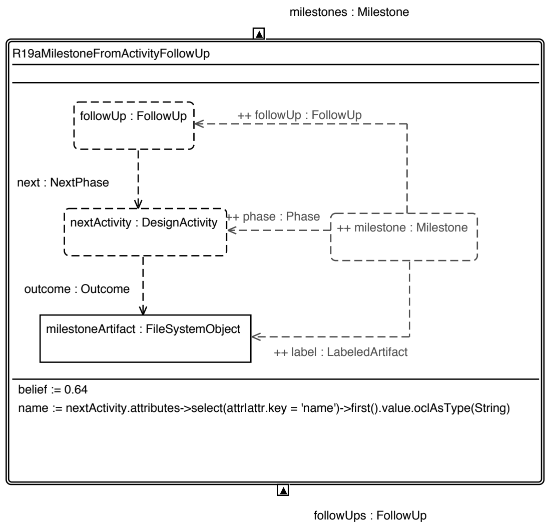
Figure G.56.: Continuation


milestones : Milestone
R18aMilestoneFromActivityFollowUp
followUp : FollowUp
++ followUp : FollowUp
previous : PreviousPhase
++ phase : Phase
previousActivity : DesignActivity
++ milestone : Milestone
outcome : Outcome
++ label : LabeledArtifact
milestoneArtifact : FileSystemObject
belief := 0.64
name := previousActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
followUps : FollowUp

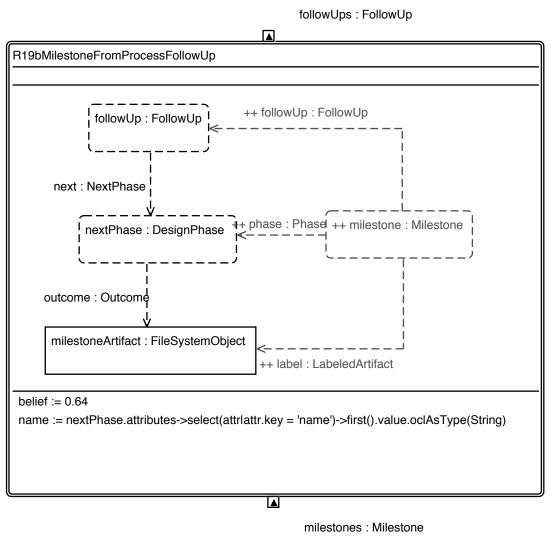
(a) If a design activity and its follow up are found, the last design artifact of the previous design activity is marked as milestone.


followUps : FollowUp
R18bMilestoneFromProcessStepFollowUp
followUp : FollowUp
++ followUp : FollowUp
previous : PreviousPhase
previousPhase : DesignPhase
++ phase : Phase
++ milestone : Milestone
outcome : Outcome
milestoneArtifact : FileSystemObject
++ label : LabeledArtifact
belief := 0.64
name := previousPhase.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(St...
milestones : Milestone

(b) If a design phase and its follow up are found, the last design artifact of the previous design phase is marked as milestone.

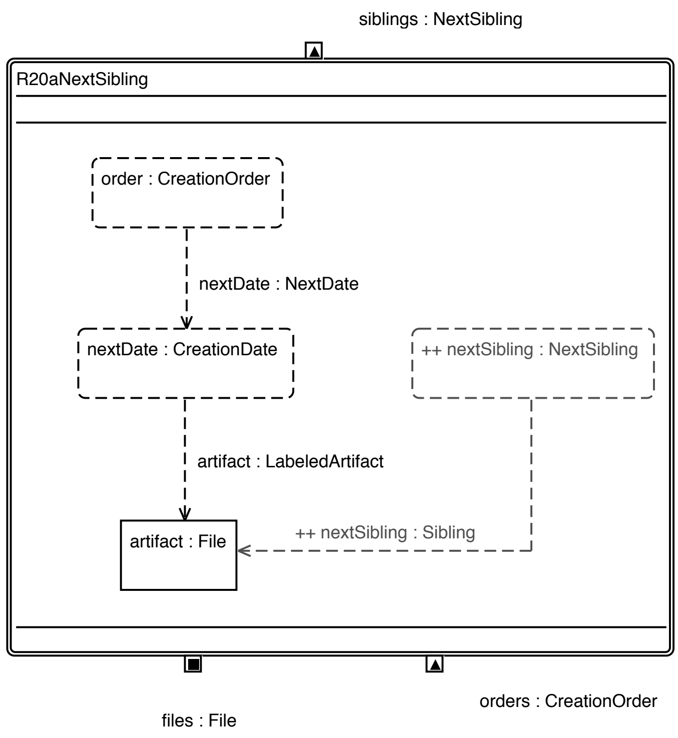

milestones : Milestone
R19aMilestoneFromActivityFollowUp
followUp : FollowUp
++ followUp : FollowUp
next : NextPhase
nextActivity : DesignActivity
++ phase : Phase
++ milestone : Milestone
outcome : Outcome
milestoneArtifact : FileSystemObject
++ label : LabeledArtifact
belief := 0.64
name := nextActivity.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
followUps : FollowUp

(c) If a design activity and its follow up are found, the first design artifact of the current design activity is marked as milestone.

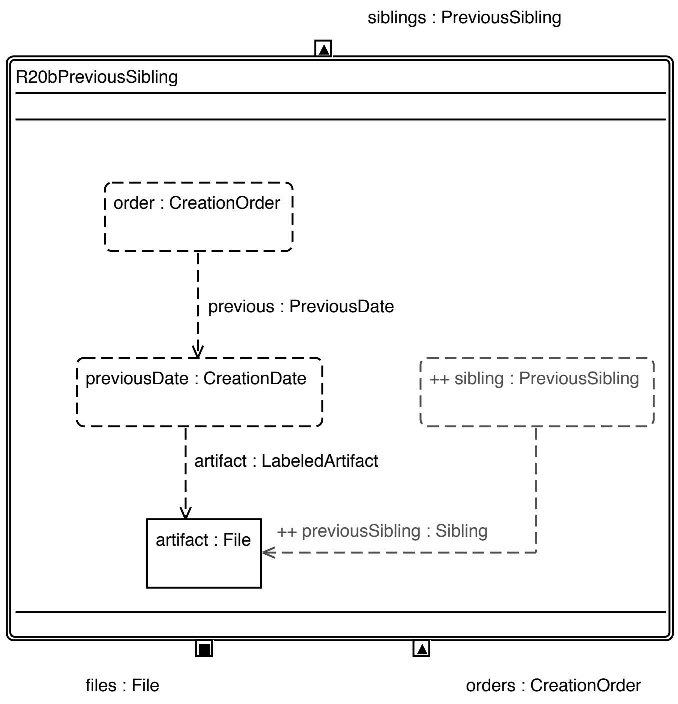


(d) If a design phase and its follow up are found, the first design artifact of the current design phase is marked as milestone.

Figure G.57.: Milestone

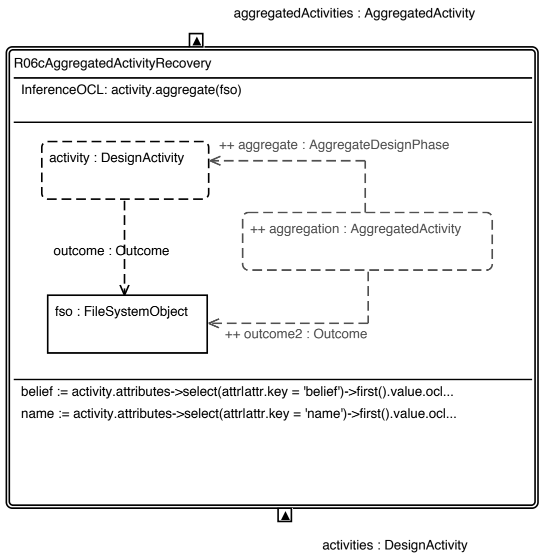


(a) Marks files as successor artifacts in creation order.

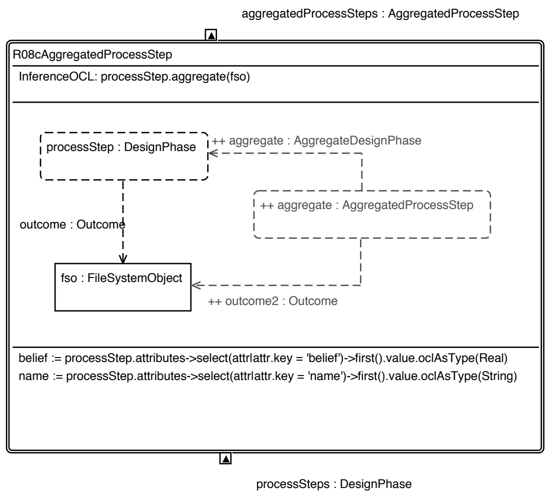

siblings : PreviousSibling
R20bPreviousSibling
order : CreationOrder
previous : PreviousDate
previousDate : CreationDate
++ sibling : PreviousSibling
artifact : LabeledArtifact
artifact : File
++ previousSibling : Sibling
files : File
orders : CreationOrder

(b) Marks files as predecessor artifacts in creation order.

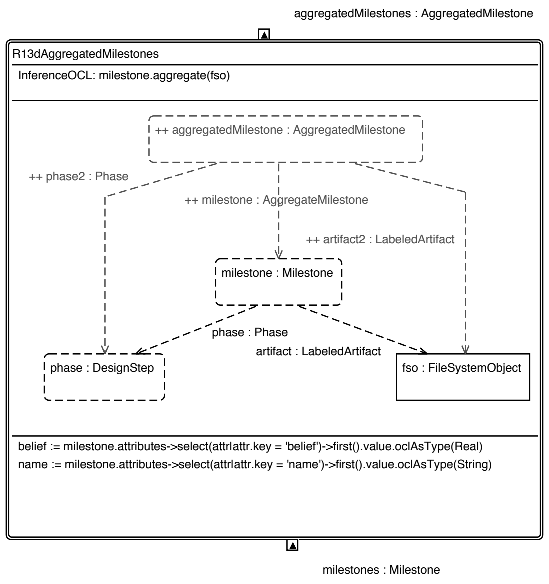
Figure G.58.: Creation order siblings


aggregatedActivities : AggregatedActivity
R06cAggregatedActivityRecovery
InferenceOCL: activity.aggregate(fso)
activity : DesignActivity
++ aggregate : AggregateDesignPhase
++ aggregation : AggregatedActivity
outcome : Outcome
fso : FileSystemObject
++ outcome2 : Outcome
belief := activity.attributes->select(attr|attr.key = 'belief')->first().value.ocl...
name := activity.attributes->select(attr|attr.key = 'name')->first().value.ocl...
activities : DesignActivity

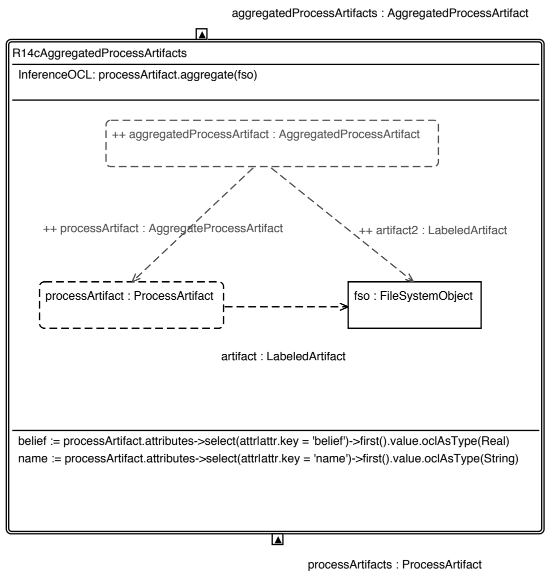
(a) Aggregates multiple recovered design activities.


aggregatedProcessSteps : AggregatedProcessStep
R08cAggregatedProcessStep
InferenceOCL: processStep.aggregate(fso)
processStep : DesignPhase
++ aggregate : AggregateDesignPhase
++ aggregate : AggregatedProcessStep
outcome : Outcome
fso : FileSystemObject
++ outcome2 : Outcome
belief := processStep.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real)
name := processStep.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
processSteps : DesignPhase

(b) Aggregates multiple recovered design phases.


aggregatedMilestones : AggregatedMilestone
R13dAggregatedMilestones
InferenceOCL: milestone.aggregate(fso)
++ aggregatedMilestone : AggregatedMilestone
++ phase2 : Phase
++ milestone : AggregateMilestone
++ artifact2 : LabeledArtifact
milestone : Milestone
phase : Phase
artifact : LabeledArtifact
phase : DesignStep
fso : FileSystemObject
belief := milestone.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real)
name := milestone.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
milestones : Milestone

(c) Aggregates multiple recovered milestones per design phase.


aggregatedProcessArtifacts : AggregatedProcessArtifact
R14cAggregatedProcessArtifacts
InferenceOCL: processArtifact.aggregate(fso)
++ aggregatedProcessArtifact : AggregatedProcessArtifact
++ processArtifact : AggregateProcessArtifact
++ artifact2 : LabeledArtifact
processArtifact : ProcessArtifact
fso : FileSystemObject
artifact : LabeledArtifact
belief := processArtifact.attributes->select(attr|attr.key = 'belief')->first().value.oclAsType(Real)
name := processArtifact.attributes->select(attr|attr.key = 'name')->first().value.oclAsType(String)
processArtifacts : ProcessArtifact

(d) Aggregates multiple recovered process artifacts.

Figure G.59.: Aggregation

H. Performance Evaluation

The following sections show the raw data of the evaluation measurements.

H.1. Comparison of Maintenance Algorithms

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
6	12442	1767	410	9971	0	0	12442
8	12425	1766	106	4967	0	0	537
11	12425	1766	0	4819	0	0	0
14	12220	1697	60	4632	0	0	248
15	12370	1646	504	4402	0	0	4315
16	12371	1644	27	3948	0	0	260
17	12361	1644	8	4320	0	0	88
18	12364	1644	1	4360	0	0	14
19	12288	1646	65	4434	0	0	547
25	12288	1646	0	4020	0	0	0
26	12332	1660	1	4013	0	0	44
28	12276	1654	31	4007	0	0	177
29	12376	1667	42	4528	0	0	401
30	12379	1667	5	4102	0	0	4
31	12344	1667	123	4549	0	0	1119
32	12475	1697	55	4695	0	0	396
33	12565	1713	29	4219	0	0	287
34	12570	1710	59	3929	0	0	570
35	12562	1705	5	3528	0	0	27
36	12560	1705	6	3650	0	0	51
38	12564	1705	25	3634	0	0	267
39	12578	1707	50	3387	0	0	513
41	14445	1852	14	3970	0	0	1867
42	14418	1848	36	4161	0	0	177
46	14486	1848	32	4381	0	0	384
47	14544	1848	88	4574	0	0	959
52	14544	1852	8	4636	0	0	41
53	15087	1906	195	4445	0	0	1993
54	20032	2725	129	9255	0	0	5356
55	20208	2755	106	9669	0	0	617
56	20217	2758	93	9741	0	0	404
58	20217	2758	6	9877	0	0	49
59	21215	2851	10	10520	0	0	998
60	21216	2851	1	9550	0	0	4
61	21172	2839	614	9320	0	0	2666
62	21217	2836	71	10008	0	0	428
63	21167	2835	34	9926	0	0	169
64	21195	2843	18	9447	0	0	83
66	21193	2843	11	10215	0	0	77
67	21021	2823	90	10078	0	0	330
68	21085	2841	101	10872	0	0	675
69	21151	2853	93	9741	0	0	422
70	21154	2853	13	11369	0	0	83
71	21158	2852	14	9983	0	0	68
72	21292	2870	88	10778	0	0	417
73	21288	2874	18	10526	0	0	43
74	21299	2874	9	9575	0	0	67
75	21321	2882	6	9437	0	0	44
76	21321	2882	34	10331	0	0	167
77	21323	2882	5	10440	0	0	3
78	21343	2879	61	10619	0	0	149
80	21344	2879	9	12064	0	0	51
81	21342	2876	20	10245	0	0	122
82	21342	2876	57	10205	0	0	163
83	21347	2876	61	9691	0	0	289
85	21349	2876	4	10329	0	0	3
92	21351	2876	6	10750	0	0	22
93	21615	2894	11	10384	0	0	312
94	21931	2933	5	11277	0	0	316
95	21976	2931	71	11032	0	0	482
97	21975	2914	40	10081	0	0	273
98	22084	2946	3	10709	0	0	109
100	22174	2982	97	11329	0	0	345

Table H.3.: Ant (naive maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
6	12442	1767	154	3685	87	951	12442
8	12425	1766	46	3800	30	747	537
11	12425	1766	0	3486	28	746	0
14	12220	1697	52	4374	130	727	248
15	12370	1646	403	3081	26	743	4315
16	12371	1644	24	3238	24	687	260
17	12361	1644	8	3424	25	698	88
18	12364	1644	2	3110	26	700	14
19	12288	1646	63	3460	25	716	547
25	12288	1646	0	3453	26	712	0
26	12332	1660	1	3417	26	727	44
28	12276	1654	36	3358	34	735	177
29	12376	1667	44	3521	25	763	401
30	12379	1667	2	3386	25	776	4
31	12344	1667	132	3428	26	768	1119
32	12475	1697	30	3341	26	801	396
33	12565	1713	33	3693	27	829	287
34	12570	1710	65	3568	27	852	570
35	12562	1705	7	3384	35	855	27
36	12560	1705	4	3764	26	857	51
38	12564	1705	27	3807	25	876	267
39	12578	1707	49	3829	25	902	513
41	14445	1852	12	5119	27	980	1867
42	14418	1848	50	4037	27	983	177
46	14486	1848	36	4258	28	1000	384
47	14544	1848	90	4514	25	1044	959
52	14544	1852	11	4310	28	1051	41
53	15087	1906	249	4791	29	1184	1993
54	20032	2725	165	9636	36	1886	5356
55	20208	2755	128	8538	38	1890	617
56	20217	2758	124	8854	38	1960	404
58	20217	2758	6	9986	37	1979	49
59	21215	2851	9	9689	35	3056	998
60	21216	2851	0	10097	38	2051	4
61	21172	2839	771	9519	37	2229	2666
62	21217	2836	78	10692	36	2254	428
63	21167	2835	29	9243	38	2892	169
64	21195	2843	17	9836	35	2247	83
66	21193	2843	10	9865	36	2256	77
67	21021	2823	97	8875	36	2251	330
68	21085	2841	119	10426	36	2270	675
69	21151	2853	128	11020	38	3836	422
70	21154	2853	13	9744	35	2371	83
71	21158	2852	15	9392	37	2376	68
72	21292	2870	114	10828	36	2435	417
73	21288	2874	29	9780	34	2400	43
74	21299	2874	8	12331	36	2428	67
75	21321	2882	6	9161	36	2461	44
76	21321	2882	36	10280	35	2498	167
77	21323	2882	5	9271	36	2517	3
78	21343	2879	84	10075	36	2512	149
80	21344	2879	9	10937	35	2564	51
81	21342	2876	22	10165	45	2531	122
82	21342	2876	73	9348	36	2592	163
83	21347	2876	79	10106	37	2634	289
85	21349	2876	5	10098	36	2653	3
92	21351	2876	6	10257	35	2625	22
93	21615	2894	10	10785	37	2665	312
94	21931	2933	5	9707	36	2743	316
95	21976	2931	71	10263	37	2753	482
97	21975	2914	41	9601	36	2695	273
98	22084	2946	3	10822	37	2771	109
100	22174	2982	129	10118	38	2843	345

Table H.4.: Ant (batch maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
6	12442	1767	103	6156	0	0	12442
8	12425	1766	42	734	28	45	537
11	12425	1766	0	30	0	0	0
14	12220	1697	44	51	85	1	248
15	12370	1646	414	1067	249	85	4315
16	12371	1644	21	557	14	68	260
17	12361	1644	7	67	4	5	88
18	12364	1644	1	127	0	3	14
19	12288	1646	50	344	8	69	547
25	12288	1646	0	29	0	0	0
26	12332	1660	0	46	0	0	44
28	12276	1654	21	332	7	44	177
29	12376	1667	26	459	22	55	401
30	12379	1667	2	29	0	0	4
31	12344	1667	79	422	3	68	1119
32	12475	1697	21	131	6	16	396
33	12565	1713	17	82	0	1	287
34	12570	1710	50	129	10	36	570
35	12562	1705	3	52	7	3	27
36	12560	1705	4	45	6	9	51
38	12564	1705	18	104	0	0	267
39	12578	1707	43	151	13	17	513
41	14445	1852	12	399	0	0	1867
42	14418	1848	16	372	6	42	177
46	14486	1848	23	183	0	1	384
47	14544	1848	77	306	8	28	959
52	14544	1852	3	41	0	2	41
53	15087	1906	121	823	52	79	1993
54	20032	2725	124	2794	11	49	5356
55	20208	2755	53	152	32	39	617
56	20217	2758	60	311	16	53	404
58	20217	2758	6	33	0	0	49
59	21215	2851	9	292	0	0	998
60	21216	2851	1	29	0	0	4
61	21172	2839	350	628	115	91	2666
62	21217	2836	50	177	17	24	428
63	21167	2835	30	103	3	19	169
64	21195	2843	16	120	23	18	83
66	21193	2843	9	46	0	0	77
67	21021	2823	62	87	63	90	330
68	21085	2841	65	251	108	88	675
69	21151	2853	40	164	23	52	422
70	21154	2853	9	63	0	0	83
71	21158	2852	8	82	12	41	68
72	21292	2870	37	468	16	58	417
73	21288	2874	6	81	2	4	43
74	21299	2874	7	87	0	35	67
75	21321	2882	3	186	21	48	44
76	21321	2882	23	101	73	46	167
77	21323	2882	4	30	0	0	3
78	21343	2879	23	217	26	64	149
80	21344	2879	6	46	0	0	51
81	21342	2876	12	86	11	39	122
82	21342	2876	20	68	11	20	163
83	21347	2876	29	108	0	0	289
85	21349	2876	4	30	0	0	3
92	21351	2876	3	34	0	0	22
93	21615	2894	9	107	0	0	312
94	21931	2933	5	120	0	0	316
95	21976	2931	59	101	16	15	482
97	21975	2914	32	89	41	39	273
98	22084	2946	3	84	0	0	109
100	22174	2982	29	264	16	49	345

Table H.5.: Ant (incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
7	12467	1352	409	7535	0	0	12467
10	44581	4693	782	27874	0	0	32114
16	48548	5445	65	36176	0	0	3967
18	49740	5616	46	37111	0	0	1192
19	49777	5635	338	31585	0	0	1651
20	50022	5667	183	35419	0	0	1013
21	50022	5667	28	32325	0	0	111
22	50131	5665	135	34299	0	0	613
24	50123	5663	20	35471	0	0	53
25	50325	5686	201	34310	0	0	974
26	50728	5714	292	36764	0	0	1397
27	50687	5705	175	35273	0	0	713
30	50678	5702	119	36580	0	0	516
33	50678	5702	88	38050	0	0	263
37	51205	5728	81	37238	0	0	766
38	51209	5728	42	35990	0	0	230
42	51211	5728	79	38312	0	0	177
44	51231	5729	1	40170	0	0	20
54	51737	5762	112	38161	0	0	889
55	52064	5792	385	38201	0	0	1238
56	52211	5800	98	38023	0	0	644
57	53069	5967	178	40067	0	0	1664
58	53076	5967	134	43725	0	0	363
61	53075	5969	53	40319	0	0	103
62	53080	5969	65	42354	0	0	127
63	53063	5971	532	42276	0	0	1310
64	53247	5988	816	38032	0	0	1806
69	53247	5988	307	41745	0	0	458
70	53374	5998	290	42844	0	0	826
71	53399	5998	153	43291	0	0	351
74	55002	6200	621	47442	0	0	2546
77	55176	6213	36	46266	0	0	298
79	55193	6222	29	49224	0	0	64
81	55195	6222	46	49479	0	0	184
82	55183	6222	19	46606	0	0	30
86	54431	6196	1679	43048	0	0	752
88	54405	6194	74	46009	0	0	33
89	53525	6154	1991	45058	0	0	1187
90	53511	6154	168	42110	0	0	102
91	53655	6169	570	41189	0	0	731
92	53621	6171	701	47732	0	0	1379

Table H.6.: Subclipse (naive maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
7	12467	1352	142	1908	85	698	12467
10	44581	4693	683	22133	60	2580	32114
16	48548	5445	63	30249	60	2854	3967
18	49740	5616	22	32530	61	2967	1192
19	49777	5635	357	34601	62	3108	1651
20	50022	5667	193	34200	62	3188	1013
21	50022	5667	37	35756	63	3199	111
22	50131	5665	150	34560	65	3311	613
24	50123	5663	22	36009	80	3856	53
25	50325	5686	233	35576	62	3380	974
26	50728	5714	325	36287	63	3598	1397
27	50687	5705	187	33836	64	3635	713
30	50678	5702	131	35508	62	3706	516
33	50678	5702	140	37518	62	3727	263
37	51205	5728	96	37228	62	3810	766
38	51209	5728	51	37499	59	3841	230
42	51211	5728	128	37664	63	3863	177
44	51231	5729	1	37891	64	4492	20
54	51737	5762	164	37186	62	3974	889
55	52064	5792	557	39084	60	4062	1238
56	52211	5800	90	39748	62	4097	644
57	53069	5967	204	42250	64	4310	1664
58	53076	5967	188	42456	64	4341	363
61	53075	5969	96	40652	63	4369	103
62	53080	5969	94	40881	64	4418	127
63	53063	5971	941	43250	64	4639	1310
64	53247	5988	1225	38820	67	4821	1806
69	53247	5988	472	43899	64	4823	458
70	53374	5998	465	41947	66	4928	826
71	53399	5998	240	41328	63	4956	351
74	55002	6200	1127	45920	65	5242	2546
77	55176	6213	33	46403	66	5281	298
79	55193	6222	52	48973	68	5320	64
81	55195	6222	50	45852	65	5343	184
82	55183	6222	24	46046	66	5351	30
86	54431	6196	2315	47411	311	5352	752
88	54405	6194	104	42574	85	5367	33
89	53525	6154	3171	45070	66	5386	1187
90	53511	6154	223	47391	66	5379	102
91	53655	6169	918	44072	66	5373	731
92	53621	6171	889	44099	66	5470	1379

Table H.7.: Subclipse (batch maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
7	12467	1352	113	4335	0	0	12467
10	44581	4693	647	18206	0	0	32114
16	48548	5445	90	2488	0	0	3967
18	49740	5616	21	293	0	0	1192
19	49777	5635	240	2039	245	174	1651
20	50022	5667	124	515	49	34	1013
21	50022	5667	19	5791	0	21	111
22	50131	5665	85	788	115	71	613
24	50123	5663	10	84	16	15	53
25	50325	5686	121	1671	142	121	974
26	50728	5714	158	1681	239	185	1397
27	50687	5705	123	1040	239	120	713
30	50678	5702	79	1341	59	108	516
33	50678	5702	42	238	0	2	263
37	51205	5728	48	845	0	6	766
38	51209	5728	34	465	13	1	230
42	51211	5728	26	146	30	39	177
44	51231	5729	1	34	0	0	20
54	51737	5762	61	1055	45	109	889
55	52064	5792	165	1846	177	151	1238
56	52211	5800	73	976	53	121	644
57	53069	5967	154	4408	141	155	1664
58	53076	5967	66	2294	47	63	363
61	53075	5969	17	129	56	16	103
62	53080	5969	20	58	0	2	127
63	53063	5971	217	4093	45	61	1310
64	53247	5988	277	2029	1045	473	1806
69	53247	5988	66	170	78	91	458
70	53374	5998	128	1740	42	54	826
71	53399	5998	62	1264	78	15	351
74	55002	6200	243	3415	61	22	2546
77	55176	6213	33	1703	0	27	298
79	55193	6222	9	92	0	31	64
81	55195	6222	30	2460	85	21	184
82	55183	6222	9	1162	0	9	30
86	54431	6196	345	28	211	0	752
88	54405	6194	12	74	15	1566	33
89	53525	6154	459	2398	359	165	1187
90	53511	6154	26	81	0	1	102
91	53655	6169	131	2752	36	46	731
92	53621	6171	257	2501	26	199	1379

Table H.8.: Subclipse (incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	59423	4330	1113	30667	0	0	59423
2	59423	4330	34	26017	0	0	150
15	59395	4330	228	25962	0	0	1170
16	59703	4356	108	22547	0	0	989
17	59597	4346	184	21505	0	0	1025
18	59597	4345	115	20750	0	0	739
19	59635	4347	340	20914	0	0	1543
20	60409	4470	25	22135	0	0	798
21	60410	4470	223	22629	0	0	985
22	60853	4515	7	22328	0	0	443
23	60822	4509	22	22679	0	0	76
24	60794	4507	134	22688	0	0	678
25	61195	4537	143	23167	0	0	719
26	61267	4554	52	23010	0	0	269
27	61470	4556	772	23409	0	0	1901
28	61539	4580	13	23919	0	0	119
29	61523	4579	49	22345	0	0	93
30	61562	4581	442	23875	0	0	1570
31	61579	4581	506	23153	0	0	1400
33	61570	4581	1244	22743	0	0	2410
34	61562	4576	41	24501	0	0	66
35	61540	4570	48	23648	0	0	98
36	61576	4590	29	23851	0	0	122
37	61782	4596	244	24567	0	0	695
38	61860	4602	13	27320	0	0	91
39	61872	4603	27	24656	0	0	167
40	61937	4606	14	24195	0	0	120
41	61932	4606	428	22951	0	0	456
42	61862	4601	6385	23410	0	0	7821
43	61862	4601	7	24772	0	0	28
44	61901	4603	1697	25740	0	0	3319
45	61914	4605	34	24467	0	0	60
47	62383	4628	656	26511	0	0	3743
48	63139	4719	9	25497	0	0	756
49	63155	4719	13	24122	0	0	28
50	63155	4719	126	25065	0	0	287
51	63145	4717	79	26712	0	0	329
54	64835	4988	34	29903	0	0	1690
56	64963	4996	506	30515	0	0	2125
57	64963	4996	64	31142	0	0	104
59	64963	4996	10	31331	0	0	21
60	65040	5004	421	31815	0	0	385
62	65156	5008	490	31517	0	0	2379
65	66298	5046	398	32361	0	0	3203
66	66298	5046	4	33715	0	0	21
68	66298	5046	498	31068	0	0	2645
71	67417	5125	42	33327	0	0	1131
73	67370	5125	419	34313	0	0	390
74	67374	5126	1440	34713	0	0	1350
80	67374	5126	7	33425	0	0	21
81	67374	5126	4	32394	0	0	21
83	67372	5126	51	33629	0	0	82
85	67372	5126	727	34983	0	0	2645
91	67454	5129	6	34550	0	0	82
100	67458	5129	67	32993	0	0	263

Table H.9.: Commons IO (naive maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	59423	4330	715	19815	173	2160	59423
2	59423	4330	23	19165	52	1769	150
15	59395	4330	202	20129	52	1736	1170
16	59703	4356	99	20593	50	1840	989
17	59597	4346	186	21213	51	1882	1025
18	59597	4345	103	21406	56	1881	739
19	59635	4347	388	20442	50	1931	1543
20	60409	4470	20	22033	52	2045	798
21	60410	4470	211	22578	51	2134	985
22	60853	4515	7	23626	52	2172	443
23	60822	4509	18	22424	51	2172	76
24	60794	4507	155	23015	70	2341	678
25	61195	4537	201	22856	52	2256	719
26	61267	4554	61	23202	53	2270	269
27	61470	4556	1111	23711	52	2568	1901
28	61539	4580	10	23100	51	2472	119
29	61523	4579	72	23918	61	2525	93
30	61562	4581	528	24907	50	2572	1570
31	61579	4581	925	21635	52	2616	1400
33	61570	4581	2332	23064	51	2678	2410
34	61562	4576	56	23827	53	2677	66
35	61540	4570	72	23661	50	2620	98
36	61576	4590	41	23111	53	2698	122
37	61782	4596	404	25271	49	3387	695
38	61860	4602	16	25332	52	2787	91
39	61872	4603	30	25128	50	2801	167
40	61937	4606	14	24989	51	2827	120
41	61932	4606	709	24664	52	2804	456
42	61862	4601	9139	24219	54	3086	7821
43	61862	4601	7	23494	52	3058	28
44	61901	4603	2607	24135	52	3159	3319
45	61914	4605	54	24779	51	3105	60
47	62383	4628	669	25784	53	3290	3743
48	63139	4719	10	25632	52	3362	756
49	63155	4719	16	27225	53	3384	28
50	63155	4719	149	27283	85	3341	287
51	63145	4717	71	26480	54	3415	329
54	64835	4988	36	31512	56	3764	1690
56	64963	4996	627	32023	56	3876	2125
57	64963	4996	93	32350	56	3951	104
59	64963	4996	12	32556	56	3980	21
60	65040	5004	660	30384	57	3982	385
62	65156	5008	490	31210	59	4021	2379
65	66298	5046	361	33889	58	4234	3203
66	66298	5046	5	33318	57	4204	21
68	66298	5046	450	31818	58	4270	2645
71	67417	5125	50	32437	58	4462	1131
73	67370	5125	711	34420	60	4420	390
74	67374	5126	2140	35244	59	4625	1350
80	67374	5126	7	34652	58	5086	21
81	67374	5126	5	34069	59	4613	21
83	67372	5126	71	35287	58	4641	82
85	67372	5126	853	35300	58	4689	2645
91	67454	5129	6	31504	57	4740	82
100	67458	5129	83	33591	58	4723	263

Table H.10.: Commons IO (batch maintenance)

Revision	**Artifacts**	**Annotations**	**Build (ms)**	**Create (ms)**	**Delete (ms)**	**Update (ms)**	**Events**
1	59423	4330	651	30818	0	0	59423
2	59423	4330	19	53	0	0	150
15	59395	4330	164	316	143	90	1170
16	59703	4356	83	208	0	0	989
17	59597	4346	155	205	177	97	1025
18	59597	4345	91	165	8	10	739
19	59635	4347	210	606	23	53	1543
20	60409	4470	15	213	0	1	798
21	60410	4470	144	206	136	5	985
22	60853	4515	7	110	0	0	443
23	60822	4509	17	47	47	18	76
24	60794	4507	91	164	24	1	678
25	61195	4537	50	274	88	182	719
26	61267	4554	32	83	32	6	269
27	61470	4556	316	672	818	445	1901
28	61539	4580	11	55	0	0	119
29	61523	4579	18	65	12	10	93
30	61562	4581	220	482	852	400	1570
31	61579	4581	197	496	25	59	1400
33	61570	4581	321	592	69	86	2410
34	61562	4576	13	56	24	24	66
35	61540	4570	20	56	35	23	98
36	61576	4590	13	66	0	7	122
37	61782	4596	60	194	35	24	695
38	61860	4602	4	57	0	1	91
39	61872	4603	21	130	25	21	167
40	61937	4606	10	51	0	4	120
41	61932	4606	66	120	0	0	456
42	61862	4601	1146	8940	1124	1462	7821
43	61862	4601	4	35	0	0	28
44	61901	4603	437	633	58	60	3319
45	61914	4605	7	41	0	1	60
47	62383	4628	442	737	59	66	3743
48	63139	4719	10	200	0	0	756
49	63155	4719	4	85	0	1	28
50	63155	4719	61	72	117	0	287
51	63145	4717	60	119	77	66	329
54	64835	4988	34	891	0	0	1690
56	64963	4996	286	432	26	53	2125
57	64963	4996	15	71	0	14	104
59	64963	4996	4	34	0	0	21
60	65040	5004	51	241	102	139	385
62	65156	5008	335	452	26	53	2379
65	66298	5046	307	569	22	44	3203
66	66298	5046	3	33	0	0	21
68	66298	5046	348	415	43	50	2645
71	67417	5125	28	263	0	2	1131
73	67370	5125	72	104	0	3	390
74	67374	5126	186	417	371	385	1350
80	67374	5126	3	34	0	0	21
81	67374	5126	4	33	0	0	21
83	67372	5126	12	107	26	72	82
85	67372	5126	355	425	43	53	2645
91	67454	5129	6	50	0	0	82
100	67458	5129	33	90	75	84	263

Table H.11.: Commons IO (incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
2	133858	20160	1649	537156	0	0	133858
3	155119	21458	396	602596	0	0	21261
4	155139	21464	938	518835	0	0	1292
7	155064	21464	1447	532679	0	0	1296
8	155050	21457	1144	581125	0	0	991
9	161485	22317	81	670908	0	0	6435
12	161460	22314	462	769675	0	0	292
13	161422	22314	1471	693744	0	0	896
14	161422	22314	0	662363	0	0	0
16	161458	22315	1403	636054	0	0	997
17	161423	22315	1370	641098	0	0	1135
18	161436	22319	400	632739	0	0	349
19	161405	22319	358	700793	0	0	210
20	161356	22303	751	621762	0	0	470
21	161411	22305	1802	570412	0	0	1035
22	161466	22303	2761	624656	0	0	1975
23	161457	22303	206	742800	0	0	167
24	161466	22339	364	764892	0	0	267
25	161477	22348	14205	654967	0	0	6494
26	165728	22810	7626	682302	0	0	7612
29	165728	22810	44	724657	0	0	54
30	168753	23719	42	683073	0	0	3025
32	179908	24947	319	624850	0	0	11155
38	179906	24947	165	680901	0	0	146
39	179906	24947	56	711941	0	0	19
40	181526	25125	5889	674345	0	0	3717
42	181521	25124	244	680222	0	0	153
48	182816	25259	14954	688560	0	0	7662
49	182815	25259	41	649625	0	0	40
50	182824	25260	4860	706468	0	0	2022
51	182706	25235	8523	705107	0	0	3719
52	182440	25202	3486	651803	0	0	1086
53	182448	25202	14	733828	0	0	17
54	182472	25218	13	707647	0	0	31
55	182478	25222	7	689524	0	0	13
56	182505	25222	181	649605	0	0	91
58	182805	25261	697	652772	0	0	539
60	182806	25261	77	745167	0	0	67
61	182859	25277	2323	655423	0	0	1115
62	183084	25306	324	663002	0	0	374
63	183085	25310	637	657974	0	0	265
64	183612	25461	3	714683	0	0	527
65	183510	25435	246	751128	0	0	102
66	183467	25434	2214	726324	0	0	892
67	183444	25430	1889	733079	0	0	518
68	183444	25430	0	745170	0	0	0
71	183442	25430	350	727792	0	0	117
73	183439	25427	174	751025	0	0	85
76	183449	25399	5995	715138	0	0	1772
77	183480	25403	117	705563	0	0	72
78	183470	25402	107	749043	0	0	79
79	183637	25421	551	726621	0	0	373
80	183773	25477	2654	752391	0	0	992
81	184391	25544	8457	672492	0	0	3852
82	184387	25546	2423	760347	0	0	908
83	184383	25546	322	758973	0	0	151
84	185242	25624	3328	758161	0	0	1917
85	185288	25630	3309	758025	0	0	1324
86	185407	25652	3102	747016	0	0	1436
87	185261	25650	2235	760829	0	0	908
88	185296	25650	763	736953	0	0	616
90	185327	25650	174	677343	0	0	109
91	185323	25650	715	731409	0	0	413
92	191262	25812	269	782292	0	0	5939
93	191267	25812	1	754508	0	0	5
94	191270	25812	0	752040	0	0	3
95	191270	25812	0	772896	0	0	0
96	191270	25812	0	693341	0	0	0
97	191314	25816	794	772817	0	0	480
98	191407	25831	871	746318	0	0	289
99	191407	25831	0	696307	0	0	0
100	191415	25828	776	779137	0	0	275

Table H.12.: Xerces (naive maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
2	133858	20160	1800	725047	372	17505	133858
3	155119	21458	528	624516	263	21336	21261
4	155139	21464	1347	623385	294	21877	1292
7	155064	21464	1857	644502	253	22203	1296
8	155050	21457	1669	559574	281	22535	991
9	161485	22317	103	803447	270	25959	6435
12	161460	22314	613	688352	299	24764	292
13	161422	22314	2137	759855	270	25359	896
14	161422	22314	0	683127	297	26033	0
16	161458	22315	2089	624820	283	25564	997
17	161423	22315	1831	695774	306	26646	1135
18	161436	22319	538	833148	275	29264	349
19	161405	22319	610	685416	294	26374	210
20	161356	22303	1000	820063	295	26387	470
21	161411	22305	2138	738755	300	26526	1035
22	161466	22303	3661	665670	313	30221	1975
23	161457	22303	304	709029	282	27595	167
24	161466	22339	498	843547	269	27311	267
25	161477	22348	20249	760181	271	28159	6494
26	165728	22810	10579	665117	315	30486	7612
29	165728	22810	72	755398	272	29812	54
30	168753	23719	52	749115	313	32372	3025
32	179908	24947	471	702148	332	36817	11155
38	179906	24947	243	762366	334	36070	146
39	179906	24947	120	779499	303	36464	19
40	181526	25125	8805	782683	386	36713	3717
42	181521	25124	392	779655	351	36957	153
48	182816	25259	21801	817649	359	38504	7662
49	182815	25259	51	819433	346	38089	40
50	182824	25260	7341	785205	321	38459	2022
51	182706	25235	12240	807105	314	38953	3719
52	182440	25202	5143	798911	310	39033	1086
53	182448	25202	61	764659	346	38780	17
54	182472	25218	29	803977	328	39118	31
55	182478	25222	10	800776	311	42673	13
56	182505	25222	340	828310	304	40993	91
58	182805	25261	1103	806014	310	42458	539
60	182806	25261	98	822659	330	43356	67
61	182859	25277	3527	745495	325	40773	1115
62	183084	25306	505	749550	347	41828	374
63	183085	25310	1072	764916	320	41743	265
64	183612	25461	6	767427	322	43437	527
65	183510	25435	286	792022	1347	42296	102
66	183467	25434	3576	780710	315	41667	892
67	183444	25430	3051	893181	313	44567	518
68	183444	25430	0	876926	320	43805	0
71	183442	25430	616	783456	319	42314	117
73	183439	25427	312	850547	328	43051	85
76	183449	25399	8942	860278	357	42239	1772
77	183480	25403	230	851924	313	44320	72
78	183470	25402	130	785148	317	43153	79
79	183637	25421	932	836744	319	42698	373
80	183773	25477	4530	860994	351	43252	992
81	184391	25544	12794	843571	305	46168	3852
82	184387	25546	3788	803942	313	44420	908
83	184383	25546	523	889489	344	47832	151
84	185242	25624	5335	847128	311	47904	1917
85	185288	25630	5157	815559	362	56980	1324
86	185407	25652	5334	788337	337	48776	1436
87	185261	25650	3762	841922	320	46292	908
88	185296	25650	1044	798776	341	46477	616
90	185327	25650	343	892185	328	45822	109
91	185323	25650	985	847500	352	46775	413
92	191262	25812	305	931208	351	46894	5939
93	191267	25812	1	855466	347	48155	5
94	191270	25812	0	889667	330	47734	3
95	191270	25812	0	812113	362	47401	0
96	191270	25812	0	883382	347	50016	0
97	191314	25816	1037	886882	321	48359	480
98	191407	25831	1575	919621	358	48338	289
99	191407	25831	0	853515	340	49866	0
100	191415	25828	1308	869350	313	50928	275

Table H.13.: Xerces (batch maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
2	133858	20160	1079	1798529	0	0	133858
3	155119	21458	337	287323	0	0	21261
4	155139	21464	903	3022	30	549	1292
7	155064	21464	1059	288	298	310	1296
8	155050	21457	768	1974	269	1041	991
9	161485	22317	39	2603	0	0	6435
12	161460	22314	296	70	313	132	292
13	161422	22314	848	702	0	110	896
14	161422	22314	1	28	0	0	0
16	161458	22315	821	1592	1468	1067	997
17	161423	22315	835	460	1594	1674	1135
18	161436	22319	238	194	220	300	349
19	161405	22319	211	48	0	2	210
20	161356	22303	438	387	1160	954	470
21	161411	22305	839	137	94	12	1035
22	161466	22303	1468	1006	619	579	1975
23	161457	22303	130	56	0	4	167
24	161466	22339	173	450	251	236	267
25	161477	22348	5741	2690	649	5883	6494
26	165728	22810	3187	5868	2182	2047	7612
29	165728	22810	33	65	0	0	54
30	168753	23719	39	1061	0	0	3025
32	179908	24947	312	2930	0	0	11155
38	179906	24947	131	524	212	232	146
39	179906	24947	23	39	0	6	19
40	181526	25125	2539	6468	323	884	3717
42	181521	25124	179	3632	104	79	153
48	182816	25259	7812	1962	6707	90	7662
49	182815	25259	40	34	0	0	40
50	182824	25260	2146	699	1038	877	2022
51	182706	25235	3923	14937	2433	1874	3719
52	182440	25202	1593	1143	1026	839	1086
53	182448	25202	4	8333	0	3	17
54	182472	25218	7	84	0	59	31
55	182478	25222	8	83	0	78	13
56	182505	25222	82	119	232	125	91
58	182805	25261	255	273	243	151	539
60	182806	25261	69	47	0	6	67
61	182859	25277	1035	708	662	526	1115
62	183084	25306	133	134	70	26	374
63	183085	25310	265	140	139	156	265
64	183612	25461	3	179	0	0	527
65	183510	25435	263	31	906	0	102
66	183467	25434	935	750	744	667	892
67	183444	25430	499	519	1395	1419	518
68	183444	25430	0	31	0	0	0
71	183442	25430	120	226	35	128	117
73	183439	25427	90	1023	118	345	85
76	183449	25399	2287	474	4757	3575	1772
77	183480	25403	39	3502	35	48	72
78	183470	25402	106	5233	139	81	79
79	183637	25421	257	112	70	11	373
80	183773	25477	895	521	407	330	992
81	184391	25544	3881	1349	1608	2174	3852
82	184387	25546	922	643	338	1513	908
83	184383	25546	133	75	0	1	151
84	185242	25624	1716	613	1318	825	1917
85	185288	25630	2019	827	773	1972	1324
86	185407	25652	1686	127196	398	942	1436
87	185261	25650	1722	683	378	1477	908
88	185296	25650	838	679	960	989	616
90	185327	25650	121	49	0	9	109
91	185323	25650	596	173	396	436	413
92	191262	25812	274	1239	0	0	5939
93	191267	25812	1	32	0	0	5
94	191270	25812	0	30	0	0	3
95	191270	25812	0	31	0	0	0
96	191270	25812	0	30	0	0	0
97	191314	25816	665	223	794	660	480
98	191407	25831	353	165	58	375	289
99	191407	25831	0	31	0	0	0
100	191415	25828	393	311	563	342	275

Table H.14.: Xerces (incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	228423	22999	5297	846766	0	0	228423
2	228423	22986	881	819525	0	0	598
14	228542	22985	1336	743151	0	0	1191
15	228541	22985	4734	712197	0	0	4018
23	228541	22985	1010	727311	0	0	750
27	229795	23013	4286	785634	0	0	4216
28	229795	23013	43	680226	0	0	24
29	229794	23012	29	736730	0	0	20
41	229794	23012	230	692994	0	0	126
55	230166	23024	4697	793863	0	0	2994
57	230165	23024	65	741759	0	0	42
59	230151	23024	3351	730899	0	0	1620
60	230204	23027	2005	787462	0	0	1017
61	230233	23029	2284	816626	0	0	1471
62	230347	23033	1260	786599	0	0	731
63	230418	23032	1403	736344	0	0	756
65	230418	23032	237	722881	0	0	273
66	230418	23032	426	686433	0	0	347
67	230611	23037	2107	700403	0	0	1028
72	230641	23038	3397	790225	0	0	1974
74	230645	23039	159	732369	0	0	68
75	230769	23047	476	820199	0	0	318
80	230786	23091	2038	831423	0	0	568
85	230786	23091	1564	733303	0	0	517
92	230784	23091	811	829752	0	0	284

Table H.15.: Commons Collections (naive maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	228423	22999	4853	805257	374	25456	228423
2	228423	22986	771	797849	247	25060	598
14	228542	22985	1407	721995	232	25447	1191
15	228541	22985	5654	744088	233	26185	4018
23	228541	22985	1327	729863	228	26387	750
27	229795	23013	5228	745190	245	27172	4216
28	229795	23013	62	743767	245	26978	24
29	229794	23012	33	715959	312	27065	20
41	229794	23012	272	819687	255	29541	126
55	230166	23024	5994	814590	238	27628	2994
57	230165	23024	75	821196	244	28017	42
59	230151	23024	4275	836170	244	28035	1620
60	230204	23027	2570	832884	240	28654	1017
61	230233	23029	2851	741129	240	29028	1471
62	230347	23033	1636	751241	241	29250	731
63	230418	23032	1970	809066	230	29547	756
65	230418	23032	296	797934	242	29750	273
66	230418	23032	515	724689	244	30109	347
67	230611	23037	2673	789012	235	29986	1028
72	230641	23038	4379	817538	238	30466	1974
74	230645	23039	231	826892	231	30545	68
75	230769	23047	662	826652	226	32102	318
80	230786	23091	3078	835573	399	31295	568
85	230786	23091	2273	749130	242	31248	517
92	230784	23091	1131	821719	241	31882	284

Table H.16.: Commons Collections (batch maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	228423	22999	4555	822159	0	0	228423
2	228423	22986	676	191	991	254	598
14	228542	22985	961	1561	795	1028	1191
15	228541	22985	3473	873	1809	1830	4018
23	228541	22985	674	410	796	879	750
27	229795	23013	2547	1186	195	79	4216
28	229795	23013	28	49	0	11	24
29	229794	23012	17	933	74	104	20
41	229794	23012	109	307	0	45	126
55	230166	23024	2407	706	224	456	2994
57	230165	23024	37	42	0	0	42
59	230151	23024	1479	3937	1797	1922	1620
60	230204	23027	907	408	587	757	1017
61	230233	23029	1141	357	0	5	1471
62	230347	23033	501	163	90	109	731
63	230418	23032	594	554	91	324	756
65	230418	23032	203	83	0	0	273
66	230418	23032	241	75	0	3	347
67	230611	23037	949	368	1938	1966	1028
72	230641	23038	1545	565	47	120	1974
74	230645	23039	56	58	0	10	68
75	230769	23047	212	89	0	6	318
80	230786	23091	425	1170	174	22	568
85	230786	23091	472	246	136	186	517
92	230784	23091	281	86	0	0	284

Table H.17.: Commons Collections (incremental maintenance)

H.2. Comparison of Network Structures

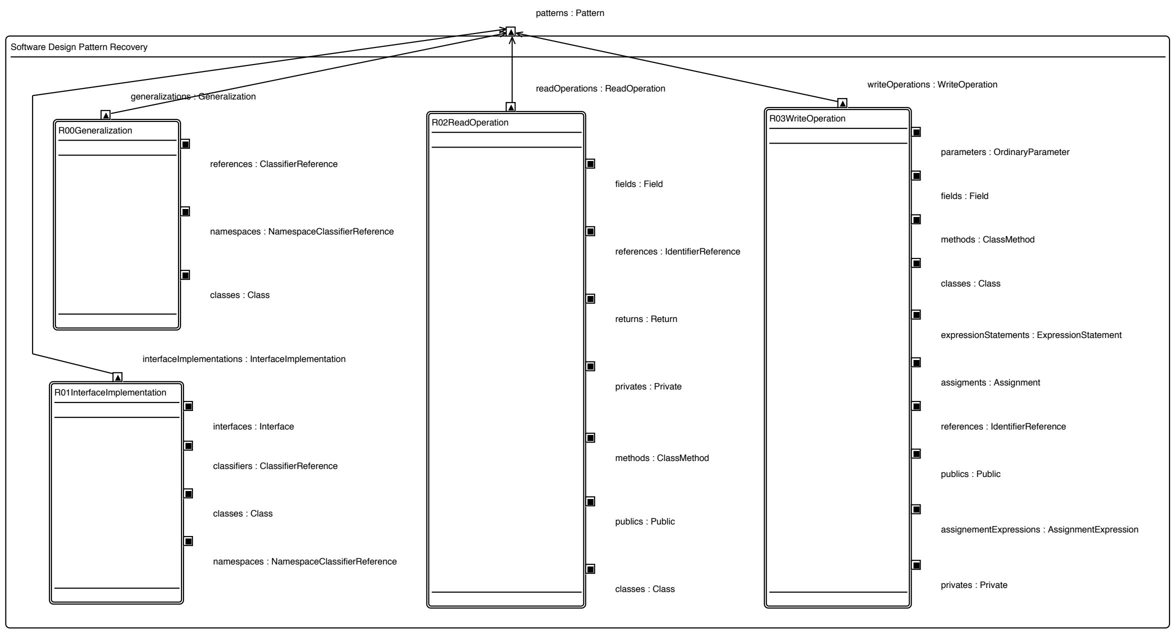

patterns : Pattern
Software Design Pattern Recovery
generalizations : Generalization
R00Generalization
references : ClassifierReference
namespaces : NamespaceClassifierReference
classes : Class
interfaceImplementations : InterfaceImplementation
R01InterfaceImplementation
interfaces : Interface
classifiers : ClassifierReference
classes : Class
namespaces : NamespaceClassifierReference
readOperations : ReadOperation
R02ReadOperation
fields : Field
references : IdentifierReference
returns : Return
privates : Private
methods : ClassMethod
publics : Public
classes : Class
writeOperations : WriteOperation
R03WriteOperation
parameters : OrdinaryParameter
fields : Field
methods : ClassMethod
classes : Class
expressionStatements : ExpressionStatement
assigments : Assignment
references : IdentifierReference
publics : Public
assignementExpressions : AssignmentExpression
privates : Private

Figure H.60.: Gator network structure for exterior evaluation

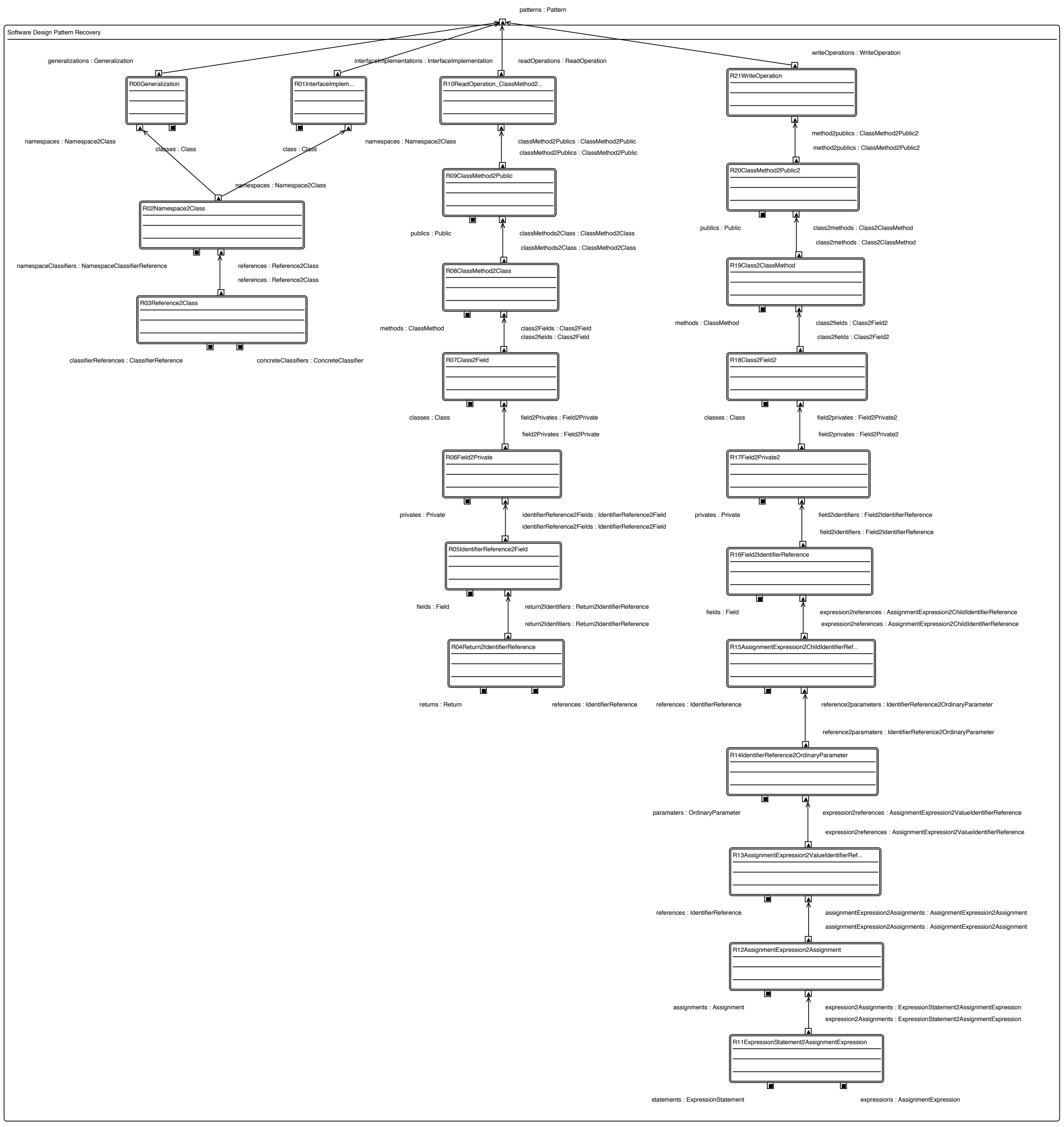

patterns : Pattern
Software Design Pattern Recovery
generalizations : Generalization
interfaceImplementations : InterfaceImplementation
readOperations : ReadOperation
writeOperations : WriteOperation
R00Generalization
R01InterfaceImplem...
R10ReadOperation_ClassMethod2...
R21WriteOperation
namespaces : Namespace2Class
classes : Class
class : Class
namespaces : Namespace2Class
classMethod2Publics : ClassMethod2Public
classMethod2Publics : ClassMethod2Public
method2publics : ClassMethod2Public2
method2publics : ClassMethod2Public2
namespaces : Namespace2Class
R09ClassMethod2Public
R20ClassMethod2Public2
R02Namespace2Class
publics : Public
classMethods2Class : ClassMethod2Class
classMethods2Class : ClassMethod2Class
publics : Public
class2methods : Class2ClassMethod
class2methods : Class2ClassMethod
namespaceClassifiers : NamespaceClassifierReference
references : Reference2Class
references : Reference2Class
R08ClassMethod2Class
R19Class2ClassMethod
R03Reference2Class
methods : ClassMethod
class2Fields : Class2Field
class2fields : Class2Field
methods : ClassMethod
class2fields : Class2Field2
class2fields : Class2Field2
classifierReferences : ClassifierReference
concreteClassifiers : ConcreteClassifier
R07Class2Field
R18Class2Field2
classes : Class
field2Privates : Field2Private
field2Privates : Field2Private
classes : Class
field2privates : Field2Private2
field2privates : Field2Private2
R06Field2Private
R17Field2Private2
privates : Private
identifierReference2Fields : IdentifierReference2Field
identifierReference2Fields : IdentifierReference2Field
privates : Private
field2identifiers : Field2IdentifierReference
field2identifiers : Field2IdentifierReference
R05IdentifierReference2Field
R16Field2IdentifierReference
fields : Field
return2Identifiers : Return2IdentifierReference
return2Identifiers : Return2IdentifierReference
fields : Field
expression2references : AssignmentExpression2ChildIdentifierReference
expression2references : AssignmentExpression2ChildIdentifierReference
R04Return2IdentifierReference
R15AssignmentExpression2ChildIdentifierRef...
returns : Return
references : IdentifierReference
references : IdentifierReference
reference2paramaters : IdentifierReference2OrdinaryParameter
reference2paramaters : IdentifierReference2OrdinaryParameter
R14IdentifierReference2OrdinaryParameter
paramaters : OrdinaryParameter
expression2references : AssignmentExpression2ValueIdentifierReference
expression2references : AssignmentExpression2ValueIdentifierReference
R13AssignmentExpression2ValueIdentifierRef...
references : IdentifierReference
assignmentExpression2Assignments : AssignmentExpression2Assignment
assignmentExpression2Assignments : AssignmentExpression2Assignment
R12AssignmentExpression2Assignment
assignments : Assignment
expression2Assignments : ExpressionStatement2AssignmentExpression
expression2Assignments : ExpressionStatement2AssignmentExpression
R11ExpressionStatement2AssignmentExpression
statements : ExpressionStatement
expressions : AssignmentExpression

Figure H.61.: Rete network structure for exterior evaluation

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
6	12442	36	113	928	0	0	12442
8	12425	36	13	185	8	7	537
11	12425	36	0	3	0	0	0
14	12220	36	14	5	0	0	248
15	12370	34	141	215	55	7	4315
16	12371	33	7	26	3	6	260
17	12361	33	3	6	0	0	88
18	12364	33	1	4	0	0	14
19	12288	32	26	41	4	11	547
25	12288	32	0	4	0	0	0
26	12332	32	1	5	0	0	44
28	12276	31	6	22	2	6	177
29	12376	30	12	27	3	6	401
30	12379	30	2	3	0	0	4
31	12344	30	25	117	0	16	1119
32	12475	30	7	8	0	0	396
33	12565	30	6	8	0	0	287
34	12570	30	16	9	0	0	570
35	12562	30	1	4	0	0	27
36	12560	30	2	4	0	0	51
38	12564	30	6	311	0	0	267
39	12578	30	14	8	0	0	513
41	14445	31	14	38	0	0	1867
42	14418	31	5	15	0	0	177
46	14486	31	7	330	0	0	384
47	14544	31	26	95	0	0	959
52	14544	31	1	3	0	0	41
53	15087	30	49	66	2	8	1993
54	20032	30	63	123	0	2	5356
55	20208	28	16	14	11	3	617
56	20217	28	27	16	0	0	404
58	20217	28	2	3	0	0	49
59	21215	28	11	17	0	0	998
60	21216	28	0	3	0	0	4
61	21172	26	107	175	6	9	2666
62	21217	26	14	23	0	1	428
63	21167	26	9	7	0	0	169
64	21195	26	21	7	0	0	83
66	21193	26	2	5	0	0	77
67	21021	26	11	5	0	0	330
68	21085	27	19	43	6	1	675
69	21151	26	12	12	7	1	422
70	21154	26	2	6	0	0	83
71	21158	26	2	5	0	0	68
72	21292	26	10	27	3	6	417
73	21288	26	1	15	0	0	43
74	21299	26	2	5	0	0	67
75	21321	26	1	19	0	6	44
76	21321	26	11	5	0	0	167
77	21323	26	4	2	0	0	3
78	21343	25	10	23	3	9	149
80	21344	25	1	5	0	0	51
81	21342	25	3	6	0	0	122
82	21342	25	5	8	0	5	163
83	21347	25	8	341	0	0	289
85	21349	25	5	3	0	0	3
92	21351	25	1	3	0	0	22
93	21615	25	6	9	0	0	312
94	21931	27	10	32	0	0	316
95	21976	27	31	9	0	0	482
97	21975	25	17	10	13	1	273
98	22084	25	3	5	0	0	109
100	22174	25	12	14	0	0	345

Table H.18.: Ant (Gator network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
6	12442	2235	99	3814	0	0	12442
8	12425	2283	44	1139	23	27	537
11	12425	2283	0	14	0	0	0
14	12220	2279	47	18	4	0	248
15	12370	2273	439	1334	132	21	4315
16	12371	2270	22	93	11	21	260
17	12361	2270	8	23	0	0	88
18	12364	2270	2	18	0	0	14
19	12288	2272	54	132	7	23	547
25	12288	2272	0	27	0	0	0
26	12332	2275	1	30	0	0	44
28	12276	2272	21	70	6	18	177
29	12376	2276	25	85	15	34	401
30	12379	2276	2	15	0	0	4
31	12344	2292	86	931	7	97	1119
32	12475	2311	23	40	1	2	396
33	12565	2318	18	30	0	1	287
34	12570	2318	53	36	15	0	570
35	12562	2314	3	15	3	1	27
36	12560	2314	5	19	0	1	51
38	12564	2314	20	913	7	20	267
39	12578	2314	63	37	18	0	513
41	14445	2454	11	184	0	0	1867
42	14418	2453	27	54	3	8	177
46	14486	2466	27	957	11	8	384
47	14544	2481	84	1034	22	26	959
52	14544	2483	4	22	4	10	41
53	15087	2513	129	218	13	62	1993
54	20032	2914	88	793	8	10	5356
55	20208	2893	51	65	16	52	617
56	20217	2891	56	47	4	11	404
58	20217	2891	5	17	0	0	49
59	21215	2963	9	94	0	0	998
60	21216	2963	1	17	0	2	4
61	21172	2960	319	1522	65	118	2666
62	21217	2961	45	51	3	5	428
63	21167	2954	27	22	10	5	169
64	21195	2957	16	24	0	0	83
66	21193	2957	8	19	0	0	77
67	21021	2943	58	22	17	0	330
68	21085	2970	59	122	34	79	675
69	21151	2960	37	49	61	22	422
70	21154	2960	9	21	0	0	83
71	21158	2961	7	22	5	0	68
72	21292	2991	34	101	14	19	417
73	21288	2991	5	29	0	0	43
74	21299	2991	6	20	0	2	67
75	21321	2999	3	59	8	40	44
76	21321	2999	21	35	19	47	167
77	21323	2999	5	16	0	0	3
78	21343	2996	21	57	12	30	149
80	21344	2996	6	17	0	0	51
81	21342	2994	13	20	2	2	122
82	21342	2994	20	45	8	18	163
83	21347	2994	28	1336	19	57	289
85	21349	2994	4	15	0	0	3
92	21351	2994	2	17	0	2	22
93	21615	3002	10	36	0	0	312
94	21931	3053	5	80	0	0	316
95	21976	3055	67	41	6	13	482
97	21975	3021	35	37	38	101	273
98	22084	3030	2	26	0	0	109
100	22174	3037	33	54	26	12	345

Table H.19.: Ant (Rete network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
7	12467	27	352	375	0	0	12467
10	44581	56	619	2383	0	0	32114
16	48548	56	78	49	0	0	3967
18	49740	56	28	17	0	0	1192
19	49777	56	57	69	13	15	1651
20	50022	57	38	24	0	0	1013
21	50022	57	5	28	0	5	111
22	50131	58	34	24	0	0	613
24	50123	58	2	5	0	0	53
25	50325	59	45	47	7	2	974
26	50728	60	46	61	0	11	1397
27	50687	60	29	48	0	11	713
30	50678	60	43	53	7	13	516
33	50678	60	19	8	0	0	263
37	51205	60	39	16	0	3	766
38	51209	60	17	10	0	0	230
42	51211	60	6	13	0	0	177
44	51231	60	1	3	0	0	20
54	51737	61	19	40	0	2	889
55	52064	63	73	61	0	4	1238
56	52211	63	20	31	0	3	644
57	53069	69	85	125	0	9	1664
58	53076	69	19	50	7	17	363
61	53075	69	5	5	0	0	103
62	53080	69	12	5	0	0	127
63	53063	69	108	65	0	10	1310
64	53247	70	90	111	20	31	1806
69	53247	70	14	12	0	0	458
70	53374	70	71	69	0	2	826
71	53399	70	27	18	0	0	351
74	55002	75	131	114	0	0	2546
77	55176	75	17	66	0	3	298
79	55193	75	4	4	0	0	64
81	55195	75	7	17	0	3	184
82	55183	75	2	5	0	0	30
86	54431	75	68	3	0	0	752
88	54405	75	2	4	0	0	33
89	53525	75	102	61	0	10	1187
90	53511	75	11	13	0	0	102
91	53655	76	32	54	0	0	731
92	53621	76	165	100	0	8	1379

Table H.20.: Subclipse (Gator network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
7	12467	1600	112	2686	0	0	12467
10	44581	4382	631	16632	0	0	32114
16	48548	4662	86	666	0	0	3967
18	49740	4850	33	233	0	0	1192
19	49777	4922	233	985	335	665	1651
20	50022	4930	121	213	24	165	1013
21	50022	4930	18	221	0	18	111
22	50131	4957	84	178	30	21	613
24	50123	4957	11	19	0	0	53
25	50325	4992	123	564	63	67	974
26	50728	5095	152	1127	44	258	1397
27	50687	5084	125	759	99	180	713
30	50678	5080	82	561	46	62	516
33	50678	5080	42	51	14	19	263
37	51205	5112	50	232	0	4	766
38	51209	5112	34	80	14	11	230
42	51211	5112	27	90	0	0	177
44	51231	5112	0	15	0	0	20
54	51737	5195	63	621	274	153	889
55	52064	5244	167	866	34	52	1238
56	52211	5262	76	448	368	202	644
57	53069	5451	156	1588	21	118	1664
58	53076	5451	65	600	28	64	363
61	53075	5451	18	30	12	0	103
62	53080	5451	21	20	0	0	127
63	53063	5452	218	946	159	102	1310
64	53247	5536	223	1678	219	723	1806
69	53247	5536	53	43	0	3	458
70	53374	5549	107	934	8	133	826
71	53399	5557	50	358	203	113	351
74	55002	5844	206	1598	26	6	2546
77	55176	5860	28	558	92	186	298
79	55193	5864	8	26	0	0	64
81	55195	5864	25	349	297	231	184
82	55183	5862	8	28	9	6	30
86	54431	5830	279	12	162	0	752
88	54405	5830	10	17	0	0	33
89	53525	5818	361	444	154	277	1187
90	53511	5814	23	81	34	0	102
91	53655	5825	109	329	113	41	731
92	53621	5834	239	1327	83	134	1379

Table H.21.: Subclipse (Rete network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	59423	22	700	1934	0	0	59423
2	59423	22	5	4	0	0	150
15	59395	22	41	18	0	0	1170
16	59703	22	26	14	0	0	989
17	59597	22	38	16	0	0	1025
18	59597	22	24	12	0	0	739
19	59635	22	57	39	0	0	1543
20	60409	22	12	11	0	0	798
21	60410	22	41	17	0	0	985
22	60853	22	6	6	0	0	443
23	60822	22	4	3	0	0	76
24	60794	22	23	13	0	0	678
25	61195	22	18	12	0	2	719
26	61267	22	8	5	0	0	269
27	61470	22	102	36	0	0	1901
28	61539	22	3	4	0	0	119
29	61523	22	4	6	0	0	93
30	61562	22	54	24	0	4	1570
31	61579	22	46	33	0	0	1400
33	61570	22	70	28	0	0	2410
34	61562	22	2	4	0	0	66
35	61540	22	4	3	0	0	98
36	61576	22	3	4	0	0	122
37	61782	22	25	11	0	5	695
38	61860	22	2	5	0	0	91
39	61872	22	5	8	0	4	167
40	61937	22	3	3	0	0	120
41	61932	22	16	5	0	0	456
42	61862	22	301	120	0	5	7821
43	61862	22	1	3	0	0	28
44	61901	22	124	35	0	0	3319
45	61914	22	1	3	0	0	60
47	62383	22	135	40	0	0	3743
48	63139	23	9	11	0	0	756
49	63155	23	2	4	0	0	28
50	63155	23	20	5	0	0	287
51	63145	23	13	6	0	2	329
54	64835	27	33	67	0	0	1690
56	64963	27	79	21	0	0	2125
57	64963	27	4	5	0	0	104
59	64963	27	1	3	0	0	21
60	65040	27	12	10	0	2	385
62	65156	27	118	24	0	0	2379
65	66298	27	106	32	0	0	3203
66	66298	27	0	2	0	0	21
68	66298	27	86	26	0	0	2645
71	67417	27	27	13	0	0	1131
73	67370	27	16	11	0	0	390
74	67374	27	42	29	0	0	1350
80	67374	27	1	3	0	0	21
81	67374	27	1	3	0	0	21
83	67372	27	3	7	0	2	82
85	67372	27	86	26	0	0	2645
91	67454	27	7	3	0	0	82
100	67458	27	8	4	0	0	263

Table H.22.: Commons IO (Gator network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	59423	2431	649	13289	0	0	59423
2	59423	2431	12	19	0	0	150
15	59395	2427	111	72	20	53	1170
16	59703	2444	61	67	0	0	989
17	59597	2455	109	136	36	81	1025
18	59597	2455	64	50	22	50	739
19	59635	2455	160	124	13	25	1543
20	60409	2492	13	117	11	0	798
21	60410	2492	105	137	5	90	985
22	60853	2503	6	39	0	0	443
23	60822	2502	12	17	14	0	76
24	60794	2498	65	120	24	27	678
25	61195	2531	36	98	6	31	719
26	61267	2536	21	29	9	2	269
27	61470	2554	221	231	41	116	1901
28	61539	2558	8	22	0	0	119
29	61523	2550	12	101	22	108	93
30	61562	2548	155	177	101	226	1570
31	61579	2548	138	127	18	31	1400
33	61570	2548	211	131	0	3	2410
34	61562	2552	9	33	25	49	66
35	61540	2542	14	26	61	98	98
36	61576	2558	9	29	0	0	122
37	61782	2564	45	115	0	18	695
38	61860	2564	4	23	12	0	91
39	61872	2564	15	91	0	16	167
40	61937	2564	7	18	0	0	120
41	61932	2564	45	30	0	2	456
42	61862	2546	798	777	156	266	7821
43	61862	2546	3	15	0	0	28
44	61901	2546	307	158	0	3	3319
45	61914	2546	5	19	0	2	60
47	62383	2553	320	197	0	4	3743
48	63139	2598	10	77	0	0	756
49	63155	2598	3	23	10	0	28
50	63155	2598	44	27	0	0	287
51	63145	2598	41	43	29	2	329
54	64835	2844	35	494	0	0	1690
56	64963	2844	204	95	0	0	2125
57	64963	2844	11	135	0	154	104
59	64963	2844	2	17	0	2	21
60	65040	2859	33	100	8	30	385
62	65156	2859	245	105	0	2	2379
65	66298	2859	244	144	0	2	3203
66	66298	2859	2	17	0	2	21
68	66298	2859	217	112	0	2	2645
71	67417	2911	26	100	13	0	1131
73	67370	2911	42	50	24	32	390
74	67374	2911	110	138	172	58	1350
80	67374	2911	3	17	0	2	21
81	67374	2911	2	16	0	0	21
83	67372	2931	8	66	0	9	82
85	67372	2931	220	104	0	2	2645
91	67454	2932	7	20	0	0	82
100	67458	2932	21	21	0	0	263

Table H.23.: Commons IO (Rete network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
2	133858	212	1255	316422	0	0	133858
3	155119	216	354	221	0	0	21261
4	155139	216	83	313	0	27	1292
7	155064	216	104	22	0	2	1296
8	155050	216	80	299	0	10	991
9	161485	241	47	279	0	0	6435
12	161460	241	30	8	78	0	292
13	161422	241	69	48	0	4	896
14	161422	241	0	3	0	0	0
16	161458	241	71	454	0	12	997
17	161423	241	75	30	0	0	1135
18	161436	241	22	8	0	0	349
19	161405	241	19	4	0	0	210
20	161356	241	40	25	0	0	470
21	161411	241	93	7	0	0	1035
22	161466	241	166	151	0	31	1975
23	161457	241	16	5	0	0	167
24	161466	241	16	40	0	12	267
25	161477	241	535	279	0	0	6494
26	165728	245	399	863	0	41	7612
29	165728	245	3	5	0	0	54
30	168753	245	45	28	0	0	3025
32	179908	258	344	841	0	0	11155
38	179906	258	10	20	0	0	146
39	179906	258	2	3	0	0	19
40	181526	261	288	334	0	0	3717
42	181521	261	13	159	0	0	153
48	182816	262	704	358	0	45	7662
49	182815	262	4	3	0	0	40
50	182824	262	194	29	0	0	2022
51	182706	262	408	614	0	73	3719
52	182440	262	109	121	0	2	1086
53	182448	262	1	5359	0	0	17
54	182472	262	1	3	0	0	31
55	182478	262	1	2	0	0	13
56	182505	262	7	7	0	2	91
58	182805	265	36	39	0	5	539
60	182806	265	9	5	0	4	67
61	182859	265	83	27	0	0	1115
62	183084	265	12	5	0	0	374
63	183085	265	24	5	0	0	265
64	183612	265	4	5	0	0	527
65	183510	265	20	3	0	0	102
66	183467	265	83	24	0	0	892
67	183444	265	39	13	0	0	518
68	183444	265	0	2	0	0	0
71	183442	265	12	18	0	8	117
73	183439	265	7	29	0	0	85
76	183449	265	156	31	0	11	1772
77	183480	265	3	159	0	0	72
78	183470	265	7	161	0	0	79
79	183637	265	27	6	0	3	373
80	183773	265	67	17	0	0	992
81	184391	265	377	45	0	0	3852
82	184387	265	65	33	0	4	908
83	184383	265	23	10	0	0	151
84	185242	265	90	25	0	0	1917
85	185288	265	116	59	0	4	1324
86	185407	265	85	683	0	60	1436
87	185261	265	85	32	0	3	908
88	185296	265	41	31	0	0	616
90	185327	265	6	2	0	0	109
91	185323	265	28	5	0	0	413
92	191262	265	267	55	0	0	5939
93	191267	265	1	2	0	0	5
94	191270	265	0	2	0	0	3
95	191270	265	0	2	0	0	0
96	191270	265	0	2	0	0	0
97	191314	265	51	7	0	0	480
98	191407	265	17	5	0	0	289
99	191407	265	0	2	0	0	0
100	191415	265	20	10	0	0	275

Table H.24.: Xerces (Gator network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
2	133858	47452	1739	330567	0	0	133858
3	155119	48189	391	2532	0	0	21261
4	155139	48189	3812	6088	0	144	1292
7	155064	48189	4453	511	0	9	1296
8	155050	48185	3152	2422	196	492	991
9	161485	51202	86	3764	0	0	6435
12	161460	51200	1240	41	3496	1519	292
13	161422	51191	3508	232	680	314	896
14	161422	51191	0	20	0	0	0
16	161458	51199	3473	2899	681	911	997
17	161423	51199	4065	289	140	402	1135
18	161436	51199	1149	48	68	147	349
19	161405	51191	1031	32	205	140	210
20	161356	51171	2133	258	1090	648	470
21	161411	51176	4033	121	820	985	1035
22	161466	51175	7027	3218	1230	991	1975
23	161457	51173	452	31	98	4	167
24	161466	51177	625	591	0	84	267
25	161477	51175	18223	1639	1410	542	6494
26	165728	51463	10272	9287	25938	18564	7612
29	165728	51463	102	29	0	0	54
30	168753	51463	41	1061	0	0	3025
32	179908	52800	298	7073	0	0	11155
38	179906	52800	393	164	0	9	146
39	179906	52800	57	20	0	0	19
40	181526	53038	6696	8320	507	836	3717
42	181521	53040	531	6881	0	0	153
48	182816	53319	24185	4340	5269	495	7662
49	182815	53319	113	22	0	0	40
50	182824	53289	6314	402	815	1429	2022
51	182706	53246	12433	9995	6923	1412	3719
52	182440	53236	5294	1892	631	145	1086
53	182448	53236	12	15979	0	50	17
54	182472	53236	21	20	0	0	31
55	182478	53236	20	20	0	0	13
56	182505	53244	260	108	380	148	91
58	182805	53303	770	182	190	270	539
60	182806	53303	216	34	0	6	67
61	182859	53316	3511	253	381	223	1115
62	183084	53316	432	53	0	30	374
63	183085	53314	858	51	126	259	265
64	183612	53318	3	43	0	0	527
65	183510	53318	904	18	0	0	102
66	183467	53307	3103	257	321	365	892
67	183444	53307	1665	81	0	18	518
68	183444	53307	0	16	0	0	0
71	183442	53307	368	352	0	151	117
73	183439	53304	285	906	65	8	85
76	183449	53304	7533	566	951	64	1772
77	183480	53304	113	7411	0	0	72
78	183470	53304	366	7576	0	0	79
79	183637	53318	803	55	118	128	373
80	183773	53342	2848	227	353	455	992
81	184391	53328	12161	543	3166	94	3852
82	184387	53332	2983	497	120	275	908
83	184383	53332	381	182	59	4	151
84	185242	53449	3751	293	1181	428	1917
85	185288	53436	4471	676	11331	587	1324
86	185407	53481	3716	21996	2022	241	1436
87	185261	53478	3754	552	9953	544	908
88	185296	53480	1827	518	291	685	616
90	185327	53480	253	22	0	4	109
91	185323	53484	1297	63	0	17	413
92	191262	54038	251	1153	0	0	5939
93	191267	54038	1	14	0	0	5
94	191270	54038	0	15	0	0	3
95	191270	54038	0	15	0	0	0
96	191270	54038	0	14	0	0	0
97	191314	54042	1411	69	239	16	480
98	191407	54044	701	53	179	247	289
99	191407	54044	0	26	0	0	0
100	191415	54038	873	110	333	135	275

Table H.25.: Xerces (Rete network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	228423	323	4258	321894	0	0	228423
2	228423	321	110	12	197	1	598
14	228542	321	98	66	0	4	1191
15	228541	321	395	49	0	0	4018
23	228541	321	67	15	0	6	750
27	229795	320	318	110	62	5	4216
28	229795	320	4	5	0	2	24
29	229794	320	2	113	0	26	20
41	229794	320	12	12	0	4	126
55	230166	320	303	212	0	4	2994
57	230165	320	6	3	0	0	42
59	230151	320	159	51	0	5	1620
60	230204	320	95	15	0	0	1017
61	230233	320	165	185	0	0	1471
62	230347	320	62	18	0	0	731
63	230418	320	91	45	0	13	756
65	230418	320	30	6	0	0	273
66	230418	320	33	5	0	0	347
67	230611	321	103	21	0	6	1028
72	230641	321	209	180	0	0	1974
74	230645	321	6	4	0	0	68
75	230769	321	21	6	0	0	318
80	230786	323	59	36	0	0	568
85	230786	323	49	8	0	0	517
92	230784	323	33	6	0	0	284

Table H.26.: Commons Collections (Gator network, incremental maintenance)

Revision	Artifacts	Annotations	Build (ms)	Create (ms)	Delete (ms)	Update (ms)	Events
1	228423	25830	4271	528306	0	0	228423
2	228423	25750	505	84	2740	4622	598
14	228542	25750	662	2156	0	1277	1191
15	228541	25752	2490	199	303	9	4018
23	228541	25752	453	85	0	20	750
27	229795	25786	1874	650	322	116	4216
28	229795	25786	20	34	0	12	24
29	229794	25785	12	2559	87	94	20
41	229794	25785	75	106	0	9	126
55	230166	25863	1669	1886	0	37	2994
57	230165	25863	32	20	0	0	42
59	230151	25863	1045	400	172	150	1620
60	230204	25866	653	84	394	229	1017
61	230233	25864	958	1802	823	28	1471
62	230347	25871	394	54	0	4	731
63	230418	25871	424	343	0	51	756
65	230418	25871	179	27	0	0	273
66	230418	25871	212	32	0	4	347
67	230611	25905	560	164	0	9	1028
72	230641	25907	1429	2007	175	56	1974
74	230645	25907	39	17	0	0	68
75	230769	25907	130	33	0	0	318
80	230786	25991	374	358	218	987	568
85	230786	25991	329	84	0	0	517
92	230784	25991	210	32	44	90	284

Table H.27.: Commons Collections (Rete network, incremental maintenance)

H.3. Comparison with EMF-IncQuery

Pattern Name	Proposed Approach (sec)	IncQuery (sec)	Speedup
Composite1	0,26	0,25	0,96
DefaultConstructor	0,2	0,2	1,00
ExtractInterface	0,28	0,24	0,86
FieldAssignment2	0,17	0,21	1,24
Generalization	0,18	0,2	1,11
InterfaceImplementation	0,18	0,2	1,11
InterfaceImplementationWithGeneralization	0,21	0,19	0,90
InterfaceImplementationWithGeneralization2	0,25	0,2	0,80
MultiLevelGeneralization1	0,16	0,19	1,19
MultiLevelGeneralization2	0,18	0,19	1,06
MultiLevelGeneralization3	0,2	0,19	0,95
OwnFieldReference	0,19	0,19	1,00
PrivateConstructor	0,17	0,19	1,12
PrivateField	0,19	0,19	1,00
ProtectedField	0,18	0,18	1,00
PublicField	0,19	0,18	0,95
PublicInstanceField	0,25	0,22	0,88
PublicInstanceMethod	0,19	0,19	1,00
PublicMethod	0,18	0,21	1,17
ReadOperation	0,25	0,19	0,76
Singleton1	0,23	0,19	0,83
Singleton2	0,19	0,18	0,95
Singleton3	0,2	0,19	0,95
TypedElement	0,19	0,19	1,00
WriteOperation	0,47	0,21	0,45

Table H.28.: Maintenance time of EMF-IncQuery and proposed approach

Pattern Name	Proposed Approach (bytes)	IncQuery (bytes)	Memory Reduction
Composite1	12544	33936	2,71
DefaultConstructor	7792	7480	0,96
ExtractInterface	9736	17360	1,78
FieldAssignment2	11264	21416	1,90
Generalization	8200	9256	1,13
InterfaceImplementation	8200	6904	0,84
InterfaceImplementationWithGeneralization	9400	15168	1,61
InterfaceImplementationWithGeneralization2	11784	25512	2,16
MultiLevelGeneralization1	9400	19664	2,09
MultiLevelGeneralization2	11784	33168	2,81
MultiLevelGeneralization3	17128	42536	2,48
OwnFieldReference	13072	38112	2,92
PrivateConstructor	8128	7768	0,96
PrivateField	8992	10304	1,15
ProtectedField	8992	9584	1,07
PublicField	8992	9760	1,09
PublicInstanceField	12024	26920	2,24
PublicInstanceMethod	10936	23416	2,14
PublicMethod	8592	13760	1,60
ReadOperation	12056	24760	2,05
Singleton1	12024	27312	2,27
Singleton2	11928	28080	2,35
Singleton3	12024	27112	2,25
TypedElement	13480	38080	2,82
WriteOperation	79280	30376	0,38

Table H.29.: Memory Consumption of EMF-IncQuery and proposed approach

H.4. Evaluation Discussion

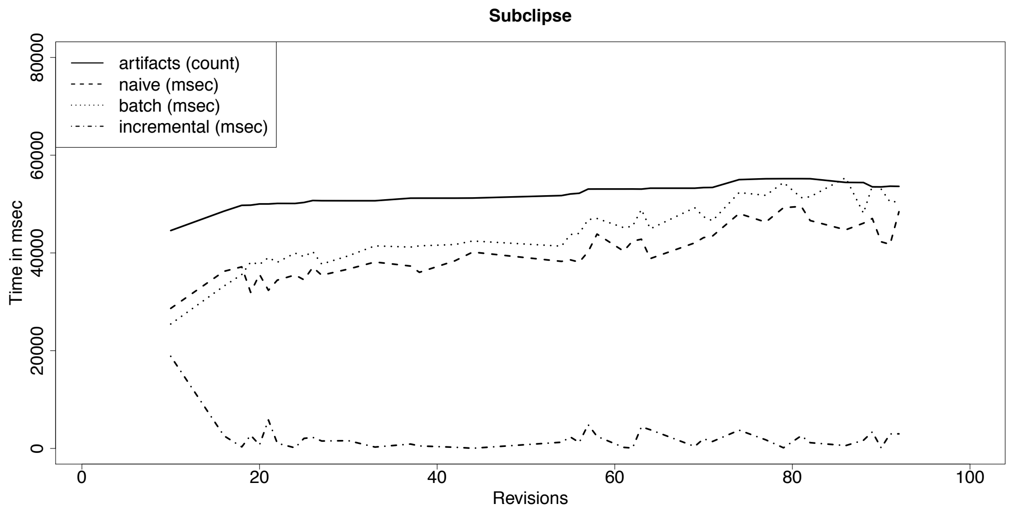


Figure H.62.: Comparison of maintenance algorithms for Subclipse data set

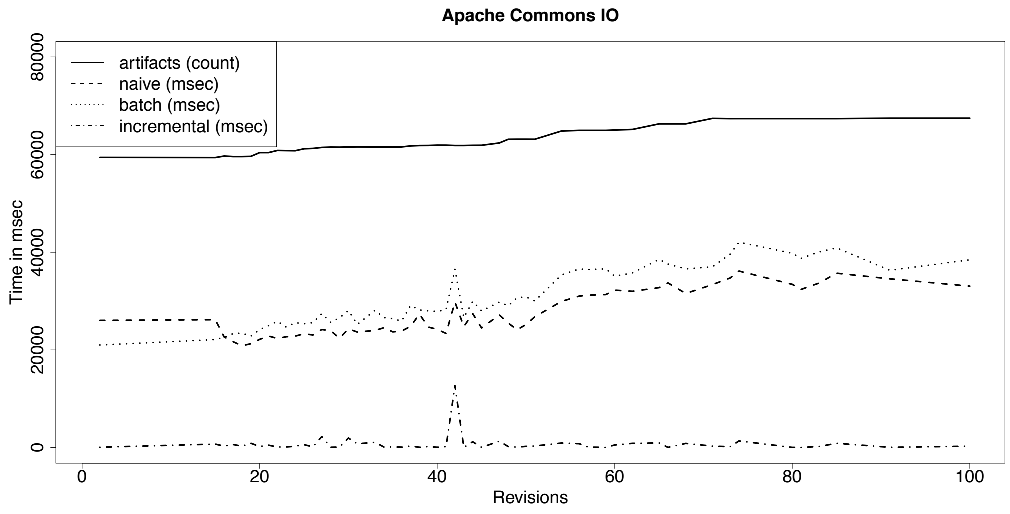


Figure H.63.: Comparison of maintenance algorithms for Apache Commons IO data set

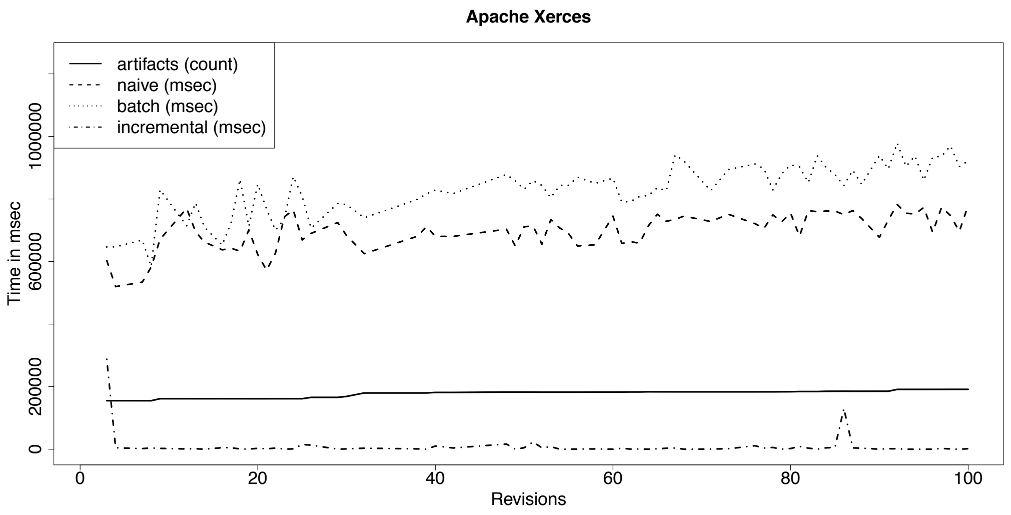


Figure H.64.: Comparison of maintenance algorithms for Apache Xerces data set

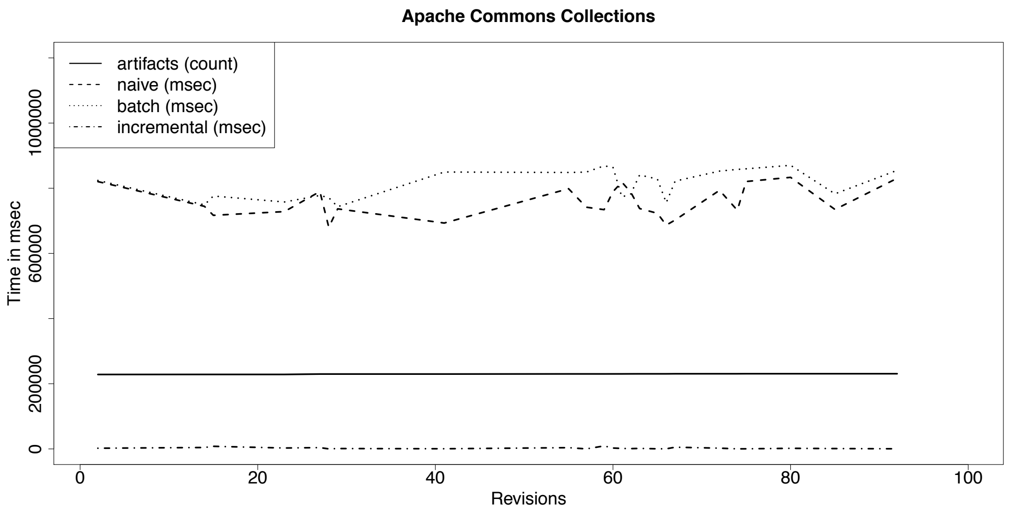


Figure H.65.: Comparison of maintenance algorithms for Apache Commons Collections data set

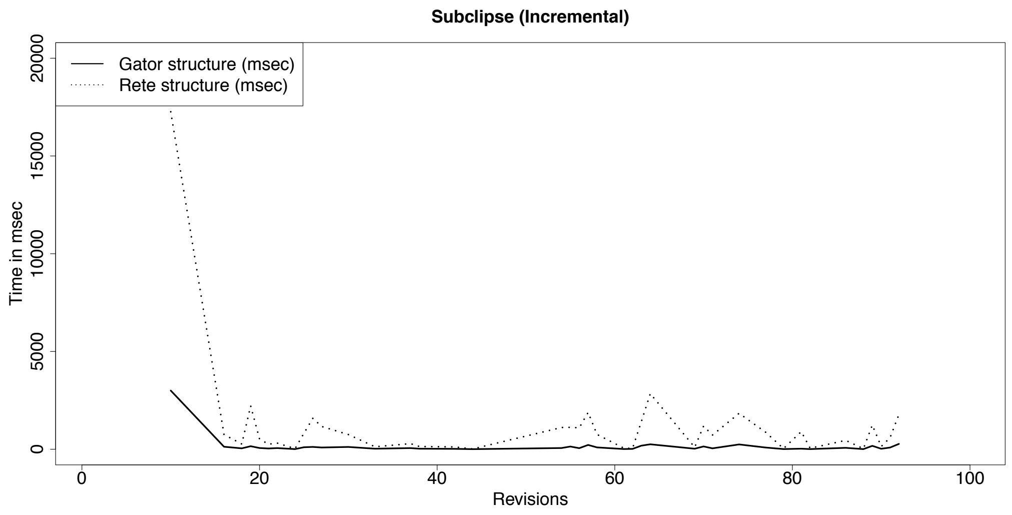


Figure H.66.: Performance comparison of Gator and Rete network structure for Subclipse

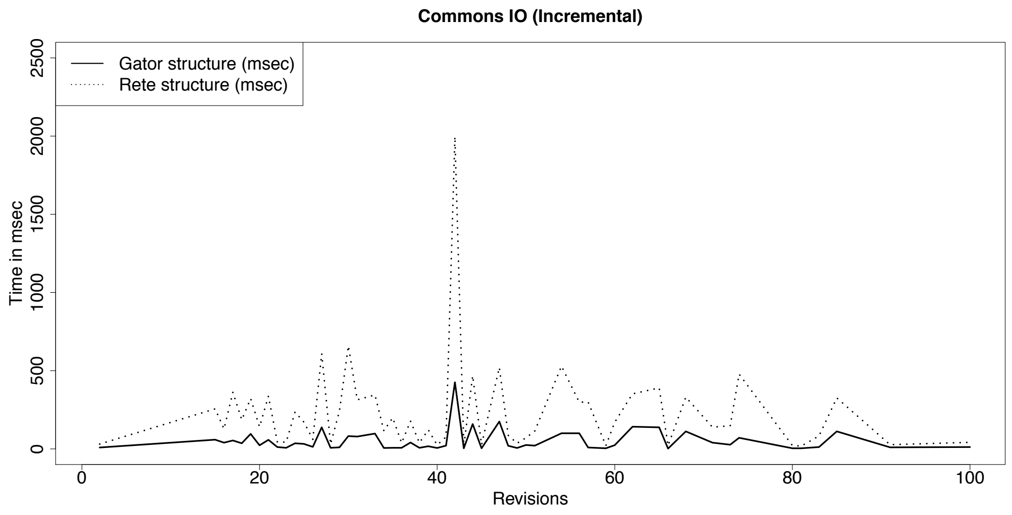


Figure H.67.: Performance comparison of Gator and Rete network structure for Apache Commons IO

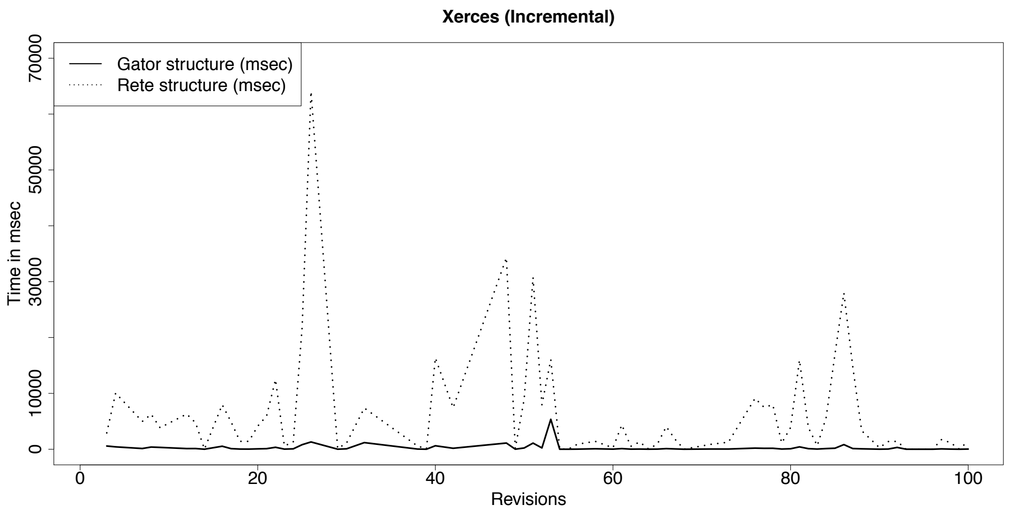


Figure H.68.: Performance comparison of Gator and Rete network structure for Apache Xerces

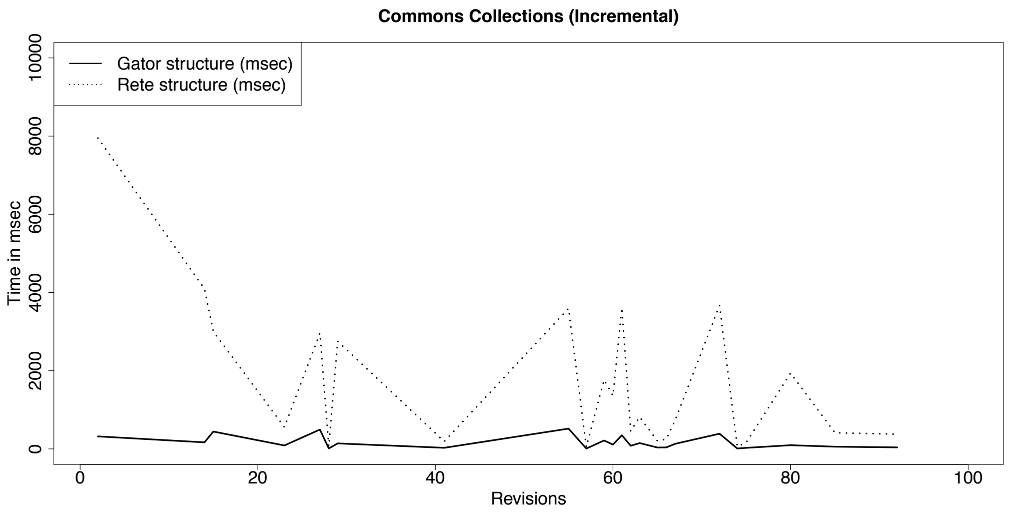


Figure H.69.: Performance comparison of Gator and Rete network structure for Apache Commons Collections

H.5. EMF-IncQuery Graph patterns

```
pattern PrivateField(field :  Field) {
  Field.annotationsAndModifiers(field, privateModifier);
  Private(privateModifier);
}
```

(a) Private field

```
pattern ProtectedField(field :  Field) {
  Field.annotationsAndModifiers(field, ProtectedModifier);
  Protected(ProtectedModifier);
}
```

(b) Protected field

```
pattern PublicField(field :  Field) {
  Field.annotationsAndModifiers(field, PublicModifier);
  Public(PublicModifier);
}
```

(c) Public field

Figure H.70.: Visibility of fields

```
pattern ReferenceContainer(reference :  Reference) {
  Reference.next(_, reference);
}
```

(a) Reference container

```
pattern FieldReferenceA(identifierReference :  IdentifierReference, field :  Field) {
  IdentifierReference.target(identifierReference, field);
  neg find ReferenceContainer(identifierReference);
}
```

(b) Field reference without this keyword

```
pattern FieldReferenceB(selfReference :  SelfReference, field :Field) {
  SelfReference.next(selfReference, identifierReference);
  IdentifierReference.target(identifierReference, field);
  neg find ReferenceContainer(selfReference);
}
```

(c) Field reference with this keyword

Figure H.71.: Field references

```
pattern WriteOperation(method : Method) {
  find FieldAssignment(assignment, assignmentField);
  find PrivateField(assignmentField);
  find PublicMethod(method, _);
  ClassMethod.statements(method, expression);
  ClassMethod.parameters(method, parameter);
  OrdinaryParameter(parameter);
  ExpressionStatement.expression(expression, assignment);
  AssignmentExpression.value(assignment, assignmentValue);
  IdentifierReference.target(assignmentValue, parameter);
}
```

(a) Write Operation

```
pattern ReadOperation(method : Method, privateField : Field) {
  find PublicMethod(method, _);
  Class.members(clazz, method);
  ClassMethod.statements(method, returnStatement);
  Return.returnValue(returnStatement, reference);
  find PrivateField(privateField);
  Class.members(clazz, privateField);
  find FieldReferenceA(reference, privateField);
} or {
  find PublicMethod(method, _);
  Class.members(clazz, method);
  ClassMethod.statements(method, returnStatement);
  Return.returnValue(returnStatement, reference);
  find PrivateField(privateField);
  Class.members(clazz, privateField);
  find FieldReferenceB(reference, privateField);
}
```

(b) Read Operation

Figure H.72.: Getter and setter

```
pattern Generalization(subClass : Class, superClass : Class) {
  Class.^extends(subClass, classifier);
  NamespaceClassifierReference.classifierReferences(classifier, namespace);
  ClassifierReference.target(namespace, superClass);

  NamespaceClassifierReference(classifier);
  ClassifierReference(namespace);
}
```

(a) Generalization

```
pattern MultiLevelGeneralization(subClass : Class, superClass : Class) {
  Class(classInTheMiddle);
  find Generalization(subClass, classInTheMiddle);
  find Generalization(classInTheMiddle, superClass);
} or {
  Class(classInTheMiddle);
  find MultiLevelGeneralization(subClass, classInTheMiddle);
  find Generalization(classInTheMiddle, superClass);
} or {
  Class(classInTheMiddle);
  find MultiLevelGeneralization(subClass, classInTheMiddle);
  find MultiLevelGeneralization(classInTheMiddle, superClass);
} or {
  Class(classInTheMiddle);
  find Generalization(subClass, classInTheMiddle);
  find MultiLevelGeneralization(classInTheMiddle, superClass);
}
```

(b) Multi-Level Generalization

```
pattern InterfaceImplementation(class : Class, interface : Interface) {
  Class.implements(class, namespaceClassifier);
  NamespaceClassifierReference.classifierReferences(namespaceClassifier, classifierReference);
  ClassifierReference.target(classifierReference,interface);
}
```

(c) Interface Implementation

```
pattern InterfaceImplementationWithGeneralization(class : Class, interface : Interface) {
  find Generalization(class, superClass);
  find InterfaceImplementation(superClass, interface);
} or {
  find MultiLevelGeneralization(class, superClass);
  find InterfaceImplementation(superClass, interface);
}
```

(d) Multi-Level Interface Implementation

Figure H.73.: Hierarchies

```
pattern FieldAssignment(assignmentExpression :  AssignmentExpression, field :  Field) {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceA(firstReference, field);
  find PrivateField(field);
  Class.members(_, field);
} or {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceA(firstReference, field);
  find ProtectedField(field);
  Class.members(_, field);
} or {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceA(firstReference, field);
  find PublicField(field);
  Class.members(_, field);
} or {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceB(firstReference, field);
  find PrivateField(field);
  Class.members(_, field);
} or {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceB(firstReference, field);
  find ProtectedField(field);
  Class.members(_, field);
} or {
  AssignmentExpression.assignmentOperator(assignmentExpression, assignment);
  AssignmentExpression.child(assignmentExpression, firstReference);
  Assignment(assignment);
  find FieldReferenceB(firstReference, field);
  find PublicField(field);
  Class.members(_, field);
}
```

(a) Field Assignment

Figure H.74.: Assignments

```
pattern ArrayField(field :  Field, targetClassifier :  ConcreteClassifier) {
  Field.arrayDimensionsBefore(field, array);
  Field.typeReference(field, namespace);
  NamespaceClassifierReference.classifierReferences(namespace, reference);
  ClassifierReference.target(reference, targetClassifier);

  ArrayDimension(array);
  NamespaceClassifierReference(namespace);
  ClassifierReference(reference);
}
```

(a) ToN associations

```
pattern ListField(list :  Field, targetClassifier :  ConcreteClassifier) {
  Field.typeReference(list, namespace);
  NamespaceClassifierReference.classifierReferences(namespace, reference);
  ClassifierReference.target(reference, referenceTarget);
  ClassifierReference.typeArguments(reference, type);
  QualifiedTypeArgument.typeReference(type, typeNamespace);
  NamespaceClassifierReference.classifierReferences(typeNamespace, typeReference);
  ClassifierReference.target(typeReference, targetClassifier);

  NamespaceClassifierReference(namespace);
  ClassifierReference(reference);
  QualifiedTypeArgument(type);
  ConcreteClassifier(referenceTarget);
  NamespaceClassifierReference(typeNamespace);
  ClassifierReference(typeReference);
}
```

(b) ToMany associations

Figure H.75.: Associations

```
pattern TypedElementA(type :  Type, typedElement :  TypedElement) {
  TypedElement.typeReference(typedElement, type);
  PrimitiveType(type);
}
```

(a) Primitive types

```
pattern TypedElementB(type :  Type, typedElement :  TypedElement, typeArgumentable :  TypeArgumentable)
{
  TypedElement.typeReference(typedElement, namespaceClassifier);
  NamespaceClassifierReference.classifierReferences(namespaceClassifier, typeArgumentable);
  ClassifierReference.target(typeArgumentable, type);
  NamespaceClassifierReference( namespaceClassifier);
}
```

(b) Classifier types

Figure H.76.: Typed elements

```
pattern PrivateConstructor(constructor :  Constructor) {
  Constructor.annotationsAndModifiers(constructor, modifiier);
  Private(modifiier);
}
```

(a) Private constructor

```
pattern PublicInstanceField(classField :  Member) {
  Field.annotationsAndModifiers(classField, publicModifier);
  Public(publicModifier);
  Field.annotationsAndModifiers(classField, staticModifier);
  Static(staticModifier);
  Field.initialValue(classField, call);
  NewConstructorCall.typeReference(call, namespace);
  NamespaceClassifierReference.classifierReferences(namespace, classifier);
  ClassifierReference.target(classifier, class);
  Class.members(class, classField);
}
```

(b) Public instance field

```
pattern PublicInstanceMethod(classMethod :  Member) {
  find TypedElementA(class, classMethod);
  find PublicMethod(classMethod, _);

  Class.members(class, classMethod);
  ClassMethod.annotationsAndModifiers(classMethod, staticModifier);
  Static(staticModifier);
} or {   find TypedElementB(class, classMethod, _);
  find PublicMethod(classMethod, _);

  Class.members(class, classMethod);
  ClassMethod.annotationsAndModifiers(classMethod, staticModifier);
  Static(staticModifier);
}
```

(c) Public instance method

```
pattern Singleton(class :  Class) {
  find PrivateConstructor(constructor);
  find PublicInstanceMethod(member);
  Class.members(class, constructor);
  Class.members(class, member);
} or {
  find PrivateConstructor(constructor);
  find PublicInstanceField(member);
  Class.members(class, constructor);
  Class.members(class, member);
}
```

(d) Singleton

Figure H.77.: Singleton design pattern

```
pattern Composite(subClazz :  Class, superClazz:  Class) {
  find Generalization(subClazz, superClazz);
  find ArrayField(field, superClazz);
  Class.members(subClazz, field);
} or {
  find Generalization(subClazz, superClazz);
  find ListField(field, superClazz);
  Class.members(subClazz, field);
} or {
  find MultiLevelGeneralization(subClazz, superClazz);
  find ArrayField(field, superClazz);
  Class.members(subClazz, field);
} or {
  find MultiLevelGeneralization(subClazz, superClazz);
  find ListField(field, superClazz);
  Class.members(subClazz, field);
}
```

(a) Composite

Figure H.78.: Composite design pattern

```
pattern ConstructorWithParameters(constructor :  Constructor) {
  Constructor.parameters(constructor, _);
}

pattern DefaultConstructor(constructor :  Constructor) {
  neg find ConstructorWithParameters(constructor);
}
```

(a) Default Constructor

```
pattern ExtractInterface(class :  Class) {
  neg find InterfaceImplementation(class, _);
  neg find InterfaceImplementationWithGeneralization(class, _);
  Class.members(class, method);
  Method.annotationsAndModifiers(method, public);

  Method(method);
  Public(public);
}
```

(b) Extract Interface

Figure H.79.: Simple and complex negations

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit mit dem Titel “A Framework for Incremental View Graph Maintenance” selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde zuvor in keiner anderen Hochschule und in keinem anderen Studiengang als Prüfungsleistung eingereicht.

Potsdam, 30.12.2017 Unterschrift: ....

Thomas Beyhl

Statutory Declaration

I declare that I have authored this thesis entitled “A Framework for Incremental View Graph Maintenance” independently, that I have not used other than the referenced sources and resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. This thesis was neither submitted to another university nor to another course of study.

Potsdam, 2017/12/30 Signature: ....

Thomas Beyhl