Reverse Engineering Using Graph Queries

Jürgen Ebert, Daniel Bildhauer
University of Koblenz-Landau
{ebert,dbildh}@uni-koblenz.de

Abstract

Software Reverse Engineering is the process of extracting (usually more abstract) information from software artifacts. Graph-based engineering tools work on fact repositories that keep all artifacts as graphs. Hence, information extraction can be viewed as querying this repository. This paper describes the graph query language GReQL and its use in reverse engineering tools.

GReQL is an expression language based on set theory and predicate logics including regular path expressions (RPEs) as first class values. The GReQL evaluator is described in some detail with an emphasis on the efficient evaluation of RPEs for reachability and path-finding queries. Applications for reverse engineering Java software are added as sample use cases.

1 Introduction

In software engineering, *modeling* is the process of abstracting parts of reality into a representation that abstracts from unnecessary details and allows automatic processing. Models may be either *descriptive*, if they represent existing artifacts, or *prescriptive*, if they are used as a blueprint for artifacts to be constructed.

In the domain of software engineering tools, a *modeling approach for software engineering artifacts* has to be chosen. The earliest tools for handling software artifacts were compilers which extracted tree models from program artifacts by a combined procedure of scanning and parsing, leading to *abstract syntax trees* as program models. Trees are easily representable in main memory, and there are many efficient algorithms for handling trees.

In general, trees are not powerful enough to keep all the necessary information about software engineering artifacts in an integrated form. As an example, definition-use chains or other additional links between the vertices of a syntax tree lead to more general graph-like structures. As a consequence of these shortcomings of trees Manfred Nagl [Nag80] proposed to use *graphs* to represent source code in compilers and beyond that for all other artifacts in software engineering environments.

In this paper, the *TGraph approach* to graph-based modeling [ERW08] is used to model software artifacts in software engineering tools. The generic graph query language GReQL is introduced which supports information extraction from graph repositories. An overview on GReQL is given and its central feature,

the regular path expressions, is described in more detail. The focus is on the evaluation of path expressions to derive reachability and path information.

Section 2 describes the use of TGraphs as software models, Section 3 summarizes the necessary definitions on graphs and regular expressions, and Section 4 introduces the query language GReQL. In Section 5 the range of regular path expressions supported by GReQL is introduced, and the algorithms for evaluating these expressions are described in detail. Section 6 gives some examples of GReQL queries in reverse engineering, Section 7 discusses related work including some performance data, and Section 8 concludes the article.

2 Software

A *software system* does not only consist of its source code, but also of all other artifacts that are constructed and deployed during its development and its usage. These additional artifacts comprise a wide range of documents from requirements specifications, design and architecture models, to test cases and installation scripts. These artifacts are written in many different languages, some of them being of textual form others being visual diagrams. Some of them have a formal semantics others remain completely informal.

Graph Representation. To treat such a heterogeneous set of artifacts simultaneously in one tool environment, a *common technological space* is needed which is equally well-suited for storing, analyzing, manipulating and rendering them. Furthermore, it should be possible to also handle inter-artifact links, such as traceability information.

Regarding the work of Manfred Nagl and others, *graphs* form a good basis for such a technological space, since they are simultaneously able to model structure in a generic way and to also keep application-specific additional knowledge in supplementary properties like e.g. attributes.

Generally, every relevant *entity* in a software artifact can be modeled by a representative vertex that represents this entity inside a graph. Then, every (binary) *relationship* between entities can be modeled by an edge that carries all information relevant for this relationship. Note that the occurrence of an object o in some context c is a relationship in this sense, whereas o and c themselves are entities. Thus, occurrences of software entities in artifacts can be modeled by edges between respective vertices.

Example. As an example for the representation of a software artifact as a graph, the following listing shows a simple Java data structure for binary trees. Using a parser, this code can be transformed into an *abstract syntax graph (ASG)*, which consists of about 75 vertices and 100 edges for the code shown, if a fine-granular model is built.

Listing 1.1. "Java implementation of a binary tree"

```
1  class Node {
2    String data;
3    Node left, right;
4
5    public Node(String s) { data = s; }
6
7    public void add(String s) {
8      if (s.compareTo(data) < 0) {
9        if (left != null) left.add(s);
10       else left = new Node(s);
11     } else ...
12   }
13 }
```

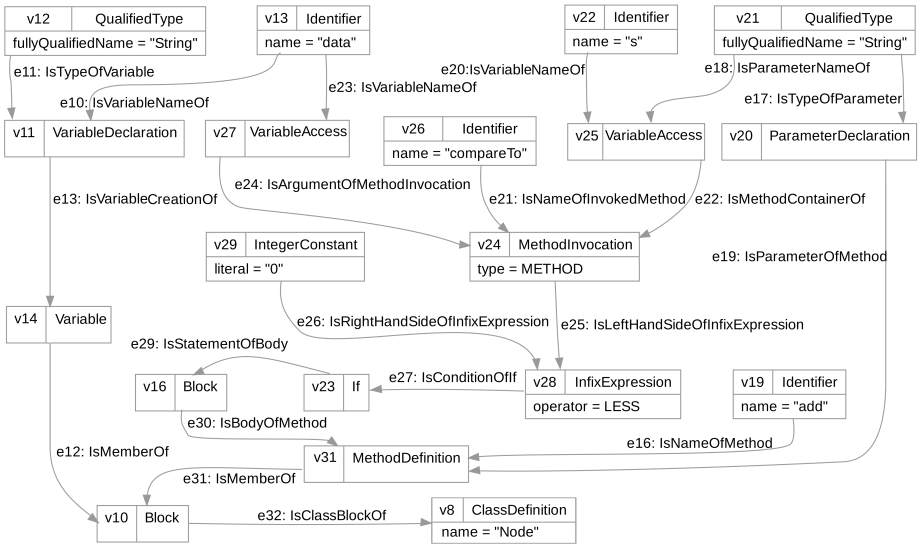


Fig. 1. Extract of the ASG for the class Node from listing 1.1

For brevity reasons, Figure 1 shows only a part of this graph, namely the definition of the class and its `data` attribute as well as the `add`-method with its outer `if` statement. The representation of the class `Node` itself is vertex `v8` at the bottom of the figure. It has the type `ClassDefinition` and holds the class name in its attribute called `name`. Methods and attributes of the class are grouped in a block represented by the vertex `v10`, which is connected to the class definition by the edge `e32`. In the same way, other elements of the source code are represented by vertices and edges connecting them. Analogically to vertex `v8`, all other vertices and edges are typed and may have attributes. For the sake of

clarity, the edge attributes keeping e.g. the position of the occurrences in the source code are omitted. Here, the ASG is a directed acyclic graph (and not only a tree). Every entity is represented only once but may occur several times (e.g. vertex `v13`).

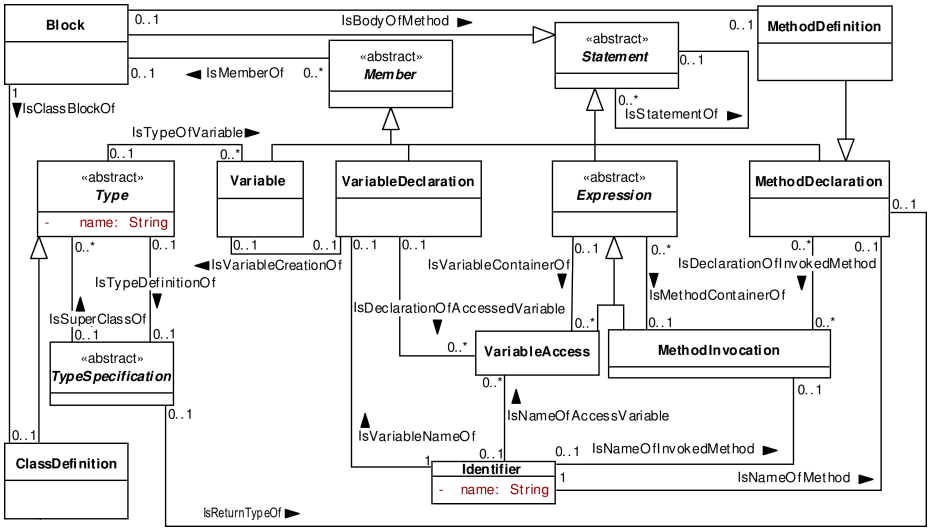


Fig. 2. Extract of the Graph Schema for the Java language

Graph schema. The types and names of the attributes depend on the types of the vertices and edges of the graph. These types can be described by a *graph schema* which acts as a metamodel of the graph. Since the Meta Object Facility (MOF) [Obj06] is a widely accepted standard for metamodeling, Figure 2 shows a small part of a metamodel for ASGs representing Java programs depicted as a MOF compatible UML class diagram. The classes denote vertex types while the associations between classes denote edge types. Thus, class diagrams may be used to describe graph schemas.

The vertex type `ClassDefinition` mentioned above is depicted at the left border of the diagram. It is a specialization of the vertex type `Type` and inherits the `name`-attribute from this class. Every class consists of a `Block`, which is connected to the class by a `IsClassBlockOf` edge as it was depicted in the example graph above. A `Block` is a special kind of a `Statement` and groups other `Statement`s. The edge type `IsStatementOf` represents the general occurrence of one statement in another and is the generalization of all other edge types which group statements in each other. These generalizations are omitted in the diagram for reasons of brevity but are important in the queries shown later in this paper.

Graph-based Tools. Schemas can be defined for all kinds of languages. There are schemas for textual programming languages, and there are schemas for visual languages. Schemas can be derived on all levels of granularity from fine-granulare abstract syntax over middle level abstractions to coarse architecture descriptions depending on their purpose. Different schemas can also be combined to larger integrated schemas. Metamodel Engineering is the discipline that deals with topic.

Given a graph schema, tools can work on graphs conforming to it. These tools extract the relevant facts from software engineering artifacts (e.g. by parsing) and store them as graphs in a graph repository. Several services on the repository (like enrichment, analysis, abstraction, and transformation) help to use these graphs for solving software engineering problems. These services may be application specific or generic.

One of the services to gather information from the repository is *querying*. In the following, this paper focuses on the generic query language GReQL. GReQL works on TGraphs, which are introduced in the following section.

3 Terminology

To describe the graph-based approach to build reengineering tools in more detail, an appropriate terminology is needed. This section introduces TGraphs and a suitable notation for algorithms on TGraphs. Furthermore, some facts about regular languages are enumerated, which are needed later for the treatment of regular path expressions.

3.1 Graphs

To establish a comprehensive graph technology on a formal basis, a precise definition of its underlying concepts is essential. In this paper, TGraphs are used.

TGraphs. TGraphs are a powerful category of graphs which are able to model not only structural connections, but also all type and attribute information needed for an object-based view on the represented model. TGraphs are typed, attributed, and ordered directed graphs, i.e. all graph elements (vertices and edges) are typed and may carry type-dependent attribute values. Furthermore, there are orderings of the vertex and the edge sets of the graph and of the incidences at all vertices. Lastly, all edges are assumed to be directed.

Definition: TGraph

Let
- *Vertex be the universe of* vertices,
- *Edge be the universe of* edges,
- *TypeId be the universe of* type identifiers,
- *AttrId be the universe of* attribute identifiers, *and*
- *Value be the universe of* attribute values.

Assuming two finite sets,
- a vertex set $V \subseteq Vertex$ *and*
- an edge set $E \subseteq Edge$,

be given. $G = (Vseq, Eseq, \Lambda seq, type, value)$ *is a* TGraph *iff*
- $Vseq \in \mathsf{iseq}\, V$ *is a permutation of* V,
- $Eseq \in \mathsf{iseq}\, E$ *is a permutation of* E,
- $\Lambda seq : V \to \mathsf{iseq}(E \times \{in, out\})$ *is an* incidence function *where*
  $\forall\, e \in E : \exists! v, w \in V : (e, out) \in ran\, \Lambda seq(v) \wedge (e, in) \in ran\, \Lambda seq(w)$,
- $type : V \cup E \to TypeId$ *is a* type function, *and*
- $value : V \cup E \to (AttrId \twoheadrightarrow Value)$ *is an* attribute function *where*
  $\forall\, x, y \in V \cup E : type(x) = type(y) \Rightarrow dom(value(x)) = dom(value(y))$.

Thus, a TGraph consists of an ordered vertex set V and an ordered edge set E. They are connected by the incidence function Λseq which assigns the sequence of its incoming and outgoing edges to each vertex. For a given edge e, $\alpha(e)$ and $\omega(e)$ denote its *start vertex* and *target vertex*, respectively. Furthermore, all elements (i.e. vertices and edges) have a type and carry a type dependent set of attribute-value pairs. Figure 1 visualizes an example TGraph.

Further Graph Properties. Given a TGraph G, paths are used to describe how a given vertex w may be reached from a vertex v.

Definition: Path

A path *from* v_0 *to* v_k *in a TGraph* G *is an alternating sequence*
$C =< v_0, e_1, v_1, ..., e_k, v_k >, k \geq 0,$
of vertices and edges, where
$\forall\, i \in \mathbb{N}, 1 \leq i \leq k : \alpha(e_i) = v_{i-1} \wedge \omega(e_i) = v_i.$
v_0 *is called the* start vertex $\alpha(C)$ *and* v_k *the* target vertex $\omega(C)$ *of the path.*
A path is called a proper path, *if all its vertices are distinct.*

Definition: Derived Edge Type Sequence

Given a path C *the corresponding sequence of edge types*
$< type(e_1), type(e_2), ..., type(e_k) >$
is called its derived edge type sequence.

The existence of an edge from v to w is denoted by $v \to w$, and the existence of a path by $v \to^* w$. Furthermore, $v \to^*$ denotes the set of all vertices reachable from v by any path.

3.2 Pseudocode

Given an appropriate data structure for TGraphs [Ebe87], graph algorithms can be implemented in a such a way that graph traversals are efficient. There is a Java-API called JGraLab[1] that allows a convenient and concise notation of algorithms which is very near to pseudo code.

[1] http://www.ohloh.net/p/jgralab

The pseudo code used in this article adheres to JGraLab's conventions. There is a type `Graph`, and graph elements are instances of the types `Vertex` and `Edge`, respectively. A few of the API operations are listed here:

```
interface Vertex {
  /** @return the sequence of outgoing edges at this vertex */
  Iterable<Edge> getAllOutEdges ();
  ...
}
```

```
interface Edge {
  /** @return the start vertex of this edge */
  Vertex getAlpha ();

  /** @return the end vertex of this edge */
  Vertex getOmega ();
  ...
}
```

According to these interfaces, the edges incident to a vertex v can be traversed in the order defined by $\Lambda seq(v)$ using a `for`-loop like

```
for (Edge e: v.getAllOutEdges()) {
  // process edge e
}
```

3.3 Regular Expressions and Automata

Since regular expressions are in center of the query language described in this article, some basic facts about finite automata and regular expressions are compiled here ([HU79]).

Definition: Regular Expression

Given some alphabet Σ, the regular expressions (REs) *over Σ are defined inductively as follows:*
(1) Φ is a regular expression and denotes the empty set $\varnothing$.
(2) ϵ is a regular expression and denotes the set $\{\epsilon\}$.
(3) For each $a \in \Sigma$, a is a regular expression and denotes the set $\{a\}$.
(4) If r and s are regular expressions denoting the languages R and S, respectively, then concatenation (rs), choice $(r + s)$, and closure r^* are regular expressions that denote the sets RS, $R \cup S$, and R^*, respectively.
The set of strings denoted by a regular expression r is called $L(r)$, the language *described by r.*

Languages described by regular expressions can also be described by deterministic finite automata.

Definition: Deterministic Finite Automaton (DFA)

A deterministic finite automaton (DFA) $dfa = (S, \Sigma, \delta, s_0, F)$ *over* Σ *consists of*

- *a set* S *of states,*
- *a transition function* $\delta : S \times \Sigma \to S$*,*
- *a start state* s_0*, and*
- *a set* $F \subseteq S$ *of terminal states.*

DFAs can be visualized as graphs, where the vertices are the states of the automaton and where there are edges $e = s_1 \to_a s_2$ iff δ maps (s_1, a) to s_2. An example is shown in figure 4.

To use automata in software, they may be implemented as graphs, as well. Only a few methods on automata are used in this paper:

```
1  interface Automaton extends Graph{
2    /** @return the state state
3    Vertex getStartState ();
4
5    /** @return true iff s is a terminal state
6    boolean isTerminal (Vertex s);
7
8    /** @return the sequence of all enabled transitions of type t out of state s */
9    Iterable<Edge> getAllEnabledTransitions (Vertex s, Type t);
10   ...
11 }
```

An automaton *dfa* accepts a string l over an alphabet Σ by a state $s \in S$ iff l is the derived type sequence of a path from s_0 to s in the graph of *dfa*. The set of strings accepted by s is called $L(dfa, s)$. Consequently, the *language* accepted by an automaton *dfa* is $L(dfa) := \cup_{s \in F} L(dfa, s)$, where F is the set of terminal states.

It is well known that every language accepted by some finite automaton is also describable by a regular expression and vice versa [HU79]. It is possible to construct an equivalent automaton $dfa(r)$ from a given regular expression r using Thompson's construction [Tho68] followed by Myhill's algorithm [Myh57].

Though in theory the size of the deterministic finite automaton built from a given regular expression r of size n can be of the order 2^n, this 'explosion' does not occur frequently in practice [ASU87]. Experience shows that DFA-acceptors for regular languages have usually only about twice the size of their regular expressions.

4 Querying

Since graphs are subject to algorithms, all decidable problems on graphs can in principle be solved by respective graph algorithms. This approach affords a lot of effort on the user side, since each request for information about the model represented by a graph leads to a specific problem which has to be solved

algorithmically. Much of this work can be avoided in practice by supplying the users with a powerful *querying facility* that allows easy end user retrieval for a large class of information needs.

GReQL. In the case of TGraphs, the *Graph Repository Query Language GReQL* supplies such a powerful facility for end user querying. GReQL was developed in the GUPRO project [EKRW02] at the University of Koblenz-Landau. The current version GReQL 2 was defined by Marchewka [Mar06]. There is a full optimizing evaluator for GReQL available on JGraLab implemented by Bildhauer and Horn [Bil08,Hor09].

GReQL can be characterized as a schema-sensitive expression language with dynamic types. It gives access to all graph properties and to the schema information. GReQL queries may be parameterized. Its most valuable feature is an elaborated concept for path expressions.

Since GReQL is an expression language, its *semantics* is self-evident. Every GReQL query is a GReQL expression e which is either atomic or recurs to some partial expressions $e_1, ..., e_k$ whose evaluation is used to compose the resulting value of e. Thus, using mathematical expressions as building blocks (see below) the value of a GReQL expression is precisely defined.

The type-correctness of GReQL expressions is checked at evaluation time. The type system itself is encapsulated in a composite Java class called `JValue` (see below) in the context of JGraLab.

Querying. The goal of *querying* is to extract information from graph-based models and to make the extracted information available to clients, such as software engineers or software engineering tools.

Querying is a *service* on TGraphs that - given a graph g and a GReQL query text q - delivers a (potentially composite) object, that represents the query's result. The universe of possible query results is given by `JValue`.

Specification: Querying

Signature:
$query : TGraph \times String \nrightarrow JValue$

Pattern:
$s = query(g, q)$

Input:
a TGraph g and a valid GReQL query q

Output:
a JValue s which contains the result of evaluating q on g (according to the semantics of GReQL)

Types. The types of GReQL values are defined by the Java class `JValue`, since the values delivered by GReQL queries are usually passed on to Java software, e.g. for rendering the result. `JValue` is a union type which comprises all possible types

that a GReQL result may have. Its structure is that of a composite, i.e. there are some basic (atomic) types and some composite types that allow structured values.

Basic types are `integer`, `double`, `boolean`, and `string`, and also a concept for enumeration types is supplied. Some graph constituents are also supported as types, namely `vertex`, `edge` and `type`, the former referring to vertex or edge types in the schema.

Composite types allow structured data (tuples, lists, sets, bags, tables, maps, and records). Also paths, path-systems (see below) and graphs are supported as types as well as path expressions.

Expressions. Since GReQL is an *expression language* the whole language can be explained and implemented along its different kinds of expressions:

```
1  0, 123                        // integer literals
2  0.0, -2.1e23                  // double literals
3  true, false                   // boolean literals
4  "hugo", "ab\n"                // string literals
5  v                             // variable expression
6  let x := 3 in x + y           // let-expression
7  x + y where x := 3            // where-expression
8  sqr(5)                        // function application
9  not true                      // unary operator expression
10 b and c, x > 0                // binary operator expressions
11 v.name                        // value access
12 x > 5 ? 7 : 9                 // conditional expression
13
14 // quantified expressions
15 exists v:V{MethodDeclaration} @ outDegree(v)=0
16 forall e:E{IsCalledByMethod} @ alpha(e) = omega(e)
17
18 // FWR expression
19 from caller, callee:V{MethodDeclaration}                //(1)
20 with caller <--{IsBodyOfMethod} <--{IsStatementOf}*     //(2)
21            <--{IsDeclarationOfInvokedMethod} callee
22 report caller, callee end                               //(3)
```

For all basic types (`integer`, `double`, `boolean`, `string`, `enum`) appropriate notations for literals are defined (lines 1 to 4). Variables stand for the value which is bound to them (line 5). Since GReQL is a *single assignment language*, variables may be bound only once in a given scope. All attribute and type identifiers from the schema are predefined variables. Local scopes may be formed using `let`- or `where`-constructs (lines 6 and 7).

Composite expressions may be constructed using function applications (line 8) or using unary or binary operators (lines 9 to 10), including a conditional clause . (Besides the usual operators known for the basic types, there are also

special operators, called regular path descriptions (RPEs) which are themselves expressions. RPEs and their usage are described in more detail in Section 5.) Attributes of graph elements are accessed using a dot notation (line 11).

Since GReQL is heavily based on set-like expressions, also quantified expressions can be used using `exists`- and `forall`-quantifiers (lines 15 to 16). Here the restriction holds that the domains of the bound variables have to be finite.

The last form of expressions supplied by GReQL is the from-with-report(FWR) expression (lines 19 to 22). Its result is a bag of values (or of tuples of values, depending on the length of the report list) containing the results of the expressions in the report clause (line 22) evaluated for each variable binding in the declaration (line 19) which fulfills the condition of the with clause (lines 20 and 21) . Alternatively the result may also have the form of a table, a set or a map.

Function Library. The GReQL types go along with a set of operations that can be applied on their instances. Besides the usual standard operations on basic types, there is also a long list of operations on graph elements (like `degree()`, `alpha()`, `omega()`, etc.) and aggregation operations (like `avg()`, `count()`, etc.). These operations are kept in an editable function library which can be extended easily. At present it contains about 100 functions.

Assuming that all functions in the library are polynomial in the size of the graph, all GReQL expressions can be evaluated in polynomial time, if all variables in quantified and FWR expressions are bound to sets whose size is also polynomial in the graph size.

5 Path Expressions

The language described up to now supports the evaluation of expressions in the usual domains of arithmetics, boolean values, and strings and thus gives a framework for extracting information from graph elements. But it does not yet give enough support for structure dependent information extraction.

Support for such a kind of information extraction is given by GReQL's so-called *regular path expressions*. To exploit the knowledge encoded in the structural part of a TGraph, connection patterns can be described by regular expressions over the set of element types.

5.1 Definitions

Simplified Syntax. Path expressions allow the comprehensive description of the sets of all edges that have the same derived edge type sequence. GReQL uses *regular* path expressions as a means to support navigation in queries.

Definition: Regular Path Expressions (simplified)

A regular path expression (rpe) *is a non-empty regular expression over the set $EdgeTypeID \subseteq TypeID$ of all edge types, according to the following rules*
(i) Given $t \in EdgeTypeID$, `-->{t}` *is an rpe.*

(ii) Given two rpes rpe_1 and rpe_2, $(rpe_1 rpe_2)$ is an rpe. [concatenation]
(iii) Given two rpes rpe_1 and rpe_2, $(rpe_1 \mid rpe_2)$ is an rpe. [choice]
(iv) Given an rpe rpe, (rpe^*) is an rpe. [closure]

According to subsection 3.3, a regular path expression *rpe* defines a language $L(rpe)$ over *EdgeTypeID*. Assuming the usual precedences (concatenation before choice before closure) unnecessary parentheses may be skipped.

This definition is simplified in the sense that only forward arrows `-->` are used which describe edges in their original direction. There are several other edge notations which may be used, as well:

`<--` describes an edge traversed in the opposite direction.
`<->` describes an edge traversed in any direction.
`<>--` describes an aggregation edge traversed from its aggregate's side.
`--<>` describes an aggregation edge traversed from its component's side.

Semantics of Path Expressions. The semantics of a regular path expression *rpe* is the set of all paths, whose derived edge type sequence conforms to the language defined by the *rpe*.

Path descriptions are used as abstract operators. Assuming that *rpe* is a regular path expression and v, w are vertices, there are several ways to apply *rpe*:

- `v rpe` is the set of vertices reachable from `v` according to `rpe`.
- `rpe w` is the set of vertices from which `w` is reachable according to `rpe`.
- `v rpe w` is the condition that `w` is reachable from `v` according to `rpe`.
- `path(v,rpe,w)` is a path from vertex `v` to vertex `w` if it exists.
- `pathSystem(v,rpe)` is a path system containing exactly one path for every vertex reachable from `v` according to `rpe`.
- `pathSystem(rpe,w)` is a path system containing exactly one path from every vertex from which `w` is reachable according to `rpe`.

All these applications can be evaluated by *search algorithms* on vertices which are explained in more detail in the next subsections.

Full Syntax. Besides the simplified definition above, there are several other notations which are also allowed in GReQL path expressions. They all can be handled as well inside the search algorithms to be described below.

(1) Restrictions to the vertex types of a path can be added by using an `&` sign in front of a type in braces, e.g. `&{MethodDeclaration}` states that only `MethodDeclaration`-vertices are allowed.
(2) For vertices and edges also boolean expressions are allowed instead of type restrictions, where the current element is denoted by `thisVertex` or `thisEdge` respectively.
(3) A specific edge can be attached to an edge symbol by embedding any expression that evaluates to an edge, as in `--e->` where `e` is a variable containing an edge. Similarly vertex variables like `v` can be used to denote a special vertex.

(4) Role names defined in the schema can be used instead of or additional to edge types. E.g. `<->{@member}` restricts the set of edges to those which are incident to a vertex with role `member`
(5) There are further operations on regular path expressions such as transitive closure (`rpe+`), exponentiation (`rpe^n`), option (`[rpe]`) and transposition (`rpe^T`).

Example. As an example, the query below denotes the set of all classes containing a method which calls a method of class `c` in a graph according to the schema of Figure 2.

```
1 c <--{IsClassBlockOf} <--{IsMemberOf} &{MethodDeclaration}
2   -->{IsDeclarationOfInvokedMethod} <--{IsMethodContainerOf}
3   -->{IsStatementOf}* -->{IsBodyOfMethod}
4   -->{IsMemberOf} -->{IsClassBlockOf} &{thisVertex <> c}
```

Here, all vertices reachable from a class c via paths according to the regular path expression are returned. These paths have the following structure: They lead from c to some member m of type `MethodDeclaration` (line 1). Starting from m, they find some call expression (line 2) where m is called. Then, the method $m1$ is determined that this call expression belongs to (line 3). Finally, the class of $m1$ is derived, and it is assured that this class is different from c.

5.2 Search Algorithms

Regular path expressions can be evaluated efficiently by *search algorithms* on the given graph. This holds for all features of regular path expressions described.

In the following, the algorithms needed for the simplified syntax are explained in detail. It should be obvious how these algorithms can be extended to handle the full syntax.

Search Algorithms. Search algorithms are *traversal algorithms*, i.e. they visit every graph element of (a part of) a graph exactly once. Search algorithms mark every vertex they visit to avoid double visits and use a collection object (usually called a *work list*) to control the spreading of this marking over the graph.

The set of marked vertices can be kept in a *set structure* according to the interface `Set`:

```
1  interface Set<E> {
2    /** @return true iff this set contains no elements */
3    boolean isEmpty ();
4
5    /** inserts the element x into this set */
6    void insert (E x);
7
8    /** @return true iff the element x is a member of this set */
9    boolean contains (E x);
10 }
```

The work list can be stored in any *list-like structure*, described by the interface `WorkList`:

```
1   interface WorkList<E> {
2     /** @return true iff this worklist contains no elements */
3     boolean isEmpty ();
4
5     /** inserts the element x into this worklist (possibly multiple times) */
6     void insert (E x);
7
8     /** returns (and deletes) an arbitrary element from the list */
9     E extractAny ();
10  }
```

Reachability. Given implementations for the interfaces `Set` and `WorkList`, a simple search algorithm can be given that visits all vertices in

$$reach_G(i) := \{v \in V \mid \exists\, C : Path \bullet \alpha(C) = i \wedge \omega(C) = v\}$$

i.e. the set of all vertices reachable from a given vertex i.

In the following pseudocode "visiting" a vertex v or an edge e is expressed by action points, which are noted as pseudo-comments like

`// process vertex v` or
`// process edge e`,

respectively. At these action points, potential visitor objects may be called to execute some action on the respective element:

```
1   Algorithm: SimpleSearch:
2
3   // vertex−oriented search starting at vertex i
4
5   Set<Vertex> m = new ...;
6   WorkList<Vertex> b = new ...;
7
8   void reach(Vertex i) {
9     m.insert(i);
10    // process vertex i
11    b.insert(i);
12    while (! b.isEmpty()) {
13      // inv: set(b) ⊆ m ⊆ i →*
14      v = b.extractAny();
15      for (Edge e: v.getAllOutEdges ()) {
16        // process edge e
17        w = e.getOmega();
18        if (! m.contains(w)) {
19          // process tree edge e
20          m.insert(w);
21          // process vertex w
22          b.insert(w);
23  } } } }
```

During the `while`-loop the work list b invariantly contains only marked vertices that are reachable from i. This is expressed by the invariant in line 13, where $set(b)$ denotes the set of all elements contained in b. Thus, all marked vertices are reachable. Conversely, any vertex reachable from i will eventually be included into b. Since insertion into m is only done for non-marked vertices and a vertex inserted into b is marked simultaneously, no vertex is inserted twice into b.

Every vertex is extracted at most once from b and the inner `for`-loop traverses its outgoing edges only. Thus, the body of the inner loop is executed at most once for each edge. Together this leads to a time complexity of $\mathcal{O}(max(n, m))$ for a graph with n vertices and m edges.

Reachability Tree. It is well-known that the spreading of the marking over the graph defines a spanning tree of the marked part. This tree is rooted in i and contains an incoming edge for every other vertex reachable from i.

Such a tree can be represented by a predecessor function

$$\texttt{parentEdge} : V \nrightarrow E,$$

which assigns its incoming edge to each non-root vertex. Such a tree is called *reachability tree* for $i \rightarrow^*$.

Partial functions like `parentEdge` on the vertices can be stored in a map-like data structure, according to the interface `VertexMarker` which stores at most one value at every vertex:

```
1  interface VertexMarker<E> {
2    /** stores the value x at the vertex v */
3    void setValue (Vertex v, E x);
4
5    /** @return the value stored at the vertex v */
6    E getValue (Vertex v);
7  }
```

Given such a vertex marker, the computation of `parentEdge` can be done in the algorithm `SimpleSearch` by refining the action point where the tree edges are processed. Here, e is the current edge and w is the newly marked vertex:

```
1  VertexMarker<Edge> parentEdge = new ...;
2
3  process tree edge e:
4  ~~~~~~~~~~~~~~~~
5    parentEdge.setValue(w,e);
```

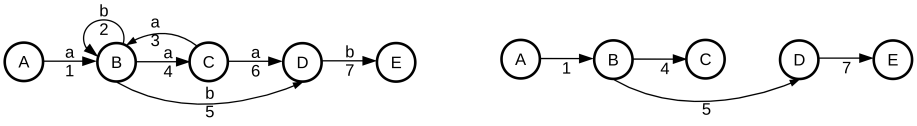
Paths. Given `parentEdge`, as computed by the algorithm `Simple Search`, a corresponding path $i \rightarrow^* v$ is implicitly given for every vertex v reachable from i. Such a path can be issued in reverse order by *backtracing* over the tree edges starting at v.

```
process path i →* v:
~~~~~~~~~~~~~~~~~~
  z := v;
  //process path vertex z;
  while (z != i) do {
    e := parentEdge.getValue (z);
    // process path edge e;
    z := e.alpha();
    // process path vertex z;
  }
```

Shortest Paths. The work list used in search algorithms can be implemented in various ways. It is well-known, that a queue-like implementation leads to a *breadth-first search (BFS)* approach, whereas a stack-like implementation implies a *depth-first search (DFS)* of the graph.

The breadth-first implementation of graph traversal is particularly interesting, since it implies that the path-tree is a *shortest path-tree*, i.e. all paths in the path tree have a minimum number of edges. Thus, GReQL uses breadth-first-search for the evaluation of regular path expressions.

Example. Figure 3(a) contains the sketch of a TGraph to demonstrate the effect of Algorithm `SimpleSearch` and the computation of `parentEdge`. Assume that the vertex set $\{A, B, C, D, E\}$ and the edge set {1,2,3,4,5,6,7} as well as the incidences are ordered according to their identifiers. The edge types $\{a, b\}$ are also shown.



(a) TGraph (b) BFS Tree

Fig. 3. Sample TGraph and its BFS tree

Using a breadth-first search starting in vertex A, the tree shown in Figure 3(b) is derived. It shows that all vertices are reachable from A. E.g. a path for vertex E can be derived by backtracing from E:

$< A, 1, B, 5, D, 7, E >$.

5.3 Automaton-Driven Search Algorithms

Algorithm `SimpleSearch` explores parts of the graph in order to find all vertices that are reachable from the start vertex i by traversing paths from i to v for every $v \in i \rightarrow^*$.

To explore paths whose edge type sequence conforms to a regular path expression *rpe* in a search algorithm, the traversal of edges must be pre-checked according to the derived edge type sequence language defined by the expression *rpe*. Only paths whose derived edge type sequence conforms to *rpe* are to be allowed.

As cited in subsection 3.3 there is an accepting deterministic finite automaton $dfa(rpe)$ for every regular path expression *rpe* with $L(dfa) = L(rpe)$. Using Thompson's and Myhill's algorithms such an automaton can be computed easily.

rpe-reachability. Let *dfa* be a DFA for the edge type sequence of *rpe*, let s be some state in S_{dfa}, and let G be a graph with a given vertex $i \in V_G$. Then

$reach_{G,dfa}(i, s) :=$

$\{v \in V \mid \exists\, C : Path \bullet \alpha(C) = i \wedge \omega(C) = v \wedge typeSeq(C) \in L(dfa, s)\}$

is the set of all vertices reachable from i by a path whose derived edge type sequence is accepted by the state s in S_{dfa}.

Then, the problem of finding all vertices v which are reachable by a path conforming to some regular path expression *rpe* reduces to the derivation of all vertices in some set $reach_{G,dfa}(i, s_t)$, where $dfa = A(rpe)$ and $s_t \in F_{dfa}$ is a terminal state.

The simple algorithmic approach described above in Algorithm `SimpleSearch` for reachability problems can easily be generalized to solve *rpe*-reachability. To achieve this, the finite automaton $dfa(rpe)$ for $L(rpe)$ is used to guide the traversal procedure.

Instead of a boolean marking of the vertices, now the vertices are marked with the states of *dfa*, i.e. a vertex set marker

`marking` : $Vertex \nrightarrow \mathbb{P}\ V_{dfa}$

is assumed. Here, a vertex v gets marked by all states s such that $v \in reach_{G,dfa}(i, s)$.

```
1  interface VertexSetMarker<E> {
2    /** inserts value x into the set at vertex v */
3    void addValue (Vertex v, E x);
4
5    /** @return true iff the value x is in the set at vertex v */
6    boolean hasValue (Vertex v, E x);
7  }
```

Using this generalized marker, which assigns sets of automaton states to the vertices of the graph, a vertex v gets marked by a state s if and only if $v \in reach_{G,dfa}(i, s)$, i.e. if there is a path from i to v whose derived type sequence is accepted by s.

```
1  Algorithm: AutomatonDrivenSearch:
2  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3  // vertex−oriented search starting at vertex i guided by the automaton a
4
5  VertexSetMarker<State> m = new ...;
6  WorkList<Vertex x State> b = new ...;
7  VertexMarker<Edge x State> parentEdge = new ...;
8
9  void reach(Vertex i, Automaton dfa);
10   s := dfa.getStartState ();
11   m.addValue(i, s);
12   // process vertex i in state s;
13   b.insert(i,s);
14   while (! b.isEmpty()) {
15     // set(b, s) ⊆ {v ∈ V | v.isMarked(s)} ⊆ reach_{G,dfa}(i, s)
16     (v, s) := b.extractAny();
17     for (Edge e: v.getAllOutEdges()) {
18       // process edge e
19       w := e.getOmega();
20       for (Edge t: dfa.getAllEnabledTransitions(s, e.getType());
21         s1 := t.getOmega();
22         if (! m.hasValue(w,s1)) {
23           parentEdge.setValue((w,s1), (e,s));
24           m.addValue(w,s1);
25           b.insert(w,s1);
26           if (dfa.isTerminal (s1))
27             // process vertex w in state s1
28 } } } } }
```

The correctness arguments for Algorithm `AutomatonDrivenSearch` can be reduced to those of Algorithm `SimpleSearch`. Assume that the automaton *dfa* consists of the states $V_{dfa} = \{s_0, ..., s_k\}$. Then, Algorithm `AutomatonDrivenSearch` corresponds to a search on a *layered graph* (Figure 5) with the vertex set $V_G \times V_{dfa}$ where an edge from (v, s) to $(w, s1)$ exists if and only if

- there is an edge $e = v \rightarrow w$ in G and
- there is an edge $s \rightarrow_t s_1$ in A, where $t = type(e)$.

Algorithm `AutomatonDrivenSearch` terminates and visits exactly those vertices that are reachable from i via paths conforming to *rpe* in the layered graph.

If k is the number of states and l is the number of transitions of the automaton *dfa*, the inner loop (lines 20-28) is executed $l \times m$ times and its `if`-statement is executes $k \times n$ times at most, leading to an overall time complexity of $\mathcal{O}(max(k \times n, l \times m)$. Since in practice k and l are small multiples of the size of the regular expression *rpe* (Subsection 3.3) the algorithm is practically feasible.

Using a breadth-first approach, the algorithm delivers shortest paths also in this case. But it should be noted, that these are paths in the layered graph. In the orginal graph, the paths themselves are not necessarily proper paths any more, i.e. vertices and edges may occur more than once on them.

Example. If the TGraph of Figure 3(a) is searched from A according to the GReQL path expression in `A (-->{a}-->{b})*-->{b}`, the automaton shown in Figure 4 can be used to drive the search. Here, s_0 is the start state, and s_2 is the (only) terminal state.

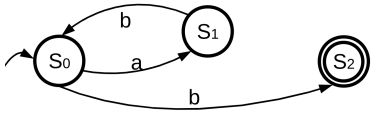
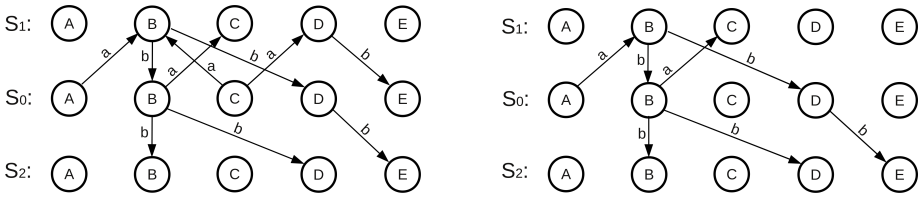


Fig. 4. DFA

Given this automaton, the resulting graph can be visualized using exactly one layer for each state (Figure 5(a)). Its breadth-first search tree is shown in Figure 5(b). Since s_2 is accepting, there are apparently three vertices in the result set $\{B, D, E\}$ and back tracing delivers three paths:

$< A, 1, B, 2, B, 2, B >$
$< A, 1, B, 2, B, 5, D >$
$< A, 1, B, 5, D, 7, E >$

All of these paths have minimum length, but only the third one is a proper path.



(a) Layered Graph (b) BFS Tree

Fig. 5. Search driven by automaton

Paths. The method to extract paths for each vertex found by Algorithm `Automaton-DrivenSearch` can also be generalized accordingly. Given a partial function

`parentEdge` : $V_G \times V_A \nrightarrow E \times V_A$

the paths can be enumerated, as well.

Assuming that s_t is a terminal state of A, and assuming that v is a vertex which is marked by s_t, i.e. $s_t \in m(v)$, a path can be computed by backward traversal in the layered graph:

```
1  process path i →* v:
2  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3    (z,s) := (v,s_t);
4    // process vertex z in state s;
5    while ((z,s) ≠ (i,s_0)) {
6      (e,s) = z.parentEdge(z,s);
7      // process edge e in state s;
8      z = e.this();
9      // process vertex z in state s
10   }
```

Since *parentEdge* contains the complete information about all paths starting in i, this function is used as a representation of path systems in `JValue`.

6 Applications of Querying

The TGraph approach was developed in the course of tool development projects. Kogge [ESU99] was a metaCASE tool, where TGraphs were used to realize an adaptable visual editor using a bootstrapping approach. Gupro [EKRW02] makes use of TGraphs as representations of software artefacts and supplies an integrated querying and browsing facility on code. Here, GReQL is the generic language used to extract information from source code. TGraphs together with GReQL are also used as the basic repository technology for artefact comparison and retrieval in ReDSeeDS [EBRS08].

Using a query language like GReQL, a lot of program analysis and program understanding tasks can be solved in a uniform manner. A query may deliver data for generic graph algorithms or can even replace a graph algorithm.

Examples for tasks that can be done using a query are

- complementing the graph by additional information that can be inferred, e.g., as a preprocessing step for other tasks,
- derivation of new views on the graph, e.g., derivation of the call graph, or
- computation of metrics.

In the following, we present some example GReQL queries to calculate some of this information on software represented as graphs according to the schema described in Figure 2 on page 4.

6.1 Edge Inference

The size and complexity of queries depends on the structure and completeness of the graphs and thus on the meta model and the parsers which are used for extracting the graphs from the source code.

Assume the parsers that extract the graph according to Figure 2 do not resolve all method invocations to their definitions but only the invocations of methods defined in the same class. Instead of computing the missing links in each query where they are needed, the work on the graph can be simplified

by querying this information once and *materializing* the result as edges in the graph.

GReQL itself is not meant to create edges, but it can be used to determine the respective set of vertex pairs which are to be connected. Then, the query is embedded into an algorithm that uses the query result and materializes the edges.

The query listed below finds the appropriate method declaration for every method invocation which is not yet properly linked to its declaration by the parser. Since GReQL is a declarative language, the query simply denotes which pairs are to be connected. (The GReQL optimizer assures an efficient evaluation of the query.)

```
1 from inv:V{MethodInvocation}, def:V{MethodDeclaration}
2 with isEmpty(inv <--{IsDeclarationOfInvokedMethod})
3  and theElement(inv <--{IsNameOfInvokedMethod}).name
4    = theElement(def <--{IsNameOfMethod}).name
5  and inv <--{IsMethodContainerOf}
6      ( <--{IsDeclarationOfInvokedMethod} <--{IsReturnTypeOf}
7      | <--{IsDeclarationOfAccessedVariable} <--{IsTypeOfVariable})
8      <--{IsTypeDefinitionOf}<--{IsClassBlockOf}<--{IsMemberOf} def
9 report inv, def end
```

The first and the last line state that pairs of `MethodInvocation` and `MethodDeclaration` vertices should be reported. The lines in between impose some constraints on the pairs to be reported by describing exactly the connection path between them. Line 2 excludes those invocations that are already linked to a declaration. Lines 3-4 force the names of the invoked method and the declaration to be equal. Method overloading is not considered here for reasons of brevity but can be included in the same way.

The most interesting part of the query is the path expression in lines 5-8. This path expression describes a path starting at the invocation and ending at the declaration. The first `IsMethodContainerOf` edge leads to the expression that returns the object on which the method is called. In the example `s.compareTo(data)` from page 3 the edge connects the call of `compareTo` to the variable access `s`. The following path alternative in lines 6-7 describes the path leading to the type of the object on which the method is called. The last part of the path in line 8 denotes the connection of this type with the method defined as a member of the main block of the class defining the type.

While the links are computed by the parser for invocations of methods defined in the same class, a proper linking of methods of other classes is possible only if the class whose method is called is known. For methods of objects which are returned by other method invocations or variable accesses, this class is obviously only known if the invocation or access is properly linked to the appropriate definition. This is a recursive problem which can either be solved in the query itself or by iterating the query in the embedding algorithm, which seems to be more elegant in this case.

6.2 Call Graph Computation

Similarly to the calculation of missing information that was not yet provided by the parser, it is also possible to calculate and materialize further information in the graph which enables a higher-level view. This is reverse engineering in the stronger sense, where more abstract information is derived from concrete data.

The query listed below computes the "call graph" of a given graph, i.e. it determines pairs of methods which call each other. This is done by looking at all method invocations contained in the body of a method. The query result can again be manifested as edges, in this case with edges of type `IsCalledByMethod`.

```
from caller, callee:V{MethodDeclaration}
with caller <--{IsBodyOfMethod} <--{IsStatementOf}*
            <--{IsDeclarationOfInvokedMethod} callee
report caller, callee end
```

6.3 Metrics Computation

Metrics constitute a quite natural application of querying. As an example, one of the metrics of the Chidamber&Kemerer metrics suite [CK91] is shown in the following: The CBO (coupling between object classes) metric assigns a natural number to each class, depicting the number of other classes it is coupled with. Here coupling means usage of variables or methods of the other class. Given a graph preprocessed as shown above, this metric can be calculated directly.

```
from c:V{ClassDefinition}
report
  c.name as "Class",
  count(
    c<--{IsClassBlockOf} <--{IsMemberOf}
    (<--{IsCalledByMethod}  |
     <--{IsBodyOfMethod,IsVariableCreationOf} <--{IsStatementOf}*
     <--{IsDeclarationOfAccessedVariable}-->{IsVariableCreationOf})
     -->{IsMemberOf} -->{IsClassBlockOf} &{thisVertex<>c}
  ) as "CBO"
end
```

For every class, the set of elements is computed that are reachable by a path that leads to an invocation of a method or an access of a variable of a different class. The size of this set is counted by the GReQL-function `count` and reported as the CBO for this class. The result of this query will be a table with two columns, named "Class" and "CBO" containing the class name and the number of classes this class is coupled with.

7 Related work

Graphs as repository structures in software engineering tools and querying of these graphs is an enabling technology [KW99] in software reengineering. The definition of GXL [HSSW06] (which is inspired by TGraphs) as an interchange format for reengineering data gives further evidence that graphs are an appropriate abstraction used by many reengineering tools.

Graph Repositories. There are libraries for graphs which keep them in memory and provide a set of predefined algorithms on them, e.g., *LEDA* (The Library of Efficient Data Types and Algorithms) [MNU97] and the *Boost Graph Library* (BGL) [SLL01]. The latter provides different implementation variants for graph such as edge lists or adjacency lists and matrices.

GRAS [KSW95] is a graph repository developed in the IPSEN project since 1985. A GRAS database is a graph pool that may contain several directed and typed graphs. (GRAS graphs have only vertex attributes and no edge ordering.) The structure of the graphs can be meta modeled by graph schemas and composition of graphs to hierarchical graphs is possible. GRAS supports direct main memory storage as well as storage in a relational database. *DRAGOS* (Database Repository for Applications using Graph Oriented Storage) [Böh06], the successor of GRAS, is inspired by the graph exchange language GXL and overcomes the restrictions on edge attributes of GRAS. Additionally, DRAGOS allows the usage of directed hyper edges which may run between vertices as well as edges or endpoints of edges. Similarly to GRAS, DRAGOS also supports nesting of graphs.

JGraLab keeps the repository in memory and provides the modeling power of full TGraphs, which are kept in a data structure designed to support graph traversal efficiently. Edges are first class objects and may be traversed in any direction without additional costs.

Graph Query Languages. Apart from the XML query languages *XQuery* [B$^+$07b] and *XPath* [B$^+$07a] and the *SPARQL* RDF Query Language [PS08], there are also some languages that work on graph models directly. One such language is *Gram* [AS92] which includes walks and hyperwalks as a concept similar to paths and path systems. *GOQL* is an extension of the Object Query Language OQL used to query object-oriented databases enriched with constructs to query typed graphs. GOQL bears some syntactical analogy to GReQL as its queries are usually select-from-where statements and it allows to formulate simple path queries including constraints on the elements in a path. All these languages follow the idea of querying depicted in section 4, i.e. given a graph and a query, a value possibly containing graph elements is computed as the query result, which may also contain graph elements.

Languages like *GraphQL* [HS08] follow a different paradigm. Here, not only the data, but also the query and the query result are graphs. Then, query evaluation is a matching of graph patterns combined with graph rewrite rules, either transforming the given graph or creating a new one. Also *G+* [MW89] and its

successor *GraphLog* [CM90] represent the query and the result as a graph and support regular path expressions and an evaluation by an automaton-driven search similar to the one used in GReQL. These kinds of languages lead to graph transformation languages in general, like PROGRES [RW08], which may be interpreted as query languages in wider sense.

Compared to these languages, GReQL supports the most powerful form of regular path expressions, and only GReQL computes also paths and path systems as first class values.

Query languages for software re-engineering. Besides graph query languages, several other kinds of languages based on predicate logic or relational calculus are established in software re-engineering. [AHR09] presents a detailed comparison of re-engineering languages based on their features. There are two main differences between all those languages and GReQL. The first refers to the representation of data, since edges as first class objects may directly represent occurrences of vertices in different places without artificial extra-nodes. The second is related to evaluation, which is search-based and does not compute and materialize relations, but explores only that part of the graph that is needed for query answering.

CrocoPat [Bey06,BNL05] is a relational programming language based on first-order predicate calculus. It uses Binary Decision Diagrams (BDDs) for internal storage of relations. Like many other tools in software reengineering, CrocoPat uses the Rigi Standard Format (RSF) to store relations in files. CrocoPat programs are written in the *RML* (Relation Manipulation Language) and consist of n-ary relational expressions. RML is an imperative language whose statements are executed sequentially, embedded in control structures such as `IF`, `FOR` and `WHILE`.

Supplying different ways to define n-ary relations and to combine existing relations to new ones, CrocoPat provides powerful means of data manipulation and retrieval. Regular path expressions can be simulated by logical operators to combine and by existential quantifiers to concatenate relations. E.g., the path expression

```
caller <--{IsBodyOfMethod} <--{IsStatementOf}*
       <--{IsDeclarationOfInvokedMethod} callee
```

used in Section 6.2 can be translated to the CrocoPat expression

```
calls(caller,callee) :=
       EX(body, IsBodyOfMethod(body,caller)
       & EX(statement, TC(IsStatementOf(statement,body))
       & IsDeclarationOfInvokedMethod(callee,statement))).
```

A language based on Tarski's relational calculus is *Grok* [Hol08]. Grok is an untyped language, where all basic elements are represented as strings, but logical and mathematical operations can be applied to these elements. Sets and binary relations, which are just sets of tuples, are supported by Grok. Grok's operators generally apply across entire relations and not just to single entities.

The simulation of regular path expressions is possible by the concatenation of relations using the $\circ$ operator. As an example,

```
(inv IsBodyOfMethod) o (inv(IsStatementOf)*)
      o (inv IsDeclarationOfInvokedMethod)
```

represents the path expression used above. The result of applying this Grok operation is again a relation, which is stored to main memory to allow a fast access.

RScript [Kli08] is statically typed in contrast to Grok. RScript reflects its primary application domain, the software analysis, in its features and data types. As an example, a data type `location` is provided, which represents a source locations as a combination of a filename and the position in the file. Besides the basic types `boolean`, `integer`, `string`, RScript provides the composite types tuples, sets and relations as well as user-defined types. Sets as well as relations can be nested and also a light version of n-ary relation is also supported in RScript. Similarily to Grok and CrocoPat, the combination of relations can be used to simulate regular path expressions.

All these languages look at graphs as sets of vertices and relations between vertices whereas GReQL has a focus on paths and traversals of graphs.

Comparison. While e.g. Alves et al. [AHR09] have compared the main features of re-engineering query languages including the ones described above, there is no comparison of their performance up to now. Below, such a comparison is done exemplarily for CrocoPat and Grok, which seem to be the best established ones. and whose interpreters are publicly available. Two of the queries depicted in section 6, namely the calculation of the call-relation (Section 6.2) between methods and the CBO metrics (Section 6.3) are used for comparison. They were applied to the ASGs of four software systems, whose source code is freely available. Two small systems, the parser generator AntLR[2] and the test-environment JUnit[3] were used as well as two bigger systems, the Build-Tool Apache Ant[4] and the TGraph library JGraLab itself.

The abstract syntax graphs of the systems were extracted from the source code using a fact-extractor from Java to TGraphs, which makes use of edge attributes and ordering. The TGraphs were converted to the Rigi Standard Format (RSF), which can be imported by CrocoPat and Grok. Neither Grok nor CrocoPat allow for attributed relationships but require artificial relation-objects to represent such attributes. To keep the graphs and queries simple, we decided to use simple links instead of relation-objects and to accept, that the attributes of the edges were lost. While the attributes are not used in the queries below, they may be necessary for more complex analyses.

The evaluation times of both queries are shown in the table below for all three languages and all ASGs. Note, that CrocoPat and Grok are interpreters whose interpretation time is included in the overall result, whereas GReQL queries are parsed and optimized before evaluation. Since GReQL queries can be precom-

[2] www.antlr.org
[3] www.junit.org
[4] ant.apache.org

piled into a query library and reused for different graphs, the net evaluation time is given, as well.

The table shows that GReQL performs quite well in comparison to CrocoPat and Grok for the examples used.

	AntLR 90 classes 73k elements	**JUnit** 110 classes 42k elements	**Apache Ant** 1400 classes 1.1m elements	**JGraLab** 700 classes 1.7m elements
Calls-Query				
CrocoPat	1.9s	1.9s	62s	70s
Grok	0.3s	0.1s	7.0s	6.7s
GReQL	0.9s (0.13s eval)	0.8s (0.08s eval)	3.7s (2.7s eval)	3.5s (2.4s eval)
CBO-Query				
CrocoPat	2s	1s	2m 40s	1m 30s
Grok	1.0s	0.5s	9.6s	10.1s
GReQL	1.1s (0.16s eval)	1.1s (0.15s eval)	6.7s (5.8s eval)	3.8s (2.9s eval)

8 Conclusion

This paper showed how knowledge from graph algorithms can be used to construct efficient software engineering tools. It presented GReQL as an efficient and convenient graph query language. The efficient evaluation of regular path expressions in GReQL by search algorithms was explicated in detail.

GReQL can be used to query, enrich, abstract or analyze the graph representation of software engineering artifact. A few example applications in reverse engineering were shown. GReQL querying is also an enabling technology for software engineering tools in general since many kinds of information can be easily be extracted from graph-based models using queries [KW99], including information needed by the tool itself.

The usage of GReQL as a key technology in the reengineering tool GUPRO is presented in more detail in [EKRW02]. GUPRO is a generic tool that supports schema dependent browsing and querying of source code, using GReQL as its query language.

GReQL has been extended to include context-free path descriptions by Steffens [Ste08]. Currently ongoing work aims at the extension of GReQL and the TGraph approach to more general distributed and hierarchical hyper-TGraphs (DHHTGraphs).

References

[AHR09] T. L. Alves, J. Hage, and P. Rademaker. Comparative study of code query technologies. Online PDF, wiki.di.uminho.pt/twiki/pub/Personal/Tiago/Publications/Alves09b-draft.pdf, 2009. 04.06.2010.

[AS92] B. Amann and M. Scholl. Gram: A graph data model and query language. In *European Conference on Hypertext*, 1992.

[ASU87] A. Aho, R. Sethi, and J. Ullmann. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1987.

[B+07a] A. Berglund et al., editors. *XML Path Language (XPath) 2.0, W3C Recommendation*. January 2007.

[B+07b] S. Boag et al., editors. *XQuery 1.0: An XML Query Language, W3C Recommendation*. January 2007.

[Bey06] Dirk Beyer. Relational programming with crocopat. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 807–810, New York, NY, USA, 2006. ACM.

[Bil08] D. Bildhauer. *Entwurf und Implementation eines Auswerters für die TGraphanfragesprache GReQL 2*. VDM Verlag, 2008.

[BNL05] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Softw. Eng.*, 31(2):137–149, 2005.

[Böh06] B. Böhlen. *Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge*. Shaker Verlag, Aachen, 12 2006.

[CK91] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM.

[CM90] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 404–416, New York, NY, USA, 1990. ACM.

[Ebe87] J. Ebert. A Versatile Data Structure For Edge-Oriented Graph Algorithms. *Communications ACM*, 30(6):513–519, 6 1987.

[EBRS08] J. Ebert, D. Bildhauer, V. Riediger, and H. Schwarz. Using the TGraph Approach for Model Fact Repositories. In *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008)*, 5 2008.

[EKRW02] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science, http://www.elsevier.nl/locate/entcs/volume72.html*, 72(2), 2002.

[ERW08] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, the TGraph Approach. In R. Gimnich et al., editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, 2008. GI.

[ESU99] J. Ebert, R. Süttenbach, and I. Uhe. JKogge: a Component-Based Approach for Tools in the Internet. In *Proceedings STJA '99*, Erfurt, 1999.

[Hol08] Richard C. Holt. Wcre 1998 most influential paper: Grokking software architecture. In *WCRE*, pages 5–14, 2008.

[Hor09] T. Horn. *Ein Optimierer für GReQL2*. GRIN Verlag GmbH, 2009.

[HS08] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418, New York, NY, USA, 2008. ACM.

[HSSW06] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: a graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 4 2006.

[HU79] J. E. Hopcroft and J. D. Ullmann. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1079.

[Kli08] Paul Klint. Using Rscript for software analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC)*, 2008.

[KSW95] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph oriented (software) engineering database system. *Information Systems*, 20(1):21–51, 1995.

[KW99] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In C. Verhoef and P. Nesi, editors, *Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering*, pages 42–50, Los Alamitos, 1999. IEEE Computer Society.

[Mar06] K. Marchewka. GReQL 2. Master's thesis, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[MNU97] K. Mehlhorn, S. Näher, and C. Uhrig. The leda platform of combinatorial and geometric computing. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 7–16, London, UK, 1997. Springer-Verlag.

[MW89] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1989.

[Myh57] J. R. Myhill. Finite automata and the representation of events. Technical Report 57-624, Wright Patterson AFB, Ohio, 1957.

[Nag80] M. Nagl. An incremental compiler as component of a system for software generation. In *Programmiersprachen und Programmentwicklung, 6. Fachtagung des Fachausschusses Programmiersprachen der GI*, pages 29–44, London, UK, 1980. Springer-Verlag.

[Obj06] Object Management Group. *Meta Object Facility (MOF) Core Specification, OMG Availiable Specification, Version 2.0*, 2006.

[PS08] E. Prud'hommeaux and A. Seaborne, editors. *SPARQL Query Language for RDF, W3C Recommendation.* January 2008.

[RW08] U. Ranger and E. Weinell. The graph rewriting language and environment PROGRES. *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany*, pages 575–576, 2008.

[SLL01] J. G. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Professional, 2001.

[Ste08] T. Steffens. *Kontextfreie Suche auf Graphen.* VDM Verlag, 2008.

[Tho68] K. Thompson. Regular expression search algorithms. *Communications of the ACM*, (11):419–422, 6 1968.