Graph Technology and Semantic Web in Reverse Engineering - A Comparison -

Gerd Gröner
ISWeb - Information Systems and Semantic Web
University of Koblenz-Landau
http://isweb.uni-koblenz.de
groener@uni-koblenz.de

Steffen Staab
ISWeb - Information Systems and Semantic Web
University of Koblenz-Landau
http://isweb.uni-koblenz.de
staab@uni-koblenz.de

Andreas Winter
Institute for Computer Science
Johannes-Gutenberg-University Mainz
http://www.gupro.de/winter
winter@uni-mainz.de

Abstract

Reverse engineering tools are mostly based on analyzing code repositories. Various technological spaces for realizing these repositories including appropriate analysis techniques exist. Graph technology and semantic web based technologies provide elaborated and sufficient means to analyze software structures. This paper elaborates differences and similarities of both technological spaces by comparing the GUPRO/GReQL program comprehension framework with OWL/SPARQL based code analysis.

1 Introduction

Reverse engineering aims at analyzing software systems to identify components and their interrelations to provide better understanding of software systems under development and maintenance. Tools used to support program understanding usually follow the *Extract-Abstract-View-Metaphor* [34]. Artefacts of a software system are extracted and identified from the source code and represented in a repository. The repository reflects an abstract representation of the software systems according the maintainers needs. If the repositories structure is made explicit [22] it is defined by conceptual modeling techniques.

There exit different realizations of a repository. In general the repository is organized in net-like structures like e.g. graphs (cf. Bauhaus [31], GUPRO [15], Rigi [35]) relations (cf. (RPA [29], SWAG Kit [19], Crocopat [9]), or logic oriented data representations (cf. DATA_Tool [11], CodeQuest [18]). According to its technological space [25] each representation is associated with appropriate analyzing technologies.

This paper aims at comparing approaches from metamodel-based graph analysis and semantic web technologies to be applied in program comprehension. As representatives for these spaces, we view at the GUPRO workbench for program comprehension and at code representations in Web Ontology Language (OWL) together with SPARQL query language.

GUPRO [15] is a graph-based approach for analyzing large and complex software systems originated from the metamodel-based software engineering. It uses a graph-based repository that realizes highly optimized graph analyzing and traversing algorithms. The software analysis is based on GReQL querying [24].

The Web Ontology Language (OWL) supports conceptual modeling based on the power and expressivity of description logics. Here, source code is described in an ontology. Based on the semantics of description logics reasoners provide powerful analyzing services and query languages like SPARQL [30] are used for interrogating ontologies.

GUPRO/GReQL and OWL/SPARQL are applied to parts of the GEOS software system [1], which is a large banking system used for stock trading transactions. Lange et al. [26] already used GEOS for comparing graph-based analyses with a database. This case study already outlined the good performance of graph-based systems for standard reverse engineering applications. Here, we apply the same analysis tasks for comparing graph-based analysis with semantic web analysis.

The remainder of this paper is organized as follows. Section 2 describes the two modeling approaches in-

cluding their tools and infrastructures. In section 3 the conceptual model reflecting the objective of analyzing GEOS system is specified. This section also introduces the required fact extraction for GUPRO and OWL. A comparative analysis of GEOS with GUPRO/GReQL and OWL/SPARQL is presented in section 4. Section 5 summarizes the differences and similarities of graph-based and web-based analysis and concludes the paper.

2 Program Analysis Techniques

This section outlines two different approaches for modeling repositories in order to provide analysis technologies that are used in reverse engineering for the GEOS system, a large and complex banking system.

2.1 GUPRO and GReQL

GUPRO (Generic Understanding for PROgrams) [16, 32], is a system for analyzing and visualizing software systems and documents. GUPRO uses a graph-based repository that implements sophisticated graph traversal algorithms. In GUPRO all informations about a software system are stored and managed in a repository.

The repository is organized as TGraphs [14] which are attributed, directed, ordered, and typed graphs. In an attributed and typed graph the vertices and edges may have assigned attribute and type values. In an ordered graph there is an order for all vertices and edges.

GUPRO implements efficient graph algorithms for querying and extracting informations from a graph. The objects of the software system are nodes of the TGraph and all relations are modeled as edges between the corresponding nodes. Objects of a software system are not only classes but also attributes, methods or database tables. Therefore nodes are all entities in the software system.

GUPRO provide different tools for the data extraction. For the extraction in [26] ANAL/SoftSpec is used. The abstract data are visualized for the user by different visualization tools and different output formats.

The conceptual model is based on the EER/GRAL [17] modeling approach. EER/GRAL is an extended entity relationship (EER) diagram augmented with a constraint language GRAL. The constraint language is used to specify integrity conditions and *path expressions* for TGraphs. A path expression describes a path in a graph and also restrictions on a path.

```
from  tab: V{Table}, col: V{Column}
with  tab.name = 'Article' AND
      col -->{isColumnOf}  tab
report col.name , col.type
end
```

Figure 1. A GReQL Query example.

2.1.1 Querying the GUPRO Repository

For analyzing TGraphs a special graph-based query language GReQL (Gragh Repository Query Language) [23] is used. GReQL is a declarative language for extracting information from the repository but doesn't change the data. A GReQL query consists of the three clauses `from`, `with` and `report`. The `from` clause declares variables for the concerning elements (nodes and edges) in the graph with the corresponding domain of each variable.

In the `with` clause are predicates declared which have to be fulfilled from the variables. These predicates are powerful graph oriented expressions like path expressions. The `report` clause determines the result structure of the query.

A GReQL query is evaluated in two steps. The first step is a test if the predicate from the `with` clause is fulfilled. In the second step the expression is computed as described in the `report` clause. Objects and relations between objects are both first order concepts in GReQL. They are attributed and typed.

An example of a GReQL query is displayed in Figure 1. The two variables *tab* and *col* are declared in the `from` clause. The types of these variables are `Table` and `Column` which are both subtypes of `Vertex`. The vertex type is referenced with `V`. There are two predicates in the `with` clause. The first predicate is the condition that the name of the table *tab* is 'Article' and the second predicate is the path expression `col -->icColumnOf tab`. This expression describes that the two vertices *col* and *tab* are in a `isColumnOf` relation. This relation is a subtype of the `edge` relation. The structure of the result set are name and type tuples of the corresponding column. Name and type are attributes of the vertex subtype `edge`.

This example also demonstrates the kind of connection which are in the repository between the vertices `Table` and `Column` that is a representation of a database and a database column.

For a better program understanding GUPRO provides facilities for source code visualization [16]. The results of a GReQL query is represented in nested tables and additionally the corresponding source code is also shown beside the tables. It is a link between the (concrete) source code and the (abstract) query result.

2.1.2 Tools and Infrastructure

GUPRO with GReQL is a stand-alone system including tools for source code extraction, repository management, querying and visualization. GreQL is provided by various interfaces [15].

2.2 The OWL Model

The Web Ontology Language (OWL) [13] is the W3C standard ontology language for the Semantic Web. OWL is an expressive language for describing ontologies. An ontology is a model of a certain domain which describes classes or concepts, relations between classes, instances or individuals and a number of axioms. An ontology consists of two description formalisms: the terminological knowledge (TBox) and the assertional knowledge (ABox). The TBox describes concepts and relations whereas the ABox formalizes facts, i.e. properties of individuals and instances.

There are two kinds of relations or properties. An *object property* is a relation between objects, i.e. the domain and range are OWL classes. A *datatype property* is a relation between an object and a datatype, e.g. a datatype property `age` has a range of positive integer values. Datatypes are all XML schema datatypes [5]. Axioms are assertions about classes, class relationship and properties.

Ontologies are used for knowledge representation as a knowledge base in information systems. A requirement for ontologies and the use of ontologies is the machine processable data representation that leads to a kind of machine understandability of the data. The main tasks that are performed on ontologies are reasoning and querying. Reasoning is used for classification, consistency checks, class subsumption and instance checking. Querying ontologies is a part of information retrieval.

OWL has a well defined syntax and semantics. The underlying logical formalism of OWL is Description Logics (DL) [8].

The OWL language family is divided into three language species with different expressiveness. There is a tradeoff in using one of the OWL sublanguages between expressivity and computational properties.

- OWL Full is the most powerful and most expressive OWL language. OWL Full is the only OWL language which is fully upward compatible with RDF and RDFS [10].
- OWL DL is in general computational efficient for reasoning tasks, except from a high worst case complexity. OWL DL covers only a subset of the OWL-Full language. OWL DL is a variant of the DL-language $\mathcal{SHOIN}(\mathcal{D})$. The language restrictions are in favor of scalable reasoning services which are based on the DL reasoning techniques.
- OWL Lite is the easiest OWL-language and provides the best computational efficiency. The language is decidable for reasoning problems. It is a subset of OWL DL with certain language restrictions like no disjunction, no enumerations and only cardinality restriction with 0 and 1. OWL Lite is a variant of the DL-language $\mathcal{SHIF}(\mathcal{D})$.

OWL DL and OWL Lite profit from the well-defined semantics and the reasoning technologies adapted from DL. Therefore in this paper only OWL DL and OWL Lite is considered. One important extension of OWL DL compared to OWL-Lite is the use of *nominals* i.e. individual names in class descriptions.

Further restrictions are *explicit typing* [7] i.e. all resources have to be explicitly stated. There are no cardinality restrictions on transitive properties [7].

Since OWL was designed for specifying ontologies for the Semantic Web, there are some assumptions that are common for the Semantic Web use. One important assumption is the open-world assumption. If a statement is missing (unknown) it is not possible to conclude that it is false.

2.2.1 Querying the OWL Model

SPARQL [30] is a query language for ontologies, originally designed for RDF. A SPARQL query contains a set of triple patterns called basic graph patterns. These triples correspond to the RDF triple notation which are triples of the kind `subject`, `predicate` and `object`. These triples are in the `WHERE`-part of the query. The `SELECT`-part contains query variables which appear also in the triples of the `WHERE`-part. The patterns are matched against the RDF triples (RDF graph) and the query result is a set (solution sequence) in which every element is a data element from the RDF graph that matched to a variable of the pattern.

Figure 2 demonstrates a simple SPARQL query. In the `SELECT` clause of the query are two variables ?*name* and ?*type* defined. Variable identifiers always start with ?. In the `WHERE` clause are four graph pattern. The variables ?*col* and ?*table* are not in the result set. The first and second terms define the type of the variables ?*table* and ?*col*. The `isColumnOf` relation is expressed with the third pattern. The result are the name and type of all columns from all tables.

For constructing more complex queries there are some algebraic operations like the `FILTER` operation

```
SELECT   ?name ?type
WHERE  { ?table rdf:type   Table .
         ?col   rdf:type   Column .
         ?col   isColumnOf ?table .
         ?col   hasName    ?name .}
```

Figure 2. A SPARQL query example.

which is used to specify further constraints on the result set, the `UNION` operator and the `OPTIONAL` operator for left join operations.

Some reasoners support the SPARQL syntax for query answering. Therefore the SPARQL syntax is also used for querying OWL DL ontologies. All tools that are used in this evaluation support SPARQL.

2.2.2 Tools and Infrastructures

A comfortable modeling tool for ontologies is Protege [4], a free ontology framework. Protege provides support for creating and visualizing ontologies in various formats. The framework consists of two main parts: the Protege-Frames editor for building frame based ontologies and the Protege-OWL-editor for working with OWL ontologies. The OWL-editor enables tasks like loading and saving OWL (and RDF) ontologies, visualizing classes, individuals, object and data properties, and it provides reasoning capabilities such as consistency checking and classification. For this service Protege has an interface to enable a connection to reasoners like Pellet [33], Racer [6] or KAON2 [3] via a port connection.

In this case study the KAON2 infrastructure is used for querying and reasoning. KAON2 is a free java implementation. It is capable to manipulate OWL DL ontologies. For reasoning and querying it supports the DL sublanguage $\mathcal{SHIQ}(\mathcal{D})$, whereas querying is intern reduced to a reasoning task. The power of KAON2 compared to other reasoners is in reasoning (or querying) over large ABoxes. As described in [28] the field of deductive databases extremely studied reasoning over large data sets. KAON2 exploits experiences of this research field by implementing an reasoning algorithm that reduces the $\mathcal{SHIQ}(\mathcal{D})$ knowledge base to an disjunctive datalog program [20].

As a first step in KAON2 before reasoning tasks can be applied the generation of the disjunctive datalog program is performed. The rule set mainly depends on the TBox. The main component of KAON2 is the reasoning engine. This component consists of a theorem prover for the transformation step. The other component is the *disjunctive datalog engine* which works on the rule set.

For this case study KAON2 is well suited. Since the TBox is manageable and not complex whereas the ABox is large with about 9,500 individuals. For this ontology KAON2 provides a scalable querying service. The query engine supports the syntax of the SPARQL query language.

These additional services are also useful in the field of reverse engineering. Some standard reasoning services are:

- The *classification* of concepts computes all subclass relations of the TBox. The result is a complete class hierarchy. For reverse engineering a classification of concepts gives information about the subclass relations of all involved concepts.
- Reasoning provides the possibility of testing the *satisfiability* of a concept. A concept is satisfiable if every individual (instance) of this concepts is consistent with the ontology.
- *Checking* the ontology for *consistency* is another reasoning task. A consistent ontology doesn't contain contradictions. This property is useful for creating the conceptual model in order to guarantee consistency.

3 Conceptual Modeling

A conceptual model contributes to the repository in three different ways. First of all the model describes the structure of the repository, i.e. the objects, the object properties and the relationship between the objects. The extraction and transformation process is based on the model structure. Source code is parsed concerning to the model. Finally the information extraction, e.g. the kind of queries performed on the repository depends on the conceptual model.

3.1 The GEOS System

GEOS (Global Entity Online System) [1, 2] is an integrated online system for stock trading and derivate activity transactions in realtime. GEOS Nostro is an optional GEOS component that is specialized for automated balancing and supports different national and international accounting standards. In this reverse engineering analysis the GEOS Nostro component is used.

The GEOS system consists of more than 1,600 components, 3,000 modules, 2,200 classes, 30,000 interfaces , 34,000 functions, 290,000 function calls and 895,110 data references. The original system was built of more than 6,279 source files and about 2,364,652 lines of code [26].

3.2 The Conceptual Model for GEOS

Maintaining GEOS requires knowledge on programming constructs, databases and their interdependencies. Thus an abstraction of GEOS has to provide information on software modules, associated database tables, methods and attributes.

A **conceptual model** defines this structure for analyzing the GEOS Nostro. It specifies the facts that are extracted from the source code. The appropriate conceptual model is given in Figure 3.

The system is divided into several *components*. Components are collections of logically related modules. Such a collection provides interfaces to other modules. A *module* is a source file and is in an *include* relation with other modules. An *Uri* is an identifier with a name attribute. It describes exactly one *component*. A *class* is a special kind of module for object oriented languages. *Modules* and *Classes* consist of *methods* and *attributes*. The *tables* refer to the database tables in the GEOS system. The `uses` relation indicates the database access. The reflexive, transitive and antisymmetric relation *isSubclassOf* expresses the class dependencies and the class hierarchy. The same property is respectively expressed by the *include* relation between the *module* components. The dependencies and hierarchies of method calls are described with the *calls* relationship.

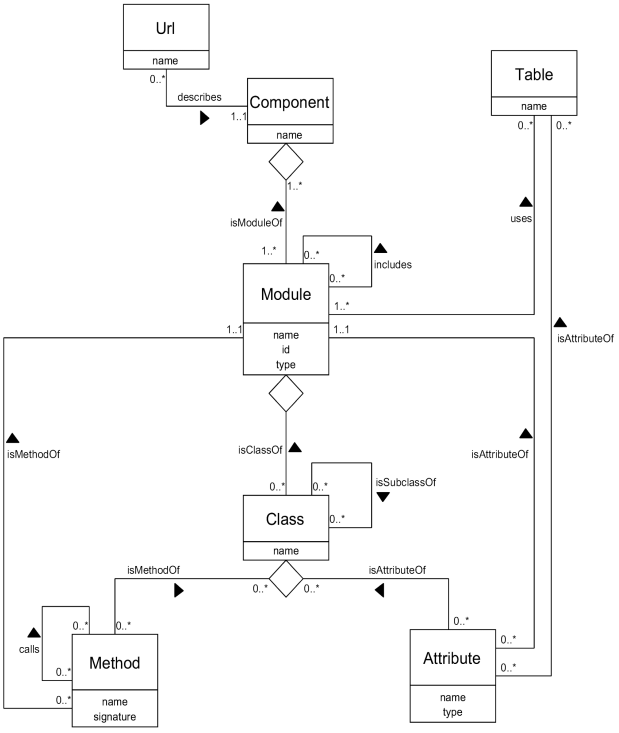

Figure 3. The Conceptual Model of the GEOS System.

3.3 Repository Based Reverse Engineering

Using ANAL/SoftSpec facts from the GEOS source code were extracted according the conceptual model in Figure 3. and stored in a relational database. To provide analysis with GUPRO, filters were created, to translate the DB content into TGraphs [26]. Further filters were based on TGraphs to create an OWL ontology representation of GEOS.

For reverse engineering tasks information is extracted from the repository Querying repositories in software reengineering is a powerful mechanism for extraction system information instead of searching in large documentations and diagrams [24].

OWL DL and OWL Lite realize a conceptual modeling approach. In conceptual modeling there is a strict separation of the *conceptual* and the *data* model [27]. The conceptual model describes the structure of the considered model that is comparable to a database schema. This includes all definitions of concepts and relations (rolls) between concepts. In an ontology this part of the model is the TBox.

On the other hand the data model consists of all individuals belonging to the concepts and the relations between this individuals. It is similar to an database instance and is the ABox of an ontology. In DL a valid instance is called a model.

A property in OWL is considered as a first class object and not just as an aspect of some classes. Therefore the ontology may contain assertions directly about a property of individuals. In GUPRO the edges which corresponds to object properties in OWL are first class objects in the same way. Assertions are expressed in GUPRO with type and attribute expressions.

The conceptual modeling with OWL DL is somewhat different to the UML-based modeling approach like in GUPRO. Some critical and perhaps not intuitive aspects are cardinality restrictions for the number of individuals that can conclude equivalence or value restrictions which implicitly conclude class membership [12]. In OWL DL cardinality restrictions can infer existence and equivalence of individuals.

The complexity of the DL ontology is $\mathcal{ALUHINC}(\mathcal{D})$. The $\mathcal{AL}$ is the DL base lan-

guage. The $\mathcal{U}$ allows union of concepts, the $\mathcal{H}$ is for describing role hierarchies such as subproperties (`rdfs:subPropertyOf`), the $\mathcal{I}$ indicates the inverse property and the $\mathcal{N}$ allows expression simple cardinality restrictions like `owl:MaxCardinality`. The identifier $\mathcal{C}$ allows complex concept negations.

3.4 Modeltransformation to OWL

Analyzing the GEOS system with SPARQL is based on the appropriate OWL representation of the facts already given in the GEOS TGraph. The conceptual model determines the structure of the TBox. The ABox contains the data of the TGraph.

The transformation from the TGraph to OWL is straightforward. All nodes are described as OWL classes. A node is defined as an OWL class and all other components of the system like modules or methods are subclasses of the node class. The definition of the class `Module` as a subclass of `node` is presented below. The attributes of this class are defined as datatype properties.

```
<owl:Class rdf:ID="Module">
    <rdfs:subClassOf rdf:resource="#node"/>
</owl:Class>
```

All edges are modeled as `ObjectProperty`. There is an `ObjectProperty` for edge and all other relations are subproperties of edge. The following listing demonstrates the definition of the `ObjectProperty includes`.

```
<owl:ObjectProperty rdf:ID="includes">
   <rdfs:subPropertyOf rdf:resource="#edge"/>
   <rdfs:domain rdf:resource="#node"/>
   <rdfs:range rdf:resource="#node"/>
</owl:ObjectProperty>
```

The class hierarchy, the call relation and the include relation are reflexive and transitive relations. There is a difference in modeling and querying the transitivity property of roles. The `ObjectProperty` role is per default not transitive, but it is possible to define a role as a transitive role. This is a difference compared to GUPRO, where it is not necessary to specify if a relation is transitive or not.

Therefore it is necessary in the transformation to decide for each role (ObjectProperty) whether it is transitive or not. These are the following relations: the `includes` role between modules, the `is SubClassOf` role between GEOSClasses and the `calls` role between methods. Reflexive roles are considered in the same way.

In a GReQL query it is specified in the path expression whether a direct connection or a transitive connection is focused. This is expressed in GReQL with $--{>}$ for the direct connection and with $--{>}*$ for a reflexive and transitive path. In OWL it is necessary to specify the edge (ObjectProperty) as a TransitiveProperty in the ontology. There is no way to define this in a plain SPARQL query.

For the performance evaluation the direct (non-transitive) queries are applied to an ontology without transitive property. For transitive queries an ontology with transitive properties (`TransitiveProperty`) is used. The queries are the same for direct and transitive edge connections but the ontologies are different.

A part of the ABox is displayed below. This describes a module `module318` with the name "nndnostr". The name attribute is a datatype property.

```
<Module rdf:ID="module318">
   <moduleHasName rdf:datatype=
   "http://www.w3.org/2001/XMLSchema#string">
   nndnostr</moduleHasName>
</Module>
```

4 Comparison of the Modeling Approaches

This case study is a comparison of the two models and query evaluation between GReQL and SPARQL and also a description of their limitations.

4.1 Comparison of the Conceptual Models

In the test case there are four possible kind of questions considered which are typical points of interest in the reengineering process. These four question types are also described in [26]. The dataset is also the GEOS Nostro system.

- The **module-function relationship** describes the functionality of the different system modules.
- Queries about the **call relationship** provide information about the relationship of the system and the functions.
- The **include relationship** contains the module inclusions and therefore also module dependencies.
- The **subclass relationship** describes the class relationship from the object oriented code modules. This contains also information like inheritance and multiple inheritance.

4.2 Include Relationship

The include relationship describes the relationship between modules. The query below selects all module pairs which are in a `include` relationship

The GReQL query in Figure 4 selects all attributes (id, name and type) of the involved modules. All variables in the GReQL query are declared in the `from` clause. The domain of a variable consists of all types of the schema i.e. all objects (vertex types) and all relations (edge types). The with clause contains the path expression `m -->{includes}inc`. The variables m and inc are declared as modules ($V\{Module\}$). The existence of an edge between these two modules is described by the predicate `-->{includes}`. It is a nested query. The report clause describes the result structure.

```
from m:V{Module}
report m.id, m.name, m.type
     from  inc:V{Module}
     with  m -->{includes}inc
     report inc.id, inc.name, inc.type
     end
end
```

Figure 4. A GReQL Query for Include.

If one is interested in the transitive closure of the include relationship, i.e. for a module m, all modules inc are searched, which are included directly by m or indirectly by an includee from m, the path predicate is expanded by the star: `m-->{includes}* inc`.

```
SELECT  ?m  ?inc
WHERE {
?m    rdf:type    a:Module .
?inc  rdf:type    a:Module .
?m    a:includes   ?inc . }
```

Figure 5. A SPARQL Query for Include.

The corresponding SPARQL query is described in Figure 5. In this example the two modules are selected without a special selection of the attributes of the module classes. In the SELECT part there are the variables $?m$ and $?inc$ for the modules. The first two triples in the WHERE clause express that the variables and therefore the result of the query are modules. For this expression the `rdf:type` statement is used. This is the predicate part of the triple. The type expression `a:Module` is a reference to the class `Module` whereas `a` is an abbreviation for the namespace prefix that is defined in the ontology. Types are covered with predicate expressions.

Queries for selecting transitive include relations are exactly the same but they are applied to another ontology with transitive role definitions.

With a similar query (Figure 6) it is possible to select only modules that include a module with the name "nndnostr". In the GReQL query an additional `with` clause is added.

```
from  inc:V{Module}
with   inc.name = 'nndnostr'
report
    from  m:V{Module}
    with  m -->{includes} inc
    report m.id, m.name, m.type
    end
end
```

Figure 6. GReQL Query with condition.

In SPARQL queries this is expressed by using the `FILTER` option (Fig. 7). This query uses a further variable $?incName$ for the name of the included module. The relationship (role) between a module and its name is expressed with the triple `?inc a:moduleHasName ?incName`. The filter expression needs this name in order to filter only those result sets which satisfy the name condition. Instead of an additional variable it is also possible to use blank nodes [30]. The `module:HasName` relationship is a `DataProperty`.

```
SELECT  ?m  ?inc ?incName
WHERE
{  ?m    rdf:type          a:Module .
   ?inc  rdf:type          a:Module .
   ?inc  a:moduleHasName   ?incName .
   ?m    a:includes        ?inc .
 FILTER  regex(?incName, "nndnostr") .}
```

Figure 7. SPARQL query with FILTER.

4.3 Call Relationship

The call relationship describes a relation between methods. It is analyzed which methods call a particular method or which methods are called by this method.

The query in Figure 8 selects all module pairs with the corresponding names which are in a call relationship, i.e. one module `(caller)` calls the other module `(callee)`.

In the corresponding SPARQL query (Figure 9) the `calls` relation is expressed as triple in the WHERE clause. The variables `?calleeN` and `?callerN` refer to the names of the methods. These variables are introduced in this query in order to get also the names of the methods in the result.

```
from   caller:V{Method}
report caller.name
      from  callee:V{Method}
      with  caller -->{calls} callee
      report callee.name
      end
end
```

Figure 8. GReQL Query for Call.

The `caller` method is in a `calls` relationship with the `callee` method. The predicate `calls` refers to the `ObjectProperty`. As in the previous case a query for all transitive connected methods is the same but it is applied to another ontology with a transitive role definition for `calls`.

```
SELECT ?caller ?callee ?callerN ?calleeN
WHERE { ?caller rdf:type  a:Method .
        ?callee rdf:type  a:Method .
        ?caller a:calls  ?callee .
        ?caller a:methodHasName ?callerN .
        ?callee a:methodHasName ?calleeN .}
```

Figure 9. SPARQL Query for Call.

The use of four variables instead of only two as in the GReQL query is a performance drawback. In GReQL it is possible to select the attributes of a vertex class as outlined in the report clause by the `caller.name` expression. SPARQL needs further variables with the corresponding graph pattern. Therefore the query evaluation demands more pattern matchings than a query without displaying the names of the methods.

As mentioned above in GReQL queries it is also possible to select edges, i.e. defining a variable in the from clause of the type edge, e.g. `e:E{calls}` defines a variable `e` of the edge relation. Since edges are modeled as `ObjectProperty` in OWL it is not possible to select edges directly with SPARQL. Such queries must be transformed to equivalent queries that select nodes and their subclasses. The drawback in this case is that a variable for an edge is replaced by two vertex variables. This increases the number of pattern matchings.

4.4 Subclass Relationship

Queries for the subclass relations are constructed in the same way. This relation describes the subclass connection of two GEOS classes.

4.5 Metrics

A further kind of queries are metrics. Which contain some measurements like counting and average values. An example for such a query in GReQL is demonstrated in Figure 10. The result is the number of all modules in the repository.

```
cnt (
from   m:V{Module}
report m
end )
```

Figure 10. GReQL Query for Counting.

One main difference in the model assumptions is the closed-world assumption in GUPRO and the open-world assumption in the OWL model. In GUPRO it is assumed that all system artefacts are contained in the repository and all referenced components are available in the repository. The user is aware of the content of the repository due to the conceptual model. In the ontology the intention is that the repository not necessarily contains all artefacts. Therefore metrics of the GEOS system are not considered in this case study.

4.6 Summary of Performance

The Table 1 outlines a short comparison of the three query kinds performed in both approaches.

Query	Tuples	eval. time (msec)	
		GReQL	KAON2
include	2935	565	2573
include_trans	2935	578	52311
include_nndnostr	65	41	3477
directCall	11318	1015	2687
transitiveCall	184506	12617	58475
directCall_Err	140	134	2599
superclasses	133	75	2126
superclasses_trans	140	77,4	53204

Table 1. Performance comparison.

In the first part of the queries the include relationship between modules is considered. The first query is the direct include relation, the second the transitive relationship and the third query selects all direct connections in which the name of the included module is "nndnostr".

The second three queries select methods that are in a call relationship. The first query `directCall` selects all methods which are in a direct relationship whereas the second query selects all methods with a transitive connection. The result of the third query contains all module pairs that are in a direct call relationship and the name of the called method is "CheckErrOut". The

performance drawback in the third query is due to further graph pattern matchings that are necessary for selecting the name for the FILTER expression.

The third part contains two queries for the `isSubClassOf` relation. The first query selects all direct connected GEOSClasses and the second query comprehends all transitive connected GEOSClasses.

5 Conclusion and Future Work

In this paper we transformed a graph-based repository that is used in reverse engineering into an OWL DL model. The graph is directly mapped into an ontology. The resulting conceptual models are similar. All components are modeled as OWL classes with the same attributes. All relations are described as object properties.

As expected for all queries of the case study the performance of GReQL is better than that of KAON2. The graph based representation and the search algorithm is more suitable for this kind of application. An advantage of GReQL is the direct access of all attributes of a vertex or edge class like the name of a vertex class or subclass. In the OWL ontology the attributes of a class are modeled in the same way but in SPARQL it is not possible to access the attributes of a class without further pattern matchings. This is due to the RDF-based triple structure of SPARQL.

A further advantage of GReQL is the possibility of selecting edges i.e. using an edge variable in the from clause. For such an edge there is a direct connection to the two corresponding vertices. The access to an vertex belonging to an edge is realized in the same way as an attribute access. There is no further search or pattern matching necessary. In SPARQL queries for OWL DL it is not possible to use a variable of an edge type since an edge is an `ObjectProperty` and this is not directly supported by the SPARQL syntax. There are at least two vertex variables necessary for replacing one edge variable. This increases the number of pattern matchings.

The performance advantages of GReQL in the transitive queries is due to the aggregation of the performance benefits in the direct connections.

Based on a case study that outlined the good performance of GUPRO the same kind of queries are used for this comparison of GUPRO and the OWL representation. The transformation of the model and the mapping of the queries is straightforward. As expected the good performance of graph-based query processing in GUPRO is not reached with OWL due to the well optimized query evaluation in GUPRO and the high expressivity of OWL, and therefore high complexity of OWL. Currently using SPARQL for querying OWL DL ontologies is not the best solution. But the research in developing and optimizing OWL DL querying is still ongoing research.

References

[1] GEOS - Global Entity Online System. Homepage.

[2] GEOS Überblick. SDS - Software Daten Service.

[3] KAON2. http://kaon2.semanticweb.org.

[4] Protege. http://protege.standford.edu.

[5] XML Schema. http://www.w3.org/TR/xmlschema-1/.

[6] RacerPro. User's Guide, Version 1.9. http://www.racer-systems.com/de/, 2005.

[7] G. Antoniou and F. van Harmelen. Web Ontology Language: OWL. In *Handbook on Ontologies*. Springer Verlag, 2004.

[8] F. Baader, I. Horrocks, and U. Sattler. *Description Logics, in: The Handbook on Ontologies in Information Systems*. Springer Verlag, 2003.

[9] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, February 2005.

[10] D. Brickley, R.V. Guha (eds.), and B. McBride. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C, 2004.

[11] G. Canfora, A. Cimitile, and U. de Carlini. A Logig-Base Approach to Reverse Engineering Tools Production. *IEEE Transactions on Software Engineering*, 18(12):1053–1064, December 1992.

[12] J. de Bruijn, R. Lara, A. Polleres, and D. Fensel. OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web, 2005.

[13] M. Dean, G. Schreiber (eds.), S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P.F. Patel-Schneider, and L.A. Stein. OWL Web Ontology Language Reference. Technical report, W3C, 2004.

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

[15] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[16] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In *10th Workshop Software Reengineering (WSR 2008)*, 2008.

[17] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In *ER'96 - Proceedings of the 15th International Conference on Cenceptual Modeling*, number 1157, pages 163–178. Springer Verlag, 1996.

[18] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 2–27, Berlin, Germany, 2006. Springer.

[19] R. C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *5th Working Conference on Reverse Engineering, Proceedings, IEEE Computer Society*, pages 210–219. 1998.

[20] U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ-Description Logic to Disjunctive Datalog Programs. Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 04), 2004.

[21] *10th International Workshop on Program comprehension*. IEEE, Los Alamitos, 2002.

[22] D. Jin, J. R. Cordy, and T. R. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *[21]*, pages 65–74. 2002.

[23] M. Kamp and B. Kullbach. GReQL – Eine Anfragesprache für das GUPRO–Repository – Sprachbeschreibung (Version 1.3). Projektbericht 8/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

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

[25] Ivan Kurtev, Jean Bézivin, and M Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.

[26] C. Lange, H. Sneed, and A. Winter. Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001), Toronto, CA, May 2001*, pages 209–218, Los Alamitos, 2001. IEEE Computer Society.

[27] B. Motik. On the Properties of Metamodeling in OWL. *Journal of Logic and Computation*, 17:617–637, 2007.

[28] B. Motik and U. Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. Proc. of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR2006), November 2006.

[29] R. Ommering, L. van Feijs, and R. Krikhaar. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.

[30] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, 2008.

[31] A. Raza, G. Vogel, and E. Plödereder. Bauhaus, A Tool Suite for Program Analysis and Reverse Engineering. In *L. M. Pinho and M. G. Harbour: Reliable Software Technologies, Ada-Europe 2006, 11th Ada-Europe International Conference on Reliable Software Technologies, Proceedings*, pages 71–82. 2006.

[32] V. Riediger, D. Werner, and A. Winter. Export und Visualisierung von GUPRO-Projektgraphen. Technical report, Universität Koblenz-Landau, Institut für Softwaretechnik, 2003.

[33] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. http://pellet.owldl.com.

[34] S. R. Tilley. Domain-Retargetable Reverse Engineering. Phd thesis, Department of Computer Science ,University of Victoria, Victoria, January 1995.

[35] K. Wong. RIGI User's Manual, Version 5.4.4, 30. June 1998.