

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

Gerd Gröner

Steffen Staab

Andreas Winter

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

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

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* [31]. Artefacts of software systems are extracted and identified from the source code and represented in repositories. These repositories reflect abstract representations of the software systems according to the maintainers needs. If the structure of the repository is made explicit [19] it is defined by conceptual modeling techniques.

There exist different realizations of a repository. In general the repository is organized in net-like structures like e.g. graphs (cf. Bauhaus [28], GUPRO [14], Rigi [33]), relations (cf. RPA [26], SWAG Kit [18], Crocopat [8]), or logic oriented data representations (cf. DATA.Tool [10], CodeQuest [17]). According to its

technological space [22] 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 the SPARQL query language.

GUPRO [14] is a graph-based approach for reverse engineering 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 [21].

The Web Ontology Language (OWL) supports conceptual modeling based on the power and expressivity of description logics. Here, the source code is described by an ontology. Based on the semantics of description logics, reasoners provide powerful analyzing and querying services for interrogating ontologies.

Topic of this paper is to compare both approaches and to elaborate differences and similarities of both technological spaces. We transformed the graph based model with repository and query language into a logic based model with corresponding modeling, representation and querying tools.

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. [23] already used GEOS for comparing graph-based analyses with a database. This case study already outlined the good performance of graph-based systems for reverse engineering applica-

tions. 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 including 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, followed by the conclusion of the paper.

2 Program Analysis Techniques

This section outlines the two different approaches for modeling and analyzing repositories to be applied to reverse engineer the GEOS system.

2.1 GUPRO and GReQL

GUPRO (Generic Understanding for PROgrams) [15, 29], is a system for analyzing and visualizing software systems and documents. GUPRO uses a graph-based repository that implements sophisticated graph traversal algorithms.

The repository uses TGraphs [13] which are attributed, directed, ordered, and typed graphs. In attributed and typed graphs the nodes and edges may have assigned attribute and type values. Ordered graphs provide an ordering for nodes, edges and incidences.

GUPRO implements efficient graph algorithms for querying and extracting information from graphs. Objects of software systems like classes, attributes, modules, methods, database tables and their interrelationships are represented as nodes and connecting edges.

GUPRO uses different tools for data extraction. For the extraction in [23] Harry Sneeds ANAL/SoftSpec is used. The abstracted data is visualized by different visualization tools and different output formats.

The conceptual model is based on the EER/GRAL [16] modeling approach. EER/GRAL is an extended entity relationship (EER) diagram augmented with the constraint language GRAL. The constraint language is used to specify integrity conditions.

2.1.1 Querying the GUPRO Repository

Analyzing TGraphs is based on the specially designed graph-based querying language GReQL (Graph Repos-

itory Query Language) [20]. GReQL is a declarative language for extracting information from the repository that accesses the queried data read only.

A GReQL query consists of 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. The **with** clause summarizes predicates which have to be fulfilled from the variables. These predicates include 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.

An example GReQL query is displayed in Figure 1. Two variables *tab* and *col* of appropriate types (**Table** and **Column**) are declared in the **from** clause. The node type is referenced with **V** (vertex). The predicates in the **with** clause contains two parts. The first part is the condition that the name of the table *tab* is 'Article' and the second is a path expression *col -->isColumnOf tab* describing that the nodes *col* and *tab* are connected by an **isColumnOf** edge. The structure of the result are name and type tuples of the corresponding nodes.

```
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.

This example also demonstrates the kind of connections which are stored in the repository between nodes of type **Table** and **Column** that represent database tables and columns.

For better program understanding, GUPRO provides various views for result visualization [15]. E.g. results of GReQL queries are represented in nested tables and additionally the corresponding source code is also shown beside the tables.

2.1.2 Tools and Infrastructure

GReQL graph querying is provided by the GUPRO Reverse Engineering Workbench, that enables interactive querying. GReQL is also part of the (J)GraLab graph libraries to facilitate GReQL-based reasoning in C++ and Java programs [15].

2.2 The OWL Model

The Web Ontology Language (OWL) [12] is the W3C standard ontology language for 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. A *datatype property* is a relation between objects and datatypes. Datatypes are all XML schema datatypes [4]. Axioms are assertions about classes, class relationships and properties.

Ontologies are used for knowledge representation as a knowledge base in information systems. 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) [7]. The formal and well-defined semantics is necessary for performing reasoning tasks.

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 complexity.

- 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 [9].
- OWL DL in general is computational efficient for reasoning tasks, except of a high worst case complexity. OWL DL only covers a subset of the OWL-Full language. OWL DL is a variant of the DL-language $\mathcal{SHOIN}(\mathcal{D})$. 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. E. g. OWL Lite does not support disjunction and enumerations and provides only cardinality restriction with 0 and 1. OWL Lite is a variant of the DL-language $\mathcal{SHIF}(\mathcal{D})$.

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

Figure 2. A SPARQL query example.

OWL DL and OWL Lite benefit 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 to use *nominals* i. e. individual names in class descriptions.

Further restrictions are *explicit typing* [6] i. e. all resources have to be explicitly stated. There are no cardinality restrictions on transitive properties [6].

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 [27] 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 comprise triples of the kind **subject**, **predicate** and **object**. These triples are depicted in the **WHERE**-part of SPARQL queries. The **SELECT**-part contains query variables which appear also in the triples of the **WHERE**-part. These patterns are matched against the RDF triples (RDF graph). The query result is a set (solution sequence) in which every element is a data element of the RDF graph that matches to a variable of the pattern.

Figure 2 demonstrates a simple SPARQL query, similar to the GReQL query in Figure 1. In the **SELECT** clause one variable $?name$ is defined. Variable identifiers always start with ?. The **WHERE** clause contains the query pattern with five triple patterns. The variables $?table$ and $?col$ are not in the result set. The first and third term define the type of the variables $?table$ and $?col$. The `isColumnOf` relation is expressed with the fourth pattern. The result are the names of all columns belonging to the table with the name "Article".

For constructing more complex queries there exist algebraic operations like the **FILTER** operation 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 as query language.

2.2.2 Tools and Infrastructures

A comfortable modeling tool for ontologies is Protege [3], 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 datatype properties, and it provides reasoning capabilities such as consistency checking and classification. Protege supplies interfaces to connect to reasoners like Pellet [30], RacerPro [5] or KAON2 [2] 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.

One difference from KAON2, compared to Pellet and RacerPro, is the transformation of knowledge bases to disjunctive datalog programs while the other reasoners implement optimized tableau algorithms. It is argued in [25] that the field of disjunctive databases reasoning over large data sets is extremely studied, and KAON2 exploits the experiences of this research field by implementing these reasoning algorithms in order to provide scalable reasoning services over large data sets.

Before reasoning can be applied in KAON2, the generation of the disjunctive datalog program has to be performed. The rule set mainly depends on the TBox. The reasoning component of KAON2 consists of a theorem prover for the transformation step. The *disjunctive datalog engine* works on the rule set.

Since the TBox is manageable and not complex whereas the ABox is large with about 9,500 individuals, KAON2 seems well suited to be used in this case study. As described in [25] the reasoning algorithm of KAON2 is optimized for large ABoxes and small TBoxes. Thus, KAON2 provides a scalable querying service. The query engine supports the syntax of the SPARQL query language. RacerPro provides an

other powerful reasoner for instance retrieval over large knowledge basis [24]. Witte et al. [32] used RacerPro in a maintenance application which is focused on an unified and integrated representation of all software artefacts, including documents.

Reasoning services are also useful in reverse engineering applications. Some standard reasoning tasks are:

- The *classification* of concepts is an inference that computes all subclass relations of the TBox. The result is a concept subsumption 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 relationships between objects. Second, the extraction and transformation process is based on the conceptual models structure. Source code is parsed and extracted according to the conceptual model. Third, the information extraction, e.g. the queries performed on the repository depend on the conceptual model.

3.1 The GEOS System

GEOS (Global Entity Online System) [1] 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 the following, we apply the two reverse engineering approaches to the GEOS Nostro component.

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

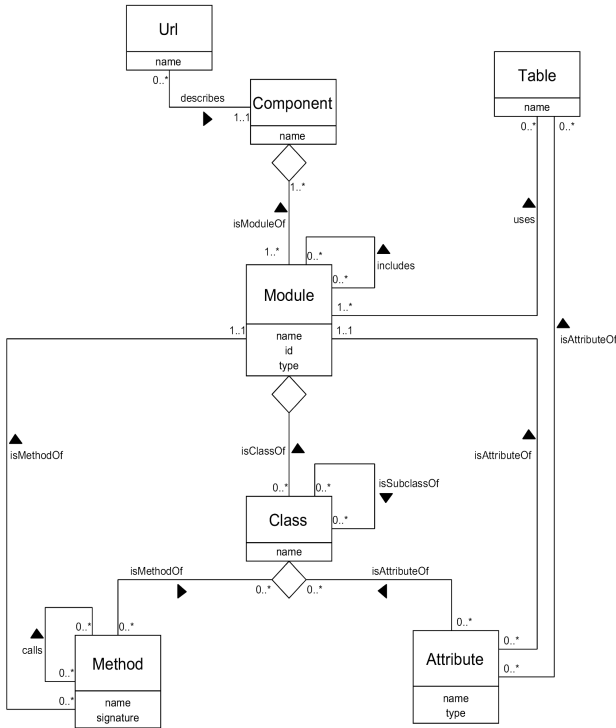


Figure 3. The Conceptual Model of the GEOS System.

of more than 6,279 source files and about 2,364,652 lines of code [23].

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 databases, methods and attributes.

A **conceptual model** defines this structure for analyzing 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. Collections provide interfaces to other modules. A *Module* is a source file that might be in an *include*-relation to other modules. An *Uri* is an identifier with a name attribute. It describes exactly one *Component*. *Classes* are special parts of modules in object oriented

systems. *Modules* and *Classes* contain *Methods* and *Attributes*. *Tables* refer to the database tables. They are accessed from *Modules* by *uses*-relations. The *is-SubclassOf* relation expresses the class hierarchy. Inclusion of *Modules* is expressed by the *include* relation and method calls are described by the *calls* relationship.

3.3 Repository Based Reverse Engineering

Facts from the GEOS source code were extracted according to the conceptual model in Figure 3 by using ANAL/SoftSpec and are stored in a relational database. To provide analysis with GUPRO, filters were created, to translate the DB content into TGraphs [23]. Further filters were used to transform the resulting TGraphs into OWL ontology representations.

For reverse engineering tasks, this information is extracted from the repository. Querying repositories in software reengineering enables a powerful mechanism for analyzing software systems instead of searching in large documentations and diagrams [21].

Like GUPRO/GReQL, OWL DL and OWL Lite realizes a conceptual modeling approach. In conceptual modeling there is a strict separation of the *conceptual* and the *data* model. 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 contains all individuals belonging to the concepts. These instances refer to the ABox of ontologies.

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 sense. Assertions are expressed in GUPRO by 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 [11]. In OWL DL cardinality restrictions can infer existence and equivalence of individuals.

The complexity of the DL ontology is $\mathcal{ACUHLINC}(\mathcal{D})$. The \mathcal{AC} is the DL base language. The \mathcal{U} allows union of concepts, the \mathcal{H} is for describing role hierarchies such as subproperties ($\text{rdfs:subPropertyOf}$), 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 GEOS 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. Node classes are represented as OWL classes. The OWL class `Node` acts as superclass for all node classes, and all node classes defined in the conceptual model like modules and methods are appropriate subclasses. The definition of class `Module` is presented below. Attributes of these classes are defined as datatype properties.

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

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

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

There is a difference in modeling and querying the transitivity property of roles. Per default, the `ObjectProperty` role is 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. The following relations have to be declared transitive: the `includes` role between modules, the `isSubClassOf` role between `GEOSClasses` and the `calls` role between methods. Reflexive roles are considered analogously.

In GReQL queries the path expression specifies whether a direct connection or a transitive connection is focused. This is expressed in GReQL with `-- >` for the direct connection and by using the Kleene star `-- > *` for a reflexive and transitive path. In OWL it is necessary to specify the edge (`ObjectProperty`) as a

transitive property 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 another ontology with transitive properties (`TransitiveProperty`) is used. The queries are the same for direct and transitive edge connections but the ontologies are different.

Instance Graphs are transferred to the appropriate ABox. A part of the ABox, representing the GEOS-TGraph is displayed below. This describes 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

The following section presents the case study comparing the two models and query evaluations in GUPRO/GReQL and OWL/SPARQL and gives also a short description of their limitations.

4.1 Comparison of the Conceptual Models

The case study in [23] considered the following kinds of questions, which address typical aspects in reverse engineering.

- The **include relationship** describes the containment structure of the system and addresses module dependencies.
- Queries about the **call relationship** provide information about the possible dynamic behavior of the system and shows possible method/function invocations.
- The **subclass relationship** is also called the inheritance relationship. Queries about this relationship contain information about generalization or specialization of a particular class.
- **Metrics** describes countable properties of software systems.

This test case covers three of these possible kinds of queries.

4.2 Include Relationship

The include relationship describes relationships between modules. The example query 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 (node types) and all relations (edge types). The `with` clause contains the path expression `m -->{includes}inc` specifying the nesting structure. 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}`. 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
```

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 or indirectly, the path predicate is only expanded by the Kleene star: `m-->{includes}* inc`.

The corresponding SPARQL query is described in Figure 5. In this example the two modules `?m` and `?inc` are selected without a special selection of the attributes of the module classes. 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. The type expression `a:Module` is a reference to the class `Module` whereas `a` is an abbreviation of the namespace prefix that is defined in the ontology. Types are covered with predicate expressions. The third triple demands the include relation between `?m` and `?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.

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 predicate 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
```

Figure 6. GReQL Query.

The appropriate SPARQL query uses the `FILTER` option (Fig. 7). This query requires a further variable `?incName` for the name of the included module. The relationship (role) between a module and its name is expressed by the triple `?inc a:moduleName ?incName`. Instead of additional variables it is also possible to use blank nodes [27]. The `module:HasName` relationship is a `DatatypeProperty`.

```
SELECT ?m ?inc ?incName
WHERE
{ ?m   rdf:type    a:Module .
  ?inc  rdf:type    a:Module .
  ?inc  a:moduleName ?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. Call relations are used to detect which methods call other methods or which methods are called by other methods.

The query in Figure 8 selects all caller/callee pairs with the corresponding names.

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

Figure 8. GReQL Query for Call.

In the corresponding SPARQL query (Figure 9) the `calls` relation is expressed by a triple in the WHERE clause. The variables `?calleeN` and `?callerN` refer to

the names of the methods. These variables are introduced in order to get also the names of the methods in the result.

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 leads to a performance drawback. Due to the local representation of attributes in nodes, GReQL directly accesses attributes of nodes 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, it is also possible to select edges in GReQL queries, 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 adjacent nodes. The drawback in this case is that a variable for an edge is replaced by two node variables. This, again increases the number of pattern matchings.

4.4 Subclass Relationship

Queries about the subclass relation (inheritance relationship) contain querying for specializations, super classes or multiple inheritance of `GEOSClasses`. These queries are constructed in the same way like the queries for the include relationship.

4.5 Metrics

A further kind of queries are metrics which contain measurements like counting and average values. An example for such a query in GReQL is demonstrated in Figure 10. This query counts the number of all modules stored 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 ontology-based modeling 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_mndnostr	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 "mndnostr".

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 for KAON2 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 further queries similar to the include relationships, analyzing the `isSubClassOf` relation. The first query selects all direct connected `GEOSClasses` and the second query comprehends all transitive connected `GEOSClasses`.

There is a remarkable performance drawback of KAON2 with transitive roles compared to GReQL. KAON2 is here used as a general purpose reasoner without special optimizations for typical reengineering questions like transitive role connections. For such applications some optimizations of the reasoning algorithms would improve the performance.

5 Case Study Result

This case study is based on the same data and queries that were used in the comparison of GUPRO/GReQL with a relational database approach by Lange et al. [23]. In this paper we compared GUPRO/GReQL with an OWL/SPARQL approach. To ensure comparability, the conceptual model, that was designed for GUPRO applications was directly transformed into an OWL model.

As expected, the performance of GReQL querying is much better than that of KAON2 for all queries applied in this case study. The graph based representation and the query algorithm is more suitable for this kind of applications. An advantage of GReQL is the direct access of all attributes of nodes and edges. In the OWL ontologies attributes of classes are modeled in the same way but in SPARQL it is not possible to access attributes 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 nodes. The access to a node of an edge is realized in the same way as for attributes. Compared to SPARQL, there is no further search or pattern matching necessary.

In SPARQL syntax it is not possible to directly access edges, since edges are represented as `OWL-ObjectProperties`. There are at least two node variables necessary for replacing one edge variable. This increases the number of pattern matchings. Protege with its integrated query panel (Version 4.0 with Pellet-Reasoner 1.5) partially supports variables of property types. But, the performance of query processing with Protege and the integrated reasoner is worse than the performance of KAON2.

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

A general benefit of OWL modeling is to dynamically extend the class structures defined in the `TBox`. Instead of referring to a concrete concept, it is possible to describe a concept or only some essential features of a concept in an abstract and general concept description. Such a class description can be used like a concrete class, e. g. extract all individuals that satisfy this description, i. e. they belong to this class or they would belong to such a concept if it exists.

Reverse engineering will benefit from those facilities, by reusing intermediate results. Whereas OWL provides a general and flexible means to define classes dynamically and independent from concrete instances, GReQL facilitates the calculation of subsets of graph elements, fulfilling certain properties. Instead of reporting a result for a given graph, a GReQL `store`-clause, stores all graph elements, matching the type constraints, given in the `from`-clause and satisfying the predicates in the `where`-clause. The resulting set can be reused in further queries like node- or edgesets defined by classes.

6 Conclusion and Future Work

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

Based on a case study that outlined the good performance of GUPRO/GReQL compared to a relational database, the same kind of queries were used for comparing GUPRO/GReQL and OWL/SPARQL. Transformation of the conceptual model, instance data and mapping of queries was 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.

Currently, using SPARQL for querying OWL DL ontologies is not an adequate solution. But the research in developing and optimizing OWL DL querying is still ongoing research.

A next step in our research is to exploit the expressivity of OWL DL and reasoning technology in order to provide new modeling and querying facilities to be applied in reverse engineering.

References

- [1] GEOS - Global Entity Online System. Homepage Software Daten Service: <http://www.sds.at/>.
- [2] KAON2. <http://kaon2.semanticweb.org>.

- [3] Protege. <http://protege.stanford.edu>.
- [4] XML Schema. <http://www.w3.org/XML/Schema>.
- [5] RacerPro. User's Guide, Version 1.9. <http://www.racer-systems.com/de/>, 2005.
- [6] G. Antoniou and F. van Harmelen. Web Ontology Language: OWL. In *Handbook on Ontologies*. Springer Verlag, 2004.
- [7] F. Baader, I. Horrocks, and U. Sattler. *Description Logics, in: The Handbook on Ontologies in Information Systems*. Springer Verlag, 2003.
- [8] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, February 2005.
- [9] D. Brickley, R.V. Guha (eds.), and B. McBride. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C, 2004.
- [10] 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.
- [11] J. de Bruijn, R. Lara, A. Polleres, and D. Fensel. OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web, 2005.
- [12] 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.
- [13] J. Ebert. A Versatile Data Structure For Edge-Oriented Graph Algorithms. *Communications ACM*, 30:513–519, 1987.
- [14] 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.
- [15] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In *10th Workshop Software Reengineering (WSR 2008)*, 2008.
- [16] 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 Conceptual Modeling*, number 1157, pages 163–178. Springer Verlag, 1996.
- [17] 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.
- [18] 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.
- [19] D. Jin, J. R. Cordy, and T. R. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *10th International Workshop on Program Comprehension.*, pages 65–74. 2002.
- [20] 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.
- [21] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In P. Nesi and C. Verhoef, editors, *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society, 1999.
- [22] Ivan Kurtev, Jean Bézivin, and M Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.
- [23] 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.
- [24] R. Möller, V. Haarslev, and M. Wessel. On the Scalability of Description Logic Instance Retrieval. In *Proceedings of the 29th Annual German Conference on Artificial Intelligence*, 2006.
- [25] 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.
- [26] 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.
- [27] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, 2008.
- [28] 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.
- [29] V. Riediger, D. Werner, and A. Winter. Export und Visualisierung von GUPRO-Projektgraphen. Technical report, Universität Koblenz-Landau, Institut für Softwaretechnik, 2003.
- [30] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. <http://pellet.owld.com>.
- [31] S. R. Tilley. Domain-Retargetable Reverse Engineering. Phd thesis, Department of Computer Science, University of Victoria, 1995.
- [32] R. Witte, Y. Zhang, and J. Rilling. Empowering Software Maintainers with Semantic Web Technologies. In *4th European Semantic Web Conference (ESWC 2007)*, 2007.
- [33] K. Wong. RIGI User's Manual, Version 5.4.4, 30. June 1998.