Are Graph Query Languages Applicable for Requirements Traceability Analysis?

Michael Rath[1], David Akehurst[2], Christoph Borowski[2], and Patrick Mäder[1]

[1] Technische Universität Ilmenau, Ilmenau, Germany
michael.rath@tu-ilmenau.de, patrick.maeder@tu-ilmenau.de
[2] itemis AG, Lünen, Germany
david.akehurst@itemis.de, christoph.borowski@itemis.de

Abstract. [**Context & motivation**] Maintaining traceability in development projects is a costly and resource expensive task. Once established, it can be used to answer questions about the product and its development process. [**Question/problem**] Typically, generic query languages for relational databases are used to formulate these questions. Additionally, specific traceability query languages have been proposed, but they are not widely adopted and show other weaknesses. [**Principal ideas/results**] Relationships among development artifacts span a rich traceability graph. Directly operating on this data structure rather than on a relational representation by using a graph query language may overcome the limitations of the current workflow. [**Contribution**] In this paper, typical traceability questions from a user survey were extracted. Using these, the resulting query expression for four different languages are briefly compared.

1 Introduction

Requirements traceability plays an important role in software development processes. Traceability analysis aids stakeholders in satisfying information needs such as requirements validation, coverage analysis, impact analysis, and compliance verification. Considerable effort needs to be invested to establish traceability [1]. However, the captured information is only of limited use without being able to effectively query and analyze it. Today's developments tools, like IBM Rational DOORS™, Enterprise Architect, HP Quality Center™, and Atlassian Jira[3] typically allow users to write queries in SQL-like languages. These languages require in depth knowledge of the underlying relational data schema [5]. Though, the nature of data, i.e., the artifacts and their relations, inherently span a graph data structure. This concept is not only difficult to model in traditional table oriented databases, but its complexity also propagates to the formulation of queries (e.g., multiple joins). NoSQL approaches gained a lot of attraction in the database world. One approach, graph databases [9], efficiently

[3] JQL query, https://confluence.atlassian.com/jiracore/blog/2015/07/search-jira-like-a-boss-with-jql

manages information in nodes and edges between them. Graph query languages were designed to traverse such graphs and to gather requested information.

In this paper, we initially investigate whether graph query languages are applicable for traceability analysis. We conducted a user survey with stakeholders of software projects and collected queries common to their daily work. These queries were systematically analyzed to extract basic features to be provided by a suitable query language. We selected two graph query languages and formulated concrete queries based on participants' replies and compare them with those written in SQL and VTML, a visual language specifically designed for traceability related information retrieval.

2 Query languages selection

The **Structured Query Language (SQL)** was introduced in the 1970s for managing data in relational database management systems (RDBMS). SQL matured over the last decades, has a large base of users familiar with its syntax, and is widely used in industry today. We selected SQL for our study as an industrial baseline for analyzing traceability information.

The **visual query language (VTML)** [4, 5] was explicitly designed to intuitively query traceability information. Users model queries without explicit knowledge of the data's schema and its distribution. The visual representation of a query closely resembles UML class diagrams enhanced by specific annotations for selecting and filtering data. VTML behaves like a meta-language, unbound to a specific underlying query language. Their inventors demonstrated a transformation from VTML to SQL. We selected VTML for our study as a scientific baseline for how traceability information can be analyzed today.

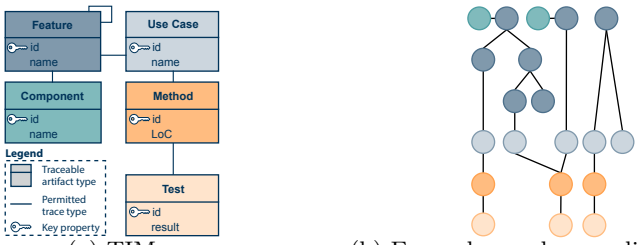
Cypher[4] is a declarative graph query language accompanying the Neo4j graph database. Cypher's syntax is inspired by SQL. Most keywords, their semantics, and the resulting query structure are borrowed making it intuitive to SQL users and easing the transition. We selected Cypher for that reason.

Gremlin refers to a graph traversal machine and language designed and developed by the Apache TinkerPop project [8]. The graph traversal machine can run on different graph databases, including Neo4j. As a query language, gremlin is a domain-specific language (DSL) built on Groovy targeting the Java Virtual Machine. In Gremlin, queries can be expressed either in an imperative or declarative way. We selected Gremlin because of its turing completeness.

3 User Survey

In September and October 2016 we conducted a user survey with engineers working in different industrial domains. Participants covered the areas automotive electronics and mechanics; embedded and distributed systems; logistics; and telecommunications. In particular, we interviewed nine participants employed as

[4] https://neo4j.com/docs/developer-manual/current/cypher/



(a) TIM (b) Example graph according to TIM

Fig. 1: TIM extracted from a participant's traceability queries (left) and example traceability graph complying to the TIM (right).

software developers (4), requirements engineers (3), and software architects (2). Out of the 35 asked questions about sociodemographic, development process, and reporting, we were interested in information like

What is the role of requirements traceability during your daily work?

What relevant information do you use for analyses, including data and metrics?

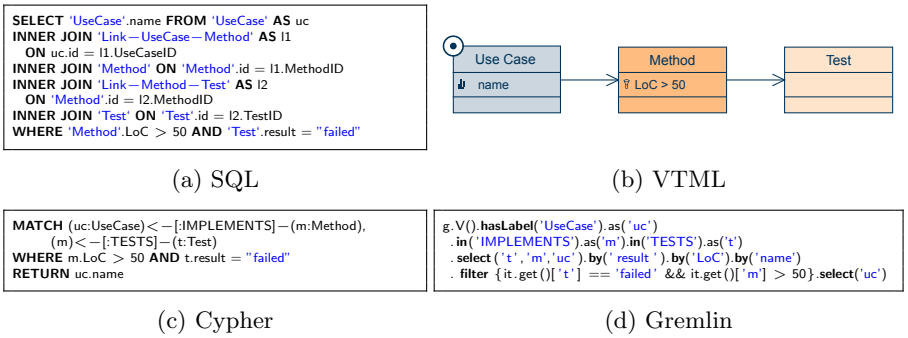
Given answers were systematically analyzed. At first, we derived an example set of queries to answer typical traceability questions, e.g. existing link patterns among artifacts and traceability metrics. Using these, a traceability information model (TIM) per participant was extracted. Additionally, we determined basis operations (e.g. artifact selection, filtering) that are to be supported by a query language expressing the queries.

3.1 Traceability Information Model (TIM)

A traceability information model is a meta-model defining prescribed traceability for a project and providing the context in which queries can be specified and executed [7]. We applied a two step approach to extract a TIM per participant of our study. In *Step 1*, we asked users to formulate trace queries, which would help them in their daily business. In *Step 2*, we annotated development artifacts and prescribed relationships involved in these queries. One example of an extracted TIM is shown in Figure 1a. The TIM consists of five development artifacts and five traceability relations. *Features* belong to architectural *components*, can be refined into more fine-grained features, and can eventually be decomposed into *use cases*. A *method* implements one or more use cases and is tested by one or more *tests*. Figure 1b exemplifies a possible traceability graph complying to this TIM. The nodes represent artifacts with their type coded in color. The edges refer to trace links. To illustrate queries, which include ones checking consistency, this graph is on purpose not complete with respect to the TIM.

3.2 Basic query language operations

We identified five operations that a query language must at least support to be applicable for the gathered queries: ❶ selecting artifacts with a specific type,



(a) SQL (b) VTML

(c) Cypher (d) Gremlin

Fig. 2: Example query: "Find all uses cases implemented by a method with > 50 lines of code that have failed test cases" using the four selected query languages.

e. g., use cases or tests; ❷ retrieving relations between artifacts, e.g., methods implementing use cases; ❸ selecting artifact and trace properties to be part of a query's result, e.g., the name of a component; ❹ filtering artifacts based on their properties satisfying a predicate, e. g., methods with more than 50 lines of code; and ❺ applying aggregation functions, e. g., counting artifacts.

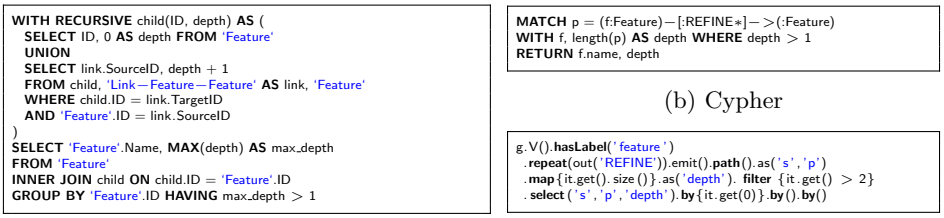
3.3 Example trace queries

We populated sqlite[5] as well as matching Neo4j databases with example artifacts and trace links according to the TIMs extracted per participant. We used these to manually execute all queries formulated by our participants, the written SQL statements and the ones generated from the VTML representation on the sqlite database; and the Cypher and Gremlin statements on the Neo4j database. Below, we discuss two trace queries selected from the set gained in the user survey.

Query 1: Which use cases implemented by method(s) with more than 50 lines of source code have failed test cases.

Finding use cases with failed tests is an important operation. Additionally, a filter is applied selecting only methods that are considered complex based on the lines of code as basic measure. The resulting queries for all selected languages are shown in Figure 2. The query involves multiple artifacts (use case, method, and test) as well as traces. In fact, traceability queries deal to a large extent with the existence of traces between artifacts [4]. In SQL, querying relations requires multiple join statements and in depth knowledge about the data's schema. Since the syntax of SQL aims to resemble natural English language, the remaining part of query is obvious. VTML's graphical notation is comprehensible to everybody with a basic knowledge of UML. Artifacts and traces are easy to spot, because of its declarative, query by example approach. Cypher is very similar to SQL, because of the borrowed keywords and syntax. The major difference being the powerful `MATCH` clause, used to describe paths in the graph, in our context

[5] https://sqlite.org/



```
WITH RECURSIVE child(ID, depth) AS (
  SELECT ID, 0 AS depth FROM 'Feature'
  UNION
  SELECT link.SourceID, depth + 1
  FROM child, 'Link-Feature-Feature' AS link, 'Feature'
  WHERE child.ID = link.TargetID
  AND 'Feature'.ID = link.SourceID
)
SELECT 'Feature'.Name, MAX(depth) AS max_depth
FROM 'Feature'
INNER JOIN child ON child.ID = 'Feature'.ID
GROUP BY 'Feature'.ID HAVING max_depth > 1
```

(a) SQL

```
MATCH p = (f:Feature)-[:REFINE*]->(:Feature)
WITH f, length(p) AS depth WHERE depth > 1
RETURN f.name, depth
```

(b) Cypher

```
g.V().hasLabel('feature')
  .repeat(out('REFINE')).emit().path().as('s','p')
  .map{it.get().size()}.as('depth'). filter {it.get() > 2}
  .select('s','p','depth').by{it.get(0)}.by().by()
```

(c) Gremlin

Fig. 3: Example query "Which features have a refinement depth larger than one and what is their depth?" using three out of the four query languages. Currently it is not possible to formulate this query with VTML.

traces between artifacts. It substitutes the complicated `INNER JOIN` clauses used in SQL. Contrasting all other languages, the Gremlin query uses an imperative style, i. e., navigating the underlying graph data structure step by step[6]. It starts at graph nodes representing use cases (`g.V().hasLabel()`) and explicitly follows specific edges (e. g.,`in('IMPLEMENTS')`). However, a Gremlin query is actually a set of Groovy statements, which might be an obstacle for users.

Query 2: Which features have an refinement depth > 1 *and what is their depth?*

The second query implements the traceability metric depth counting layers that traceability extends up- and downwards from a given layer [2]. For our example, a feature can be refined multiple times (see self-relation in TIM), which requires recursive querying to follow a relation arbitrarily often (see Figure 3). With common table expressions (CTE), a rather recent concept of SQL [6], the query is possible but quite complex with SQL. Currently, it is impossible to formulate Query 2 with VTML due to its missing notion of recursion. Advanced pattern matching and variable length relations make it easy to formulate this query in Cypher. Its core being expressed as `(:Feature)-[:REFINE*]->(:Feature)`. The partial expression `repeat(out('REFINE')).emit().path()` serves the same purpose in Gremlin.

4 Evaluation and Conclusions

Overall we tried to formulate 45 trace queries from the user survey in all four languages. With this knowledge, we conducted an early evaluation using three evaluation criteria for query languages as defined by Jarke et al. [3] (see Table 1).

Effort measures the amount of syntactic elements (tokens) the user needs to enter in order to pose the query. VTML and Cypher perform best in this category, because of their versatile syntax elements and terse notation. *Readability* describes the difficulty to understand a query. Being a visual language, VTML is the clear winner. Gremlin, with its programing language syntax, is hard to read.

[6] Gremlin also supports declarative programming using the `match()` step.

Language	Effort	Readability	Expressiveness
SQL	medium	medium	medium
VTML	low	high	low
Cypher	low	medium	high
Gremlin	medium	low	medium

Table 1: Comparison of the four query languages across all queries regarding *effort*, *readability*, and *expressiveness*.

We define *expressiveness* as the ability to successfully state a query as well to express complex computations in intuitive ways. Cypher with its few but powerful clauses is the best and VTML the worst, because currently some essential features are missing and therefore some queries cannot be executed.

In this paper, we studied the applicability of graph query languages for requirements traceability analysis. Appropriate query languages are rare and users are often forced to use generic ones like SQL. Based on a relational table model, this is not a natural fit when queried data are graphs of artifacts connected by trace links. Therefore we selected two graph query languages (Cypher and Gremlin) and a visual language (VTML), designed for traceability analysis, and compared them to SQL. A preliminary user survey collected queries from engineers working in the field of requirements analysis. We found that graph query languages can be successfully applied for traceability analysis. Future work will provide a more detailed discussion and will include an in depth language comparison and performance analysis.

Acknowledgment We are funded by the BMBF grants: 01IS14026A, 01IS16003B, DFG grant: MA 5030/3-1 and by the EU EFRE/TAB grant: 2015FE9033.

References

1. Cleland-Huang, J., Gotel, O.C.Z., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: Trends and future directions. pp. 55–69. ACM Press (2014)
2. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer-Verlag New York, Inc., New York, NY, USA, 3rd edn. (2010)
3. Jarke, M., Vassiliou, Y.: A framework for choosing a database query language. ACM Computing Surveys 17(3), 313–340 (Sep 1985)
4. Mäder, P., Cleland-Huang, J.: A Visual Traceability Modeling Language. In: Proc. 13th International Conference on Model Driven Engineering Languages and Systems. Springer (2010)
5. Mäder, P., Cleland-Huang, J.: A visual language for modeling and executing traceability queries. Software & Systems Modeling 12(3), 537–553 (Jul 2013)
6. Melton, J., Simon, A.: SQL:1999: Understanding Relational Language Components. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
7. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Trans. Softw. Eng. 27(1), 58–93 (Jan 2001)
8. Rodriguez, M.A.: The Gremlin graph traversal machine and language (invited talk). pp. 1–10. ACM Press (2015)
9. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. 41(1), 50–60 (Apr 2012), http://doi.acm.org/10.1145/2206869.2206879