TQL: A Query Language to Support Traceability

Jonathan I. Maletic and Michael L. Collard
Kent State University
Kent, Ohio, USA
jmaletic@cs.kent.edu, collard@cs.kent.edu

Abstract

A query language for traceability is proposed and presented. The language, TQL, is based in XML and supports queries across multiple artifacts and multiple traceability link types. A number of primitives are defined to allow complex queries to be constructed and executed. Example queries are presented in the context of traceability questions. The technical details of the language and issues of implementation are discussed.

1 Introduction

To fully realize the potential usefulness of traceability between software artifacts we feel that one must be able to ask questions about the current state of a system in the context of traceability. That is, the traceability links and the artifacts represent a graph and we want to easily query this graph to determine the state of the system. There are a number of pertinent questions one can ask. Is a particular requirement actually implemented in the design and realized by source code? How does a change impact the requirements? How does a change to the requirements impact the safety of the system? Is a given set of requirements covered by test cases? These questions rely on the ability to construct and ask queries on the traceability between artifacts.

We are a part of a group of researchers (including J. Hayes and J. Cleland-Huang) who are developing a framework called *Traceability*$^{+}$. The framework supports services that deliver higher-level functionality (hence the plus) through utilizing and enhancing the underlying traceability links to perform useful services for the user. Queries are specified using the Trace Query Language (TQL) and supporting tools. Here, we present some of the technical details behind the development of TQL and the traceability graph model over which the queries are applied. Artifact, link representation, and TQL are all based in XML.

Artifact representation is discussed briefly in section 2. The traceability graph and link representation is discussed in section 3. Section 4 presents details of TQL and a number of example queries. Related work and conclusions follow.

2 Artifact Representation

For TQL all artifacts are represented in XML, enabling us to leverage a wide array of XML tools and infrastructure (see Figure 1). The artifacts of interest include any part of the software lifecycle including requirements, design, code, etc. While all artifacts must be in XML, a simple translator function can be written for the vast majority of artifacts [9].

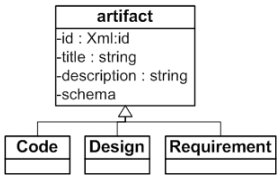

Figure 1. General artifact model.

For source code, we use srcML [4]. The srcML translator, *src2srcml*, converts source-code (in a text format) to the XML representation (i.e., srcML) in a stream-oriented (SAX) approach. The translator can work at 16 KLOC per second, and the entire Linux kernel containing 228 MB of text can be translated to the srcML representation in under ten minutes.

For design, the classML [13] format can be used. Alternatively, XMI could be used however it is not a document oriented format. There are a number of document oriented XML formats for requirements [3] and most are appropriate. Whether to store the XML view of the artifact, or convert on the fly, is an optimization decision.

3 Traceability Graph

We now discuss the components and semantics of the link model used in our approach [13]. Figure 2 describes our link model. We borrow concepts from the XML Linking Language, XLink but do not follow the XLink implementation scheme. TQL processes the links and interprets the semantics as needed by the users requirements.

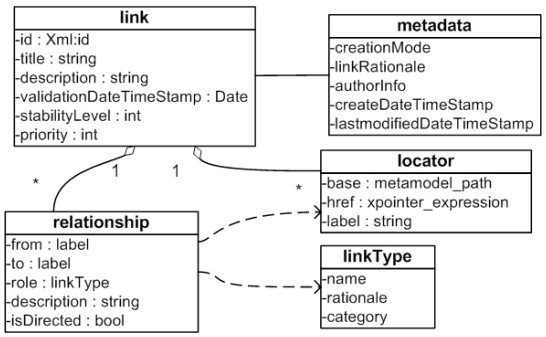

Figure 2. Traceability link model.

A traceability link is uniquely identified by its id represented in W3C's xml:id recommendation. The two crucial classes in our link model are the locators and relationships. A link has several locators associated with it. A locator consists of a base, href and a label. The base gives the path to a document within a model and href is an XPointer expression. Using XPath we can address paths within elements of our structured artifact models such as srcML. The label within a locator is used in the relationships to identify directionality. A relationship consists of a source and a sink denoted by the *from* and *to* attributes. This corresponds to the arc feature within the XLink standard. The linkType class describes link semantics.

With the link and artifact models, the traceability graph can be defined as in Figure 3. The graph consists of a number of artifacts connected by links. TQL does not depend explicitly on any one particular artifact representation however certain aspects of the queries do rely some details of the schema.

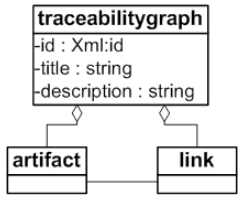

Figure 3. Model of the traceability graph.

4 Trace Query Language

A query language for traceability has many issues to deal with. First, traceability links are formed between artifacts with heterogeneous formats. Although it would be easier for TQL to work only with a single, universal format, it would prevent the query from including details of the individual formats. We want the queries to be able to include any details for any artifact form. Secondly, there may be many different types of links between two artifacts. The query language needs to provide features to use the links between artifacts, but also to work directly with the links.

TQL is built on top of the XML addressing language, XPath (see www.w3.org). XPath is a good choice for this problem as it is standard, is widely implemented, and forms a subset of more complete query languages, i.e., XQuery. In this work we will focus on the extraction of the artifacts and parts of artifacts that are important to answer the queries. The embedding of the XPath queries into XQuery primarily provides transformation options after the data is extracted.

Queries written directly in XPath require explicit knowledge of the format/representation of the artifacts, and the link representation. In order to work at the level necessary for traceability of artifacts and links, we use XPath extension functions to hide the low-level details of the artifact and link formats. XPath extension functions can be written directly in XPath 2.0. Also, many tools that support XPath 1.0 provide features to implement these functions, e.g., XSLT allows for the creation of XPath extension functions in XSLT. As such, TQL allows for full use of XPath, with a set of extension functions that mirror the traceability abstraction of the system.

4.1 Artifact Primitives

We now describe some of the primitives (extension function) used in TQL. The first issue is how to reference the different artifacts in the system. TQL includes a number of intuitive functions that provide access to specific artifacts:

```
art:UseCase()
art:Requirement()
art:Design()
art:Code()
art:TestCase()
art:BugReport()
```

The extension function `art:Artifact()` represents all artifacts in the system and is the union of all specific types of artifacts. Note that XPath extension functions are required to have a non-default prefix, e.g., `art:Artifact()`. For clarity, we leave out the artifact prefix `art:` in following examples.

The view of the artifact given by the extension function is fairly simple. For example, the function `Requirement()` provides access to all of the individual requirements in the system. To refer to a specific requirement a parameter may be provided as in `Requirement(X)`. In general a parameter is mapped to a particular id in the artifact, although this can be customized. Further details of the underlying representation can be easily extracted. For example, the expression `Code(x)//src:class` returns all the classes in a given piece (i.e., file `x`) of code that are represented in the srcML format. Note that this part of

the query depends on the structure of this artifact, and can utilize the full power of XPath to select parts of the document.

4.2 Traceability Primitives

A number of key primitives have been defined that allow tracing to and from artifacts over different link types. The two most general are the following:

```
tql:traceTo(from, to)
tql:traceFrom(to, from)
```

The first returns all the artifacts (i.e., nodes in the traceability graph) in the set `from` that trace directly or indirectly to an artifact in the set `to`. The prefix `tql` is used for the namespace for our language in XML. The second returns all the elements in the set `to` that trace directly or indirectly from an artifact in the set `from`. Both of these operations compute a *transitive closure* over all available traceability graphs to get not only the direct, but the indirect links. As an example, the following TQL query returns all requirements that trace to source code:

```
tql:traceTo(Requirement(), Code())
```

To refer to a specific part of the source code, the function can be parameterized, e.g., `Code(X)`. Because these TQL primitives find the transitive closure, all possible paths between requirements and code are used. To specify a particular path in the traceability graph, i.e., through a third specific artifact, a nesting of the primitives are used. For example, the following query results in all requirements that trace to design that traces to code, i.e., all the requirements that have resulted in design that has been implemented.

```
tql:traceTo(Requirement(),
            tql:traceTo(Design(),
                        Code(foo)))
```

Obviously, the results of the second query will be a subset of the first query. These primitives facilitate transitivity across models, meaning that the framework is not limited to tracing across a single link from one artifact to another but can also support queries that traverse three or more artifact types. An advantage of this extended traceability is that it allows insights into relationships across a broader swath of the development process and records the results of these inquiries for future use.

The following primitives give us access to the edges (links) in the traceability graph. In general, these are used to implement other extension functions (artifact primitives), or to form a subset of the edges to consider. The first link primitive returns the set of links between two sets of artifacts:

```
tql:link(to, from)
```

This forms the transitive closure of the links between the two sets of artifacts.

To find only the direct links between two sets of artifacts the following primitive is used:

```
tql:directlink(to, from)
```

Given a specific set of artifacts, it is useful to know all the links to or from it. The following primitive provides that: `tql:link(artifact)`.

Using XPath we can form subsets of links. For example, to find all the links that are of the type *satisfy* the following query can be used:

```
tql:link(to, from)[@type='satisfy']
```

It is also possible to encode this subset of links as an extension function, e.g., `tql:satisfy(to, from)` as another primitive.

In order to tie the artifacts and links together an additional extension function is used to unite the two: `tql:artifact(links)`. This returns all the artifacts that are involved in a given set of links.

The following provides all traceability links: `tql:link(Artifact())`. We can further use this to find all artifacts in a system that have no traceability:

```
set:difference(Artifact(),
  tql:artifacts(tql:link(Artifact())))
```

We use the EXSLT extension function `set:difference` on the entire set of artifacts, and the artifacts involved in links.

By default, the primitives `traceTo` and `traceFrom` use all the links. A more specialized form of these primitives allow for the consideration of only a subset of the links:

```
tql:traceTo(from, to, links)
tql:traceFrom(to, from, links)
```

With this form we can ask questions such as: Which links from design to code were automatically generated?

```
tql:traceTo(
      Design(),
      Code(),
      tql:links(
         Artifacts()[@type='autogen'])
```

Primitives, that we have not fully realized, such as satisfies are specialized versions of the traceTo/From functions. These require additional information and link semantics to be fully supported.

4.3 Example Queries

We now present a number of traceability questions and the corresponding query in TQL. These illustrate the language through a number of high-level traceability queries.

Question: Are all requirements covered by a test case?

```
set:difference(
        Requirement(),
        traceFrom(Requirement(),
                  TestCase() )  )
```

Question: Are all non-functional requirements addressed by one or more parts of the implementation?

The query to find this is based on finding the non-functional requirements that traces to code. We use an extension function NFR() to select the subset of requirements that are non-functional. This question has three answers depending on the path taken through the traceability graph.

The first approach is to find the non-functional requirements that trace through code through any path.

```
tql:traceTo(
   Requirement()[NFR()],
   Code())
```

The second approach is to find the non-functional requirements that trace through the design first, then to the code.

```
tql:traceTo(
   Requirement()[NFR()],
   Tql:traceTo(Design()
               Code()) )
```

The third approach is to find the non-functional requirements that trace through the testcases first, then to the code.

```
tql:traceTo(
   Requirement()[NFR()],
   Tql:traceTo(TestCase()
               Code()) )
```

Once this subset of requirements is found, it can be compared to the entire set of requirements to find out what is missing. The different results reflect the different paths, i.e., set of links, considered. The correct set depends on the specifics and context of the traceability question being asked by the user.

Question: Which parts of code covered by a requirement have documented pre and post condition?

```
tql:traceFrom(
  Requirement(R),
  Code()//src:function
        [@requires | @ensures])
```

In this case, the indication of pre or post conditions is given by additional attributes on a function in the srcML representation. These attributes could be added by additional processing on the srcML, or replaced with the exact srcML XPath needed to detect the pattern of pre and post conditions.

Question: Are any test cases missing?

```
set:difference(
    Requirement(),
    tql:traceTo(
          Requirement(),
          TestCase()))
```

Question: If any given test case fails, which requirements might be impacted?

```
tql:traceTo(
  Requirement(),
  tql:traceTo(Requirement(),
              TestCase(X)) ))
```

Question: What is the impact of changing a requirement on the safety of the system?

We define a new function `tql:impactAnalysis`. The query is designed to first trace from a specific requirement *R* to the code, and then secondly to analyze the code, associated test cases, and related maintenance history to predict the difficulty of the change [7]. The TQL query is formulated as:

```
tql:impactAnalysis(
  tql:traceFrom(
    Code(),
    Tql:traceFrom(Design(),
                  Requirement(R))))
```

Note that the impact analysis function itself incorporates a number of underlying traceability queries that retrieve test case results and maintenance logs, but also which include other lower level services such as code complexity analyzers that may help predict the impact of a change at least at a coarse grained level.

5 Related Work

A constraint based approach to traceability is taken by Reiss [11] and Nentwich et al. [6, 10]. Reiss [11] views software artifacts as imposing constraints on each other. Meta-constraints are rules that are defined on the set of artifacts. SQL queries are then generated for constraints based on meta-constraints.

Nentwich et al. present xlinkit [10] that manages consistency of distributed and heterogeneous documents. It is based on XLink and is responsible for generating consistency links and optionally suggests repair actions. A constraint language, CLIX- based on first order logic and XPath, expresses relationships between elements in multiple documents. An application of xlinkit to check consistencies between UML models is given in [6]. We have not seen xlinkit been applied to evolution of links and believe this is due to its generality. They use XMI and JavaML as models of design and code respectively. Creating paths from XMI is tedious due to the nature of its representation and it does not scale well. Our work is closely related to the above two approaches [10, 11].

The traceability framework presented by Sherba et al. [14] has an evolution service that analyzes the changes to a set of relationships over time. It analyzes existing links in different versions without evolving the links themselves. von Knethen [15] presents a fine-grained conceptual trace model that determines types of artifacts to be traced in order to support impact

analysis and requirements changes. Riebisch [12] introduce feature models that act as an intermediary in structuring traceability links between artifacts. Hypermedia systems such as Chimera [2] also discuss maintaining and versioning links within a software development environment. Ivkovic et al. [8] use model synchronization techniques from model driven architecture to achieve traceability consistency.

Zisman et al. [16] build consistency rules between distributed documents using XPointer and present a set of consistency rules. Alves-Foss et al. [1] describe traceability between design (XMI) and code (JavaML) using an XLink link base. The links are rendered in HTML using XSL and can be traversed via hyperlinks. Conte de Leon et al. [5] describe TraceML, a XLink based language to specify relationships between artifacts.

6 Discussion and Future Work

We have demonstrated the usefulness of TQL (Trace Query Language) via a number of example questions relating to traceability. By using XML as the underlying representation one can easily leverage tools such as XPath to implement the query language.

With regards to performance, our experience has shown that this approach is feasible. We are currently executing complicated pattern matching using XPath on srcML and getting speeds of over 20 KLOC/sec. As such, supporting systems with less than 10,000 links is well within reason.

We are extending this prototype of TQL into a fully working language and tool that will support queries across a number of artifacts we are using.

This work is funded in part by the National Science Foundation under NSF grant CCF 08-11-21.

7 References

[1] Alves-Foss, J., deLeon, D. C., and Oman, P., "Experiments in the Use of XML to Enhance Traceability between Object-Oriented Design Specifications and Source Code", in Proceedings of 35th Annual Hawaii International Conf. on System Sciences (HICSS'02), 2002, pp. 276-284.

[2] Anderson, M. K., Taylor, N. R., and Whitehead, J. E., "Chimera: Hypermedia for Heterogeneous Software Development Environments", ACM Transactions on Information Systems, vol. 18, no. 3, July 2000, pp. 211-245.

[3] Cleland-Huang, J., "Toward Improved Traceability of Non-Functional Requirements", in Proceedings of 3rd ACM International Workshop on Traceability in Emerging Forms Of Software Engineering, Long Beach, California, USA, Nov 8th 2005, pp. 14-19.

[4] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[5] DeLeon, D. C. and Alves-Foss, J., "Experiments on Processing and Linking Semantically Augmented Requirements Specifications", in Proceedings of 37th Hawaii International Conference on System Sciences (HICSS'04), 2004.

[6] Gryce, C., Finkelstein, A., and Nentwich, C., "Lightweight Checking for UML Based Software Development", in Proceedings of 2002 Workshop on Consistency Problems in UML-Based Software Development, 2002, pp. 124-132.

[7] Hayes, J. H. and Zhao, L., "Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems", in Proceedings of IEEE International Conference on Software Maintenance (ICSM) Budapest, Hungary, Sept 2005.

[8] Ivkovic, I. and Kontogiannis, K., "Tracing Evolution Changes of Software Artifacts through Model Synchronization", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Sept 2004, pp. 252-261.

[9] Maletic, J. I., Collard, M. L., and Simoes, B., "An XML-Based Approach to Support the Evolution of Model-to-Model Traceability Links", in Proceedings of 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05), Long Beach, CA USA, Nov. 8th 2005, pp. 67-72.

[10] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A., "xlinkit: A Consistency Checking and Smart Link Generation Service", in Proceedings of ACM Transactions on Internet Technology, 2002, pp. 151-185.

[11] Reiss, P. S., "Incremental Maintenance of Software Artifacts", IEEE Transactions on Software Engineering, vol. 32, no. 9, 2006, pp. 682-697.

[12] Riebisch, M., "Supporting Evolutionary Development by Feature Models and Traceability Links", in Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), Brno, Czech Republic, 2004, pp. 370-377.

[13] Sharif, B. and Maletic, J. I., "Using Fine-Grained Differencing to Evolve Traceability Links", in Proceedings of 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07), Lexington, KY, USA, March 22-23 2007, pp. 76-81.

[14] Sherba, S., Anderson, A., and Faisal, M., "A Framework for Mapping Traceability Relationships", in Proceedings of 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03), October 2003.

[15] von Knethen, A., "Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems", in Proceedings of International Conference on Software Maintenance (ICSM 02), 2002.

[16] Zisman, A., Emmerich, W., and Finkelstein, A., "Using XML to Build Consistency Rules for Distributed Specifications", in Proceedings of Workshop on Software Specifications & Design (IWSSD), 2000, pp. 141-148.