A Visual Traceability Modeling Language

Patrick Mäder and Jane Cleland-Huang

DePaul University, Chicago, IL, USA
patrick.maeder@tu-ilmenau.de, jhuang@cs.depaul.edu

Abstract. Software traceability is effort intensive and must be applied strategically in order to maximize its benefits and justify its costs. Unfortunately, development tools provide only limited support for traceability, and as a result users often construct trace queries using generic query languages which require intensive knowledge of the data-structures in which artifacts are stored. In this paper, we propose a usage-centered traceability process that utilizes UML class diagrams to define traceability strategies for a project and then visually represents trace queries as constraints upon subsets of the model. The Visual Trace Modeling Language (VTML) allows users to model queries while hiding the underlying technical details and data structures. The approach has been demonstrated through a prototype system and and evaluated through a preliminary experiment to evaluate the expressiveness and readability of VTML in comparison to generic SQL queries.

1 Introduction

Software and systems level traceability is a well-known concept, supporting a number of software engineering tasks such as impact analysis, requirements validation, and coverage analysis. However, studies suggest that developers and other project stakeholders often create traceability links only because they are required to by external regulations or by process improvement initiatives. Although the required link creation process serves a useful purpose for helping to validate that the system being constructed meets its requirements, studies have shown that stakeholders rarely re-use traceability links during the long-term use and maintenance of the system [8, 1, 5]. This failure can be partially attributed to the fact that current tools make it difficult for project stakeholders to construct non-trivial, yet useful traceability queries.

In contrast to the recent research focus on decreasing the costs of trace creation, this paper introduces an expressive Visual Trace Modelling Language (VTML) designed to increase the benefits of tracing, through making it more accessible to software developers and other project stakeholders. This follows the approach taken in database research and practice to develop visual query methods that allow users to formulate database queries in a relatively simple and intuitive way [13]. Instead of creating an entirely new notation, our approach utilizes standard UML class diagrams to model trace queries as a set of constraints enforced onto a subset of a traceability meta-model. Taking this more conservative approach means that VTML can be adopted by any organization familiar

with UML, and also that queries can be modeled and executed using standard tools available on most projects. VTML is implemented using a goal-oriented approach which enables project stakeholders to clearly define their traceability needs for the project, develop an associated strategy for capturing the necessary traceability links, and model complex traceability queries in a relatively intuitive way.

The remainder of the paper is structured as follows, Section 2 provides a brief overview of the relevant traceability features included in common development and requirements management tools. Section 3 reviews related work on modeling traceability queries. Section 4 describes an usage-centered traceability process and how traceability queries contribute to it. Section 5 discusses our visual traceability query language and its main concepts. Section 6 shows sample queries, and discusses the application of visual traceability queries, and their definition and validation. Section 7 then discusses an experiment to evaluate the ease of use and understandability of our modelling language.

2 State of Practice in Trace Query Modeling

Almost all leading requirements management tools provide support for common traceability tasks such as coverage and impact analysis based on traceability links created by the user. However this trace functionality is quite rudimentary. Coverage analysis is typically achieved through filtering out unrelated elements within a structural component of the model, while impact analysis is achieved through showing elements related through established traceability links. For example, IBM Rational RequisitePro/Systems Developer™ provides a feature called a Traceability Query, which allows users to create a diagram of all elements dependent upon a selected one or all elements on which a selected element depends. IBM DOORS™ provides a feature that visualizes chains of links across multiple types of artifacts. Similarly, Sparx Enterprise Architect™ provides a feature for generating implementation reports based on user created traces of a specific, pre-defined type.

In most projects, support for more complex traceability queries is provided through a tool-specific API or by direct access to the underlying data structures. For example, Enterprise Architect allows user-defined queries to be modeled as SQL statements on the underlying database, but these queries require substantial knowledge of the tool's internal data structures or of its API. This type of approach does not make it easy for users to develop and use trace queries as an integral day-to-day component of their work.

3 Related Work

To address these limitations, several researchers have developed languages and notations for supporting trace queries, or of adopting standard query languages such as SQL or XQuery. One goal of any such query language is to allow users to specify their queries at an abstraction level that focuses on the purpose of

the trace, as opposed to its underlying data representation. However, there are several specific challenges that make trace queries difficult to handle. Among other issues, traceable artifacts such as requirements, design, code, and test cases, are often represented in heterogeneous formats with different underlying data structures. Although ideally in the future the use of integrated case tools might lead to more standard representations, current traceability solutions must deal with an enormously broad representation of data types and formats.

Maletic and Collard [6] describe a Trace Query Language (TQL) which can be used to model trace queries for artifacts represented in XML format. TQL specifies queries on the abstraction level of artifacts and links and hides low-level details of the underlying XPath query language through the use of extension functions. Nevertheless, TQL queries are non-trivial for users without knowledge of XPath and XML to understand. Zhang et al. [12] describe an approach for the automated generation of traceability relations between source code and documentation. The authors use ontologies to create query-ready abstract representations of both models. The Racer query language (nRQL) is then used to retrieve traces; however nRQL's syntax requires users to have a relatively strong mathematical background.

Wieringa [11] discusses the use of Entity Relationship Models (ERM) to represent traceability links. He points out that "... an ER model of links can be implemented using any database technology" meaning that ad hoc queries can be easily constructed. As ERMs are now often represented as class diagrams, VTML extends this notion by utilizing class diagrams to visualize both the structure of the traceability information and the queries built upon it. Schwarz et al. [9] utilize a meta-model referred to as the Requirements Reference Model (RRM) to store artifacts and relations, and then issue queries using the Graph Repository Query Language (GReQL). The authors show two sample queries with syntax similar to SQL, but provide no further detail concerning the implementation of their approach nor its validation. Nevertheless, their use of a defined meta-model for representing the underlying data is very useful.

Sherba et al. [10] discuss the specification of a traceability system, called TraceM, based on information integration and open hypermedia. This work provides an interesting foundation for VTML, as it describes an optimal basis for our approach. The authors address the problem of heterogeneous artifact representations through proposing a service-based architecture with translators that normalize the heterogeneous data, and schedulers that allow the user to define when to update the normalized data. Among these services is also a query service that "allows filtering of relationships so that different views of the information space can be created based on the needs of various stakeholders." In related work, Lin et al. [3] implemented Poirot, a service-oriented approach for retrieving artifacts dynamically at runtime from a variety of requirements management tools such as RequisitePro™ and DOORS™. Poirot retrieves data using adapters that interface with standard APIs provided by each case tool, and then transform the data into Poirot's XML schema. Our query language could be integrated with the query services of either TraceM or Poirot.

4 Defining the Traceability Information

VTML assumes the presence of an underlying meta-model, that we refer to as the Traceability Information Model (TIM). The TIM provides the context in which VTML queries can be specified and executed [8]. Our approach utilizes a goal-oriented method for identifying long-term strategic trace queries and the underlying data and traceability links needed to support them. This approach minimizes the effort involved in trace creation and maintenance while maximizing its value. The techniques used to identify traceability goals and to construct the TIM are founded in the systematic Goal-Question-Metric (GQM) approach proposed by Basili et al. [2]. There are three steps involved in the process and these are described in the following subsections.

Step 1: Identify tasks that require traceability. In this first step, specific tasks that are dependent upon traceability should be defined. For example, in a safety critical project a safety officer might need to retrieve all requirements that mitigate identified hazards in order to construct a safety case, or a developer might need to check whether the code she/he is editing either directly or indirectly impacts specific quality constraints captured in the software requirements specification. Such questions can be identified systematically through identifying project goals and then analyzing the project roles and their related tasks.

Step 2: Define traceability. Once trace related tasks have been identified, it is necessary to define a project level trace strategy to ensure that the necessary traceability links are created and maintained. Many researchers agree on the necessity of such a project-level definition as it facilitates a consistent and ready-to-analyze set of traceability relations for a project. This definition is commonly called a traceability information model or traceability meta-model and usually represented as a UML class diagram. Figure 1 shows an example of a traceability information model.

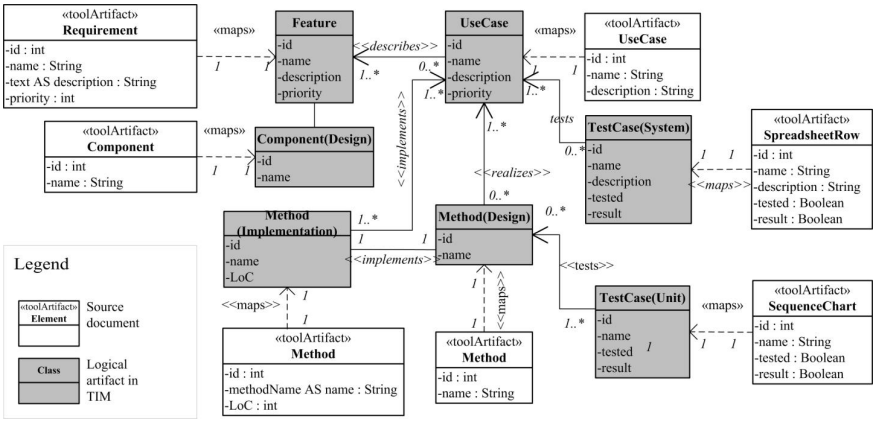


Fig. 1: Example of a project-specific traceability information model

Such an information model is composed of two basic types of entities: traceable artifact types represented as classes, and the permitted traceability relations between the artifact types represented as associations. Traceable artifact types serve as the abstractions supporting the traceability perspective of a project, but they do not necessarily reflect concrete datasets that exist in the traced models. For example, a traceable artifact type might represent an abstraction of several different concrete artifact types existing in the related models, or conversely it could refer to a single artifact type in a tool. Figure 1 also shows the mapping of traceable artifact types to their source documents, each one stereotyped as a 'toolArtifact'. There are several reasons for distinguishing between tool artifact types and abstract traceable artifact types; however the pertinent issue here is that a tool artifact provides information about how a certain traceable artifact type is represented within a concrete tool or model. A more concrete discussion of the traceability information model is given in [4]. As trace creation and maintenance can be expensive, each proposed trace should be evaluated to ensure that it serves a useful purpose. It is also useful to define important properties for each of the traceable artifacts. For example, in Figure 1 the 'UseCase' artifact type includes 'id', 'name', and 'description' properties, all of which can be returned as trace query results or used to define constraints that filter out unwanted artifacts.

Step 3: Define traceability queries. Once traceability tasks have been identified (Step 1) and the TIM established (Step 2), it is necessary to define a set of trace queries that provide an efficient way of supporting the defined tasks. This step is largely ignored by current tools, which assume that trace queries will either be overly simple or that high-end users will export data and write customized scripts to support their more advanced trace queries.

As the TIM provides a graphical representation of logical dependencies between artifacts in a development project, it is natural to use it to specify traceability queries too. There are several benefits to this approach. First traceability queries can be constrained to act on the traceable artifacts and traceability relations defined in the TIM, with the underlying assumptions that associated data capture is integrated into the software development and management process. Second, visualizing trace queries in this way can make them more intuitive for typical project stakeholders. This conjecture is tested through the experiment described in Section 7 of this paper.

There are several well-established query languages such as SQL and XQuery which can provide the same results for a specific dataset as the method we propose in this paper; however there are two specific issues that we believe justify using VTML:

– Traceability queries deal to a large extent with the existence of relations between artifacts and with the count of those relations, although such queries can be specified in standard query languages such as SQL, they lead to rather complex, recurring constructs. For example, a simple query against the TIM in Figure 1, designed to identify implemented methods related to a given set of use cases, translates into the following SQL statement:

```
SELECT "UseCase".id, "Method(Implementation)".id
```

```
FROM "Method(Implementation)","LINKS_Method(Implementation)_Method(Design)",
"Method(Design)", "LINKS_Method(Design)_UseCase", "UseCase"
WHERE
"Method(Implementation)".id ="LINKS_Method(Implementation)_Method(Design)".sourceID AND
"LINKS_Method(Implementation)_Method(Design)".targetID = "Method(Design)".id
AND "Method(Design)".id = "LINKS_Method(Design)_UseCase".sourceID AND
"LINKS_Method(Design)_UseCase".targetID = "UseCase".id AND "UseCase".id
```

– A large part of a traceability query specified in a standard language refers to the underlying data structure. For traceability purposes this has already been described in the TIM, and redefining it in each trace query introduces unwanted redundancy.

By re-using information previously specified in the TIM, VTML hides most of the technical details and creates queries at the traceability perspective of a project.

5 Defining Visual Traceability Queries

This section describes the way in which VTML queries are modeled over the TIM. The discussion is separated into a specification of the general structure of a query, a specification of constraints on a query and finally the inclusion of aggregation functions as part of a query.

5.1 Query Structure

Class diagrams provide a convenient way of representing a query, which can be modeled as a structural subset of the traceability information model. This means that a query may be composed from all traceable artifact types across all permitted traceability relations defined within the current traceability information model of a project. This approach also has the significant benefit of utilizing a widely adopted modeling language, with all its associated tool support.

In addition to modeling traceable artifact types and their relationships, the TIM also associates a set of properties with each traceable artifact. These properties, which are defined as attributes for each artifact type, can be used to specify query constraints and can also be returned as results of a trace query. When these properties are used within a query, they are stereotyped to show whether they represent a 'result' or a 'filter constraint'. As depicted in Figure 2, each stereotype is associated with a graphical symbol placed in front of the property name. For example, attributes stereotyped as 'results' are represented by a bar graph symbol, while attributes used to filter the results are annotated with a filter symbol. Most UML modeling tools support the use of graphical symbols in place of stereotypes. An identifier property exists by default for each traceable artifact type and is used to join the underlying data structures (artifacts and traces) automatically. This property is only shown within a query if it is intended to be returned in the result set.

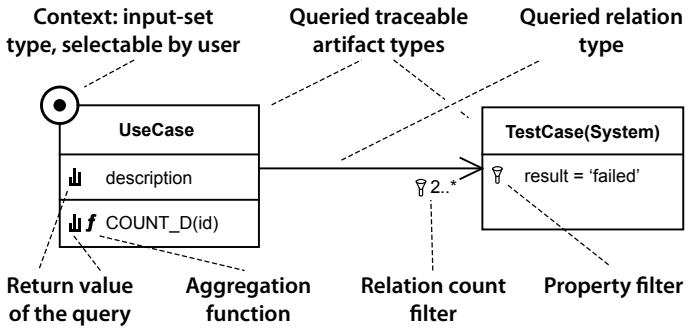


Fig. 2: Features of a visual traceability query

5.2 Defining Constraints

While structural elements support queries across traceable artifacts, more specific queries can only be obtained by specifying constraints. There are three kinds of constraints that can be specified in our notation: constraints to properties of traceable artifacts, constraints to the number of existing traceability relations between artifacts, and constraints on the scope which defines the user-selected input set of a query.

The first type of constraint refers to the properties of traceable artifacts. As previously discussed, these properties can become part of a query's result or may be used to filter out unwanted artifacts. In VTML the constraint is defined after the name of a property attribute within a traceable artifact type. A stereotype 'filter' is attached to the attribute and visually represented as the filter symbol (see Figure 2). The constraint is specified as a logical expression consisting of the property name, a logical comparison ($=$, $<$, $>$, $<=$, $>=$, $!=$) and a value or several values as boolean expression (&&, ||, !).

Multiplicity constraints refer to the number of existing traceability relations between two artifacts. By specifying multiplicities for a traceability relation between two traceable artifact types, it is possible to constrain the query results to only those artifacts that provide the specified number of traceability relations. Similar to property constraints, a stereotype 'filter' is attached to the multiplicity and visually represented as a symbol (see Figure 2). As standardized in UML class diagrams, multiplicities can be defined as a single number, a list of numbers, or as a range of numbers, and therefore provide significant flexibility in specifying constraints with respect to the number of existing traceability relations. For the current prototype implementation we decided to interpret an unspecified multiplicity as $1..*$. Multiplicity constraints facilitate a wide variety of trace queries, for example, to retrieve all requirements with no (zero) related acceptance tests, or conversely all requirements that fan-out to 2 or more design elements.

The final type of constraint refers to a so-called query scope which defines the traceable artifact type that a query can be applied to. While executing a query, the user may choose to perform the query on all artifacts of that type within a model or to constrain it to a subset of those. The scope is defined by attaching the stereotype 'scope' to one of the traceable artifact types of a query and is visually represented as an encircled dot (see Figure 2). The example in Figure 2 means that the query is applicable to use cases and the user may decide to provide a specific input-set of use cases to be queried or to perform the query on all use cases.

In order to increase readability of queries we apply directed associations starting from the scope element. Although, this is not required for the automated interpretation of the query by a tool, feedback from early user studies has shown that it can increase readability for human users.

5.3 Aggregation Functions

Some trace queries may require aggregation of the query results. For example, instead of requesting a list of concrete artifacts which fulfill a certain query, a user might require the trace query to return their count. For this purpose standard query languages provide a set of aggregation functions. VTML currently supports the same functions as SQL. These functions are defined as methods within traceable artifact types and a stereotype 'function', visually represented as a f symbol, is attached (see Figure 2).

5.4 Integrating Other Techniques

In addition to standard aggregation functions VTML supports an extended set of customized functions, implemented as code snippets. For example, a function could be developed to aggregate code metrics for all classes or methods that traced from a specified requirement. For evaluation purposes we developed two such functions and successfully incorporated them in the VTML and executed them as trace queries.

5.5 Limitations and Analysis of the Approach

It is important to observe that all defined constraints of a query apply in parallel. That means that for each artifact within the user-selected scope all defined constraints must be fulfilled in order to be part of the results. We found that limitation acceptable as we were able to express a broad range of desired queries during the development of VTML. However, the notation does not support some specific types of queries, for example, artifacts that either have a certain property value or a relation to another artifact. Such queries need currently to be performed separately, concatenating the results as an additional step. We could not find an appropriate visual way of representing those dependencies between constraints, while keeping the simplicity of the visual notation. We are currently

evaluating the use of filter references that can be used to write complex boolean expression involving all filters as an additional text.

Moody [7] describes nine principles for designing cognitively effective visual notations against which we qualitatively validated our VTML approach. As advocated by Moody, our notation provides a 1:1 mapping between semantic constructs that we are aiming to express and the graphical symbols used to represent them (semiotic clarity). All our symbols are clearly distinguishable from each other (perceptual discriminability). The participants of our experiment reported no problems in identifying the meaning of our symbols (semantic transparency). Visual traceability queries show only the actual queried part of the available traceability information (complexity management). The representation of our queries builds upon the representation of the traceability information model (cognitive integration). We apply a cognitively manageable number of visual symbols (graphic economy) where appropriate and use text to complement these graphics (dual coding). Finally, we assumed that the traceability information model is represented as a UML class diagram and created our notation accordingly; however if the traceability information model were to be represented using a different notation the VTML notation should be updated accordingly (cognitive fit).

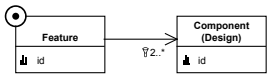
6 Applying Visual Traceability Queries

This section provides examples of visual traceability queries and discusses different aspects of their application. Although we only depict a small sampling of queries in this paper, we have used VTML to express a much wider variety of useful trace queries in a mid-sized industrial project.

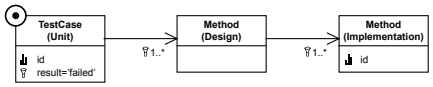
6.1 Example Queries

Figure 3 shows four query examples that demonstrate VTML's ability to express a variety of traceability queries, including ones that could not easily be modeled in existing requirements management tools.

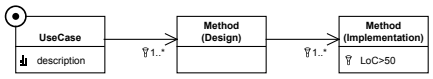
The query shown in Figure 3a finds features that are implemented by more than one component of the design model and so highlights possible deficiencies in the design. Figure 3b depicts a query that returns all methods implementing a 'failed' unit test case and so facilitates the analysis of the discovered problem in the source code. The query in Figure 3c returns the description of all use cases that are implemented by methods with more than 50 lines of code. The purpose of such a query could be to identify and review complex usage scenarios. Figure 3d shows a query, inspired by a real world example, that finds redundant traceability relations between use cases and implementation methods. While the traceability information model allows both routes, the idea is that either the one or the other should be chosen by the user in order to avoid conflicts during other analyses. Both routes could be allowed, because only some of the use cases are documented in the design model and can be traced via such artifacts.



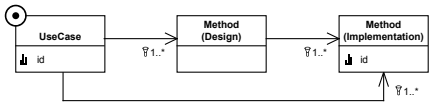
(a) Query: Find features scattered among multiple components



(b) Query: Find methods implementing 'failed' unit test cases



(c) Query: Show uses cases implemented by methods with $> 50 LoC$



(d) Query: Find redundant traces between use cases and implementation

Fig. 3: Example queries

6.2 Transformation Into Executable Queries

One of the major benefits of VTML is that trace queries are specified over the TIM, and do not need to reference the underlying data structures. This means that a user specifies and reads queries from the traceability perspective of a project. However, in order to execute these queries it is necessary to transform them into a query format that is supported by the actual data sources. Although VTML is not bound to any specific underlying query language, we demonstrate its feasibility through a transformation of visual traceability queries into SQL queries executable on the traceability repository of our *traceMaintainer* prototype. The transformation is fully automated and converts the features of a visual traceability query step by step into an executable SQL query.

The VTML transformation is implemented using a XSLT script that translates queries in XMI format, exported from a compatible UML modeling tool, into SQL statements executable on *traceMaintainer's* database. The transformation is not only dependent upon the target query language, but also on the structure of the repository. For the prototype implementation we decided to store each traceable artifact type, defined with the traceability information model, as a separate table as well as each traceability relation defined among these types. This is one possible way of implementing the data structure, but not the only one. The rationale behind our implementation decision was that different traceable artifact types as well as different defined traceability relations might have varying numbers and types of properties making it more difficult to store all in the same table.

6.3 Supporting the Creation and Validation of Queries

In order to execute the defined queries, our current prototype requires the user to export the created queries into XMI format, which is supported by all major modeling tools. Future iterations of our prototype tool could include a VTML

wizard to provide interactive guidance on how to create queries for certain common purposes (e.g., counting elements over several artifact levels). Furthermore, as all queries are subsets of the traceability information model, the TIM can be used to validate the structural correctness of each query. Additionally, defined constraints can be validated for their syntactical correctness by using regular expressions. While extensions to the traceability information model will have no effect on defined queries, deletions and modifications could invalidate a query if the required information becomes unavailable following the change. Our tool revalidates queries each time they are transformed into the executable format.

7 Evaluation

We designed a preliminary experiment to comparatively evaluate the understandability and the ease of use of VTML with respect to other query languages. However, the experiment reported in this paper, was limited to a comparison with SQL, which represents an expressive and broadly adopted query language used in industry. We formulated two research questions:

Q1 Reading: Does the use of Visual Traceability Queries result in a more accurate and faster understanding of a query's purpose compared to equivalent techniques?
Q2 Constructing: Does the use of Visual Traceability Queries result in a more accurate and faster construction of traceability queries compared to equivalent techniques?

Our experiment had one independent variable, the query notation, and two treatments: VTML and SQL. Our experiment aimed to find out whether there is a causal relationship between the treatment and the time and correctness for reading and constructing queries.

7.1 Experimental Set-up

In order to answer these two research questions, we designed a controlled experiment which included trace queries we had previously seen executed in actual industrial projects.

Subjects The subjects comprised 18 practitioners and students with a basic knowledge of UML modeling and database engineering as well as writing and understanding SQL queries. Our participants had an average experience of 3.5 years in using SQL queries but only an average of 2.2 years with UML. This indicates that for many of the subjects we were evaluating a well-known technique against a relatively new approach.

Procedure and Tasks All the data was gathered via questionnaires. In addition to providing actual answers to the questions in the questionnaire, the time it took to complete individual tasks was recorded. The experiment consisted of the following steps:

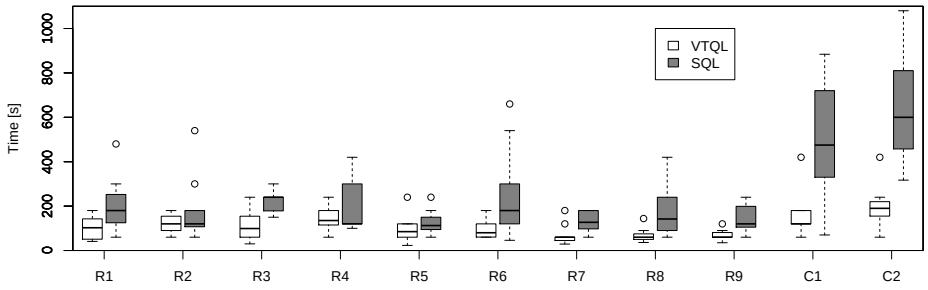


Fig. 4: Time required to understand (R1–R9) and construct queries (C1, C2)

1. All subjects completed a series of questions to describe her/his background and experience in the field of software and data engineering.
2. All subjects read a tutorial about the general purpose of software traceability, the use of a traceability information model, and the purpose of traceability related queries. The material also contained a table comparing features of a query expressed in SQL and VTML. The subjects were allowed to use the tutorial material throughout the entire experiment.
3. All participants were given a set of nine different queries, each expressed in either SQL or VTML. For each query we provided four possible answers, and the participants were directed to select the answer which they felt most closely represented the meaning of the query. Each query was presented to 9 participants in SQL and 9 in VTML.
4. All subjects were also asked to construct two queries, one written in SQL and one modeled in VTML. The assignment was random.
5. All subjects completed a questionnaire concerning their experience using both VTML and SQL to read and construct traceability queries.

7.2 Results

Q1 Reading queries Table 1 shows that subjects viewing our visual notation responded on average (mean) 26% to 63% faster to the nine questions (R1–R9) than subjects viewing the same query in SQL notation, thereby reducing the time to understand a query by 45%. However the difference was statistically significant in only six of the nine queries (see p-values in column t-test) due to the high variability in the response time. Figure 4 visualizes response time and variability across all tasks. The variability could have been caused by different experience levels of the subjects; however this will be analyzed in a future experiment. Post experiment interviews also suggested that the multiple choice design allowed users to guess the answer without fully understanding the query, which certainly could have impacted the results of the query reading task. For reading visual queries, 14.9% of the given answers were incorrect using the visual notation, while only 10.5% were incorrect using SQL. However, half of the incorrect answers

Table 1: Time differences [s] for performing tasks

	VTML		SQL		diff	
task	mean	sd	mean	sd	VTML	t-test
R1	101.6	53.6	206.9	134.0	-51%	0.03
R2	121.1	45.5	178.2	153.5	-32%	0.16
R3	112.1	70.4	220.9	48.3	-49%	0.00
R4	145.0	55.0	210.3	123.2	-31%	0.09
R5	94.1	63.4	126.8	57.1	-26%	0.14
R6	95.1	42.3	257.9	208.3	-63%	0.02
R7	71.7	48.2	131.1	45.7	-45%	0.01
R8	68.8	34.5	171.2	113.3	-60%	0.01
R9	68.3	25.2	143.5	62.1	-52%	0.00
Ø					**-45%**	
C1	153.0	100.4	500.5	276.6	-69%	0.00
C2	202.9	112.7	647.4	271.0	-69%	0.00
Ø					**-69%**	

in both notations referred to the same query, suggesting that the answers we provided might have been misleading. Furthermore, two-thirds of the incorrect answers for visual queries were given for the first three questions, suggesting that comprehension of visual queries increased with experience.

Q2 Constructing queries Table 1 shows that subjects constructed the same query in our visual notation on average 69% faster than in SQL. Despite the relatively large variability, especially in the time spent constructing SQL queries (see Figure 4), the differences for both construction tasks (C1, C2) are statistically significant. Again, this is likely due to differences in experience of our subjects. Only 6.7% of the constructed visual queries (one query) were partly incorrect, while 89.5% of the constructed SQL queries were at least partly incorrect. This suggests that our approach facilitates a significantly faster and more correct specification of traceability queries than SQL.

7.3 Threats to Validity

Important threats to the validity of the experiment are divided into four common categories.

External Validity Our experiment shows results of subjects with a diverse background in the field of our experiment, from practitioners to students, with practical experience, for example, as product managers, developers, requirements engineers and designers. Nevertheless, the relatively small size of our sample does not allow us to draw general conclusions, we rather see our experiment as an initial validation which will now lead into an extended study. All of the presented queries had a realistic purpose and were determined based on our knowledge of traceability in industrial settings.

Internal Validity To decrease variability in knowledge across participants we provided an introductory tutorial. The written form of the material minimized

the possible influence of the experimenters on the results. The notation in which a query was represented was randomly assigned in order to balance learning effects. None of the participants provided more than two incorrect answers suggesting a sound understanding of the topic. Although, we improved the multiple-choice answers for the questions during pilot tests, some of the answers might still have been misleading as previously discussed.

Reliability We expect that replications of the experiment will offer results similar to those presented here. Concrete measured results will differ from those presented here as they are specific to the subjects, but the underlying trends and implications should remain unchanged. Our participants had a large variety of experience regarding the topic of the experiment.

Construct Validity Our experiment aimed at evaluating the understandability and the ease of use of our visual notation compared to existing techniques. We decided to focus on reading and constructing of traceability queries as we believe that those are the most important applications for a visual traceability modeling language. If a notation is easier to use and comprehend, then the measures of time and correctness should correspondingly show lower values. Our experiments therefore focused on these measures.

8 Conclusions and Future Work

This paper has presented a usage-centered traceability process that first defines the traceability strategies for a project and then models traceability queries visually using VTML. It introduces a novel way to specify traceability queries that utilizes the project's TIM and builds on UML concepts that are well known to most users. In this way users apply the same technique to describe and execute traceability queries as they use for modeling the overall project artifacts. Furthermore, the specification of queries is constrained to entities defined within the TIM, facilitating a consistent traceability view of a project as well as limiting possible choices in the specification of queries to the actual available ones.

The experiment we performed has demonstrated that users are able to read and construct traceability queries more quickly using VTML. This was especially marked following an initial learning curve. This curve appeared most evident for users with less prior UML experience. Our experiment further suggests that visually constructed traceability queries are substantially more correct compared to the same queries constructed with SQL. As a proof of concept and to gain more experience we developed a prototype implementation. Future work will involve augmenting the prototype to include more advanced features to guide the user through the task of creating and validating trace queries. Furthermore, although our current prototype uses XSLT to transform visual queries into executable ones, we are exploring more general transformations that can be customized to different underlying data schemes and various query languages such as SQL, XQuery, and LINQ. Finally, we intend to conduct a more comprehensive study that evaluates whether VTML can by used by stakeholders to create traceability links that help them perform useful tasks in an industrial settings.

Acknowledgments

This work was partially funded by the National Science Foundation grant #CCF: 0810924.

References

1. Arkley, P., Riddle, S.: Overcoming the traceability benefit problem. In: Proceedings 13th International Requirements Engineering Conference. pp. 385–389. IEEE Computer Society (2005), ISBN 0-7695-2425-7
2. Basili, V.R., Caldiera, G., Rombach, H.D.: Goal Question Metric Paradigm. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1, pp. 528–532. John Wiley & Sons (1994)
3. Lin, J., Lin, C.C., Cleland-Huang, J., Settimi, R., Amaya, J., Bedford, G., Berenbach, B., Khadra, O.B., Duan, C., Zou, X.: Poirot: A distributed tool supporting enterprise-wide automated traceability. In: RE. pp. 356–357. IEEE Computer Society (September 2006)
4. Mäder, P., Gotel, O., Philippow, I.: Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. In: 5th Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2009). In conjunction with ICSE09. Vancouver, Canada (May 2009)
5. Mäder, P., Gotel, O., Philippow, I.: Motivation Matters in the Traceability Trenches. In: Proceedings of 17th International Requirements Engineering Conference (RE'09). Atlanta, Georgia, USA (August 2009)
6. Maletic, J.I., Collard, M.L.: Tql: A query language to support traceability. In: TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering. pp. 16–20. IEEE Computer Society, Washington, DC, USA (2009)
7. Moody, D.L.: The 'physics' of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng 35(6), 756–779 (2009)
8. Ramesh, B., Jarke, M.: Toward reference models of requirements traceability. IEEE Transactions on Software Engineering 27(1), 58–93 (2001)
9. Schwarz, H., Ebert, J., Riediger, V., Winter, A.: Towards querying of traceability information in the context of software evolution. In: 10th Workshop Software Reengineering, 5-7 May 2008, Bad Honnef. LNI, vol. 126, pp. 144–148. GI (2008)
10. Sherba, S.A., Anderson, K.M., Faisal, M.: A framework for mapping traceability relationships. In: Second International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003) (Oct 2003)
11. Wieringa, R.: An introduction to requirements traceability. Tech. Rep. IR-389, Faculty of Mathematics and Computer Science (November 1995)
12. Zhang, Y., Witte, R., Rilling, J., Haarslev, V.: An ontology-based approach for the recovery of traceability links. In: 3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). Genoa, Italy (October 1st 2006)
13. Zloof, M.: Query-by-example: A database language. In: Query-by-Example: A Database Language. pp. 324–343. IBM Systems Journal (1977)