Softw Syst Model
DOI 10.1007/s10270-012-0237-0

SPECIAL SECTION PAPER

A visual language for modeling and executing traceability queries

Patrick Mäder · Jane Cleland-Huang

Received: 16 March 2011 / Revised: 18 January 2012 / Accepted: 26 January 2012

Abstract Current software and systems engineering tools provide only basic trace features, and as a result users are often compelled to construct non-trivial traceability queries using generic query languages such as SQL. In this paper, we present an alternative approach which defines traceability strategies for a project using UML class diagrams and then constructs trace queries as constraints upon subsets of the model. The visual trace modeling language (VTML) allows users to model a broad range of trace queries while hiding underlying technical details and data structures. The viability and expressiveness of VTML for use in a real project are demonstrated through modeling a broadly representative set of queries for a web-based health-care system. It is then evaluated through an experiment with human users to assess the readability and writability of VTML queries in comparison to generic SQL queries. We found that users read and constructed traceability queries considerably faster using VTML than using SQL. Furthermore, visually constructed traceability queries were substantially more correct compared to the same queries constructed with SQL.

Keywords Visual traceability queries · Requirements traceability · Visual query language · Traceability information model (TIM)

Communicated by Prof. Dorina Petriu.

Electronic supplementary material The online version of this article (doi:10.1007/s10270-012-0237-0) contains supplementary material, which is available to authorized users.

P. Mäder (✉) · J. Cleland-Huang
DePaul University, Chicago, IL, USA
e-mail: pmader@cdm.depaul.edu

J. Cleland-Huang
e-mail: jhuang@cs.depaul.edu

1 Introduction

Software traceability is a required component of many software development processes. It provides support for numerous software engineering tasks such as requirements validation, impact analysis, coverage analysis, compliance verification, and derivation analysis. However, despite its potential usefulness, studies have shown 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 [1–3]. 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. As a result, trace users must often resort to writing quite complex trace queries using a general query language such as SQL.

This paper extends our prior work presented at the 2010 International Conference on Model Driven Engineering and Systems (MODELS 2010) [4] which introduced a visual trace modeling language (VTML) designed to alleviate the difficulties of writing trace queries. VTML increases the benefits of traceability through making it more accessible to software developers and other project stakeholders. VTML 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 [5]. 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 metamodel. Taking this more conservative approach means that VTML can be adopted by any organization familiar with

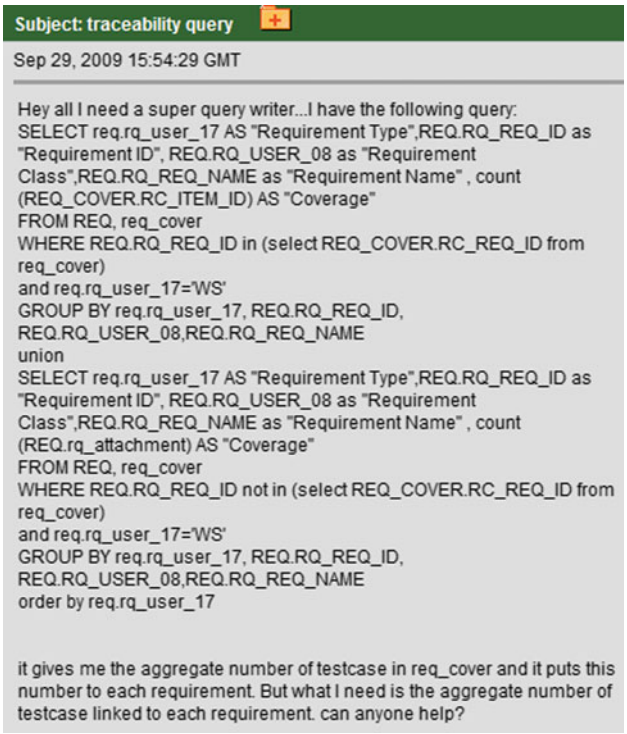
Published online: 08 April 2012

UML, and also that queries can be modeled and executed using standard tools available on most projects. Our approach incorporates a goal-oriented process for clearly defining the stakeholders' traceability needs for the project, developing a strategy for capturing the necessary traceability links, and modeling a set of useful traceability queries.

The remainder of the paper is structured as follows: Sect. 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 a 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 describes a case study we performed to evaluate the expressiveness of VTML within a realistic project. It also reports an experiment conducted to evaluate the readability and writability of our modeling language. Finally, Sect. 8 summarizes our findings and suggests areas for future work.

2 State of practice in trace query modeling

Most leading software and systems development 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 showing all of the dependencies on a selected element or all elements on which a selected element depends. Rational DOORS™ provides a feature that visualizes chains of links across multiple types of artifacts. Sparx Enterprise Architect™ provides a feature for generating implementation reports based on user created traces of a specific, pre-defined type. Similarly, HP Quality Center™, which is a test management software package with support for requirements management and full life-cycle management integration capabilities, generates traceability matrices showing aggregated numbers of test cases covering specific functionality in the development. These representative products illustrate that traceability support and rudimentary traceability analyses are available across software and systems development tools. Unfortunately, practitioners often need advanced traceability capabilities which are not readily accessible in these tools without advanced technical skills.


Subject: traceability query

Sep 29, 2009 15:54:29 GMT

Hey all I need a super query writer...I have the following query:

```
SELECT req.rq_user_17 AS "Requirement Type",REQ.RQ_REQ_ID as
"Requirement ID", REQ.RQ_USER_08 as "Requirement
Class",REQ.RQ_REQ_NAME as "Requirement Name" , count
(REQ_COVER.RC_ITEM_ID) AS "Coverage"
FROM REQ, req_cover
WHERE REQ.RQ_REQ_ID in (select REQ_COVER.RC_REQ_ID from
req_cover)
and req.rq_user_17='WS'
GROUP BY req.rq_user_17, REQ.RQ_REQ_ID,
REQ.RQ_USER_08,REQ.RQ_REQ_NAME
union
SELECT req.rq_user_17 AS "Requirement Type",REQ.RQ_REQ_ID as
"Requirement ID", REQ.RQ_USER_08 as "Requirement
Class",REQ.RQ_REQ_NAME as "Requirement Name" , count
(REQ.rq_attachment) AS "Coverage"
FROM REQ, req_cover
WHERE REQ.RQ_REQ_ID not in (select REQ_COVER.RC_REQ_ID from
req_cover)
and req.rq_user_17='WS'
GROUP BY req.rq_user_17, REQ.RQ_REQ_ID,
REQ.RQ_USER_08,REQ.RQ_REQ_NAME
order by req.rq_user_17
```

it gives me the aggregate number of testcase in req_cover and it puts this number to each requirement. But what I need is the aggregate number of testcase linked to each requirement. can anyone help?

Fig. 1 A post from the Hewlett Packard help forum illustrating the challenges of modeling trace queries using SQL [6]

In most tools, support for more complex traceability queries is provided through a tool-specific API or by direct access to the underlying data structures. For example, tools such as Enterprise Architect and HP Quality Center allow 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. Unfortunately, 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. This problem is illustrated by Fig. 1, which shows a discussion post at Hewlett Packard's Quality Center™ help forum from a person requesting assistance with the logistics of writing a rather convoluted trace query using SQL. Although there are clearly much simpler representations for this query, this example illustrates first of all that trace users are finding the need to write complex trace queries, and secondly that writing queries can be challenging and error prone. Furthermore, the depicted post is not a unique request, the forum contains a variety of similar questions.

3 Related work

To address these limitations, several researchers have developed languages and notations for supporting trace queries, or for utilizing standard query languages such as SQL or XQue-

ry. 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 [7] 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. [8] describe an approach for the automated generation of traceability relations between source code and documentation. They 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.

Guerra et al. [9] focus on inter-modeling, the activity of building models that describe how modeling languages are related. Their technique is able to transform existing traceability data within a model into a different representation (e.g. a table). Though very powerful, the creation of transformation patterns requires a very experienced user.

Graphical query languages have been proposed and commercially offered in the database domain for a long time. The PICASSO approach by Kim et al. [10] was one of the first graphical query languages and has been built on top of the universal relation database system System/U. Visual SQL by Jaakkola et al. [11] translates all features of SQL into a graphical representation similar to entity relationship models (ERM) and UML class diagrams. Furthermore, there are a variety of commercial and open source tools that provide graphical support for the specification of queries (e.g., Microsoft Visual Studio™, Microsoft Access™, Active Query Builder and Visual SQL Builder). While the graphical representation we propose has similarities to these approaches and tools, there are fundamental differences. First, these approaches assume that data are stored in a structured, most likely relational, format. However, a typical software intensive project stores artifacts in a variety of file formats, databases, and proprietary case tools. VTML, therefore, takes a layered approach in which the underlying data are mapped to a logical perspective against which traceability queries are specified. This allows traceability strategies and queries to be reused across different projects. Second, VTML provides a rich set of traceability queries by supporting queries based on simple associations between traceable artifacts as well as a rich set of filters based on attribute types and cardinalities. Throughout this paper, we provide equivalent trace queries in both VTML and SQL to highlight their differences.

Störrle [12] proposes a visual model query language (VMQL). This notation facilitates searching in and querying of models. According to the author, these activities are not well supported by current CASE tools. VMQL uses the same language for formulating queries and presenting results as the models it queries. The intention of VMQL is to match structures in diagrams and it is powerful and intuitive in this regard. However, the notation is not able to express the majority of queries needed to support traceability for two main reasons. First, it does not support queries across different models, such as requirements and code models. This is fundamental to traceability. Second, traceability queries often return multiple target elements related to a single source element. However, VQML does not provide support for returning an unknown number of elements in a single result. Nevertheless, VMQL offers a very interesting approach and provides a new perspective on searching in models. VMQL could in future work be integrated with VTML to create queries that match model element structures in different models, related by traceability.

Wieringa [13] discusses the use of 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. [14] 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 or its validation. Nevertheless, their use of a defined meta-model for representing the underlying data are very useful.

Sherba et al. [15] 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. TraceM addresses the problem of heterogeneous artifact representations through utilizing a service-based architecture with translators to normalize heterogeneous data and to keep it updated. It also provides support for filtering relationships to create different views for various stakeholder groups. In related work, Lin et al. [16] 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

Visual trace modeling language assumes the presence of an underlying meta-model, referred to as the traceability information model (TIM). This meta-model provides the context in which trace queries can be specified and executed [1]. Although there are many different approaches which could be used to specify a TIM, in this section, we describe a goal-oriented process which we have found to be particularly effective for strategically designing the TIM and identifying associated trace queries for a given project. This approach is based on the commonly adopted goal-question-metric (GQM) approach proposed by Basili et al. [17] for systematically identifying software metrics. The concept of software metrics has strong similarities to that of traceability because the determination of a metric may require traceability. Accordingly, we suggest the application of the GQM technique for the definition of a project-specific traceability strategy. When applied to the traceability problem, GQM focuses on strategically identifying software development tasks that would benefit from traceability, and then identifying and specifying a set of supporting traceability queries. This approach decreases the costs, and increases the benefits of the traceability process by ensuring that all traceability links are constructed in a purposeful manner.

There are three general steps involved in the definition of traceability and these are described in the following subsections.

4.1 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 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.

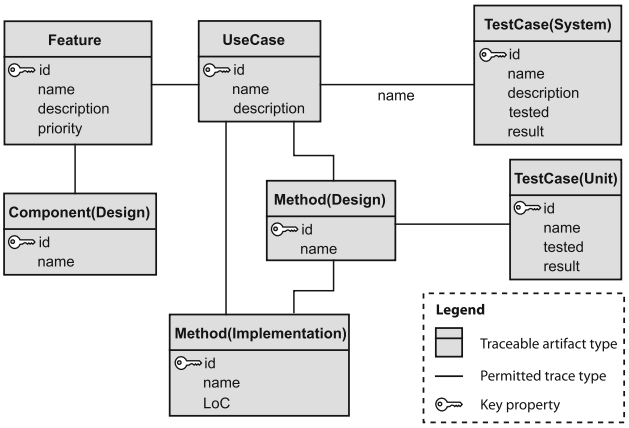


Fig. 2 Example of a project-specific traceability information model

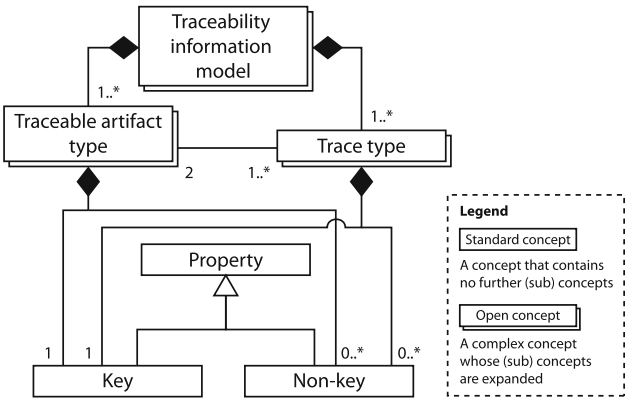


Fig. 3 Meta-model defining a minimal project-specific traceability information model

4.2 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 [18]. Each proposed trace should be evaluated to ensure that it serves a useful purpose. Many researchers agree on the necessity of 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 TIM or traceability meta-model and, as depicted in Fig. 2, is often represented as a UML class diagram.

Figure 3 shows a basic meta-model for defining TIMs. The meta-model shows that an information model is composed of two basic types of entities: traceable artifact types represented as classes, and the permitted trace types between the artifact types represented as associations. Traceable artifact types serve as the abstractions supporting the traceability perspective of a project. It is also useful to define important properties for each of the traceable artifact types and trace types. For example, in Fig. 2, 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. Similarly, for the trace type between 'UseCase' and 'TestCase(System)', a property 'name' has been defined as association label and can be queried as well. Although not explicitly shown in this paper, additional trace type properties would be represented as association classes. A TIM may be augmented so that the traceability paths are augmented with multiplicities, names, and roles. A more extensive discussion of the TIM is provided in a related paper [19].

4.3 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 for 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 three major advantages 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 Fig. 2, designed to identify unit test cases related to a given set of use cases, translates into the SQL statement shown in Fig. 4.
- 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.
- Visual notations can provide highly effective support for communicating and modeling complex ideas [20].

```
SELECT 'UseCase'.id, 'TestCase(Unit)'.id
FROM 'TestCase(Unit)', 'LINKS-TestCase(Unit)-
   Method(Design)', 'Method(Design)',
   'LINKS-Method(Design)-UseCase', 'UseCase'
WHERE
   'TestCase(Unit)'.id = 'LINKS-TestCase(Unit)-
   Method(Design)'.sourceID AND 'LINKS-
   TestCase(Unit)-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 IN (@inputSet)
```

Fig. 4 Example of a SQL query for retrieving implemented methods related to a given set of use cases

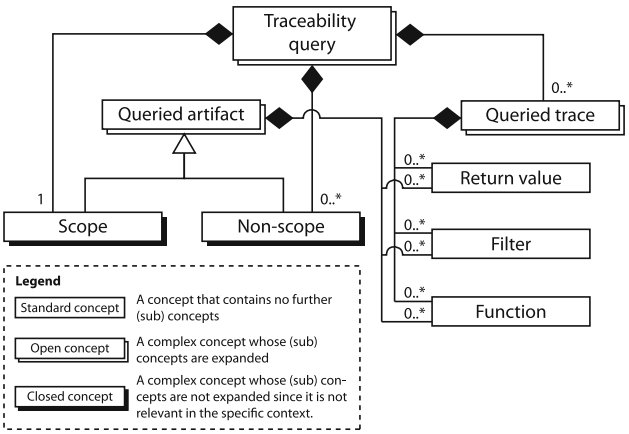


Fig. 5 Meta-model defining the concepts used to create traceability queries and their relations

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. Figure 5 shows a meta-model for defining VTML queries.

5.1 Query structure

Class diagrams provide a convenient way of representing a query, which can be modeled as a structural subset of the TIM. This means that a query may be composed from all traceable artifact types across all permitted traceability relations defined within the project's TIM. This approach also

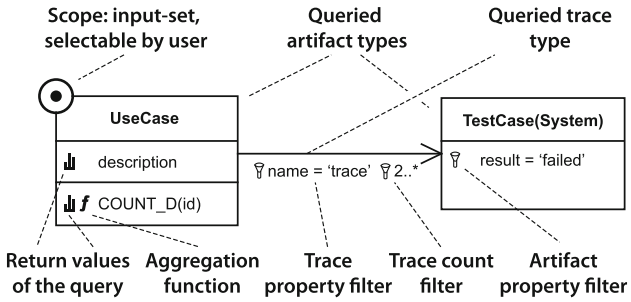


Fig. 6 Features of a visual traceability query. The example query retrieves a list of use case descriptions and a count of their related test cases for all uses cases in the input set that have more than two failed test cases associated by traces named trace

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 filters 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', a 'filter' or both. As depicted in Fig. 6, each stereotype is associated with a graphical symbol placed in front of the property name. For example, properties stereotyped as 'results' are represented by a bar graph symbol, while attributes used as filters 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.

5.2 Filters

While the structural elements form the basis of a query, more specific queries can only be obtained by specifying filters. There are four kinds of filters that are supported by our notation: artifact property filters, trace property filters, trace count filters, and the user selectable scope of a query. The TIM meta-model (see Fig. 3) shows that the traceability perspective of a project deals with two types of entities: traceable artifacts and traces. Both may have properties. Artifact and trace property filters allow selecting entities based on their properties. Trace count filters allow selection based on the relation between traceable artifacts and traces. The scope facilitates integration into a tool. The user can select a single element or a group of elements of the same type and defined queries matching that type can be displayed and executed. The query is constrained to the element(s) selected beforehand. Our aim was to keep VTML as simple as possible to facilitate its application without intensive training and we found these four types of filters sufficient for the definition of traceability queries.

Artifact property filter As previously discussed, artifact properties can become part of a query's result or may be used to filter out unwanted artifacts. In VTML, such a filter is defined after the name of a property within an artifact type. A stereotype 'filter' is attached to the property and visually represented as the filter symbol (see Fig. 6). The constraint is specified as a logical expression consisting of the property name, a logical comparison (=, <, >, <=, >=, !=, LIKE) and a value or several values as boolean expression (&&, ||, !).

Trace property filter Similar to artifact property filters, it is also possible to filter on properties of traceability relations. The notation of these filters follows the same syntax and allows the same logical expressions as described for artifact property filters.

Trace count filter These filters refer to the number of existing traceability relations of a certain type. By specifying a multiplicity for a trace type between two artifact types, it is possible to constrain the query results to only those artifacts that are traced by a number of traces corresponding to the specified trace count filter. Similar to property constraints, a stereotype 'filter' is attached to the multiplicity and visually represented as a filter symbol (see Fig. 6). 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 trace count as a $1..*$ filter. Trace count filters facilitate a wide variety of trace queries, e.g., to retrieve all use cases with no (zero) related system tests, or conversely all use cases that fan-out to two or more system tests (as defined in Fig. 6).

Scope The query scope defines the 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 artifact types of a query and is visually represented as an encircled dot (see Fig. 6). The example in Fig. 6 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 existing use cases.

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 Results

Return values of a query are defined as stereotyped attributes and represented by a bar graph symbol in front of an attribute. The form in which results are returned, depends on the underlying data repository. Assuming a relational database, as used for our prototype implementation and for the examples and comparisons shown in this paper, an executed query will return a table in which each defined return value is represented as a column. Each result to the query forms a row in the table. If the return values originate from more than one artifact types and the concrete artifacts are related by more than one trace then the results form a forest (union of trees). In this case, each permutation of artifact types with return values forms a row in the results table, similar to the results of a SQL query that joins multiple tables.

The formatting and presentation of results to the user are outside the scope of our work as it is dependent upon the capabilities of the selected case tools and upon the nature of the artifact being traced. However, most software development tools provide features for defining and generating filtered views of the stored data. In fact, VTML could be easily integrated into such features to provide advanced support for modeling and executing a broad range of traceability queries. Numerous tools, such as IBM Rational DOORS^{TM}, which currently provides only limited support for filtering across artifact types, would benefit from such an integration.

5.4 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 Fig. 6).

5.5 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.

5.6 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 to be part of the results. We found this 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, it does not currently support a query involving an *or* condition in which artifacts that have either a certain property value or a relation to another artifact are to be included in the trace. In the current version of VTML, such queries must be performed separately, and the results concatenated 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. Therefore, we are currently evaluating the use of filter references that can be used to write complex boolean expression.

Moody [20] 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 TIM (cognitive integration). We apply a cognitively manageable number of visual symbols (graphic economy). Currently, our notation does not use any dual coded information (graphics complemented with text), because of the relatively small number of symbols and the fact that the participants of our experiment had no difficulties in understanding their meaning. However, a modus for the novice user in which symbols are annotated with explanations will be considered for an evolved version of VTML. Finally, we assumed that the TIM is represented as a UML class diagram and created our notation accordingly; however, if the TIM were to be represented using a different notation the VTML notation should be updated accordingly (cognitive fit).

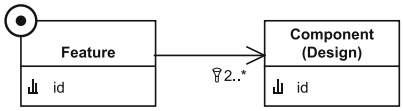
6 Applying visual traceability queries

This section provides examples of several visual traceability queries and discusses different aspects of their application. Although we only depict a small sampling of queries in this

Table 1 Example queries written in SQL and VTML

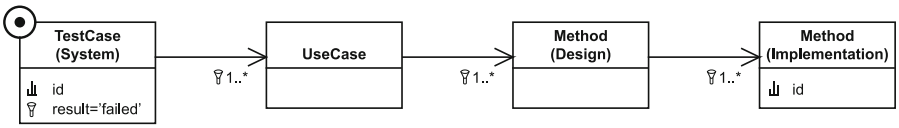
Query A: Find features scattered among multiple components

```
SELECT ‘Feature".id, ‘Component(Design)’.id
FROM ‘Component(Design)’, ‘LINKS-Component(Design)-Feature’, ‘Feature’
WHERE ‘Component(Design)’.id = ‘LINKS-Component(Design)-Feature’.sourceID AND
‘LINKS-Component(Design)-Feature’.targetID = ‘Feature’.id AND ‘Feature’.id IN (SELECT ‘Feature’.id FROM
‘Component(Design)’, ‘LINKS-Component(Design)-Feature’, ‘Feature’ WHERE ‘Component(Design)’.id =
‘LINKS-Component(Design)-Feature’.sourceID AND ‘LINKS-Component(Design)-Feature’.targetID = ‘Feature’.id
GROUP BY ‘LINKS-Component(Design)-Feature’.targetID HAVING
COUNT(‘LINKS-Component(Design)-Feature’.targetID) > 1) AND ‘Feature’.id IN (@scope)
```



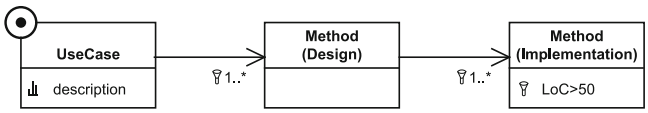
Query B: Find methods implementing ’failed’ system test cases

```
SELECT ‘TestCase(System)’.id, ‘Method(Implementation)’.id
FROM ‘Method(Implementation)’, ‘LINKS-Method(Implementation)-Method(Design)’, ‘Method(Design)’,
‘LINKS-Method(Design)-UseCase’, ‘UseCase’, ‘LINKS-TestCase(System)-UseCase’, ‘TestCase(System)’
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
‘TestCase(System)’.id = ‘LINKS-TestCase(System)-UseCase’.sourceID AND
‘LINKS-TestCase(System)-UseCase’.targetID = ‘UseCase’.id AND ‘TestCase(System)’.result=‘failed’ AND
‘TestCase(System)’.id IN (@scope)
```



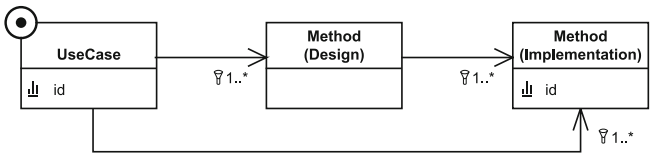
Query C: Show uses cases implemented by methods with $> 50 LoC$

```
SELECT ‘UseCase’.description
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
‘Method(Implementation)’.LoC > 50 AND ‘UseCase’.id IN (@scope)
```



Query D: Find redundant traces between use cases and implementation

```
SELECT ‘UseCase’.id, ‘Method(Implementation)’.id
FROM ‘Method(Implementation)’, ‘LINKS-Method(Implementation)-UseCase’,
‘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
‘Method(Implementation)’.id = ‘LINKS-Method(Implementation)-UseCase’.sourceID AND
‘LINKS-Method(Implementation)-UseCase’.targetID = ‘UseCase’.id AND ‘UseCase’.id IN (@scope)
```



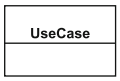
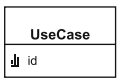
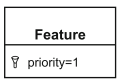
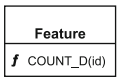
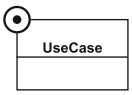
paper, we have used VTML to express a much wider variety of useful trace queries in a mid-sized industrial project.

6.1 Example queries

Table 1 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.

Query A shown in Table 1 finds features that are implemented by more than one component of the design model and so highlights possible deficiencies in the design. Query B returns all methods implementing a ‘failed’ system test case and so facilitates the analysis of the discovered problem in the source code. Query C 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. Query D is inspired by a real world example that finds redundant traceability relations between use cases and implementation methods. While the TIM allows both routes, the idea is that either the one or the other should be chosen by the

Table 2 Comparison of query features expressed in SQL and VTML

	VTML	SQL
Queried artifact type	Specified as class with the name of the queried artifact type 	Specified in the FROM clause `...FROM 'UseCase'...`
Queried trace type	Specified as association between two queried artifact types 	Specified in the FROM clause `...FROM 'LINKS-Method(Design)-UseCase'...`
Return values of a query	Specified as attribute within class and distinguished with results symbol 	Specified in the SELECT clause `SELECT 'UseCase'.id...`
Artifact property filter	Specified as attribute within class and distinguished with filter symbol 	Specified in the WHERE clause `...WHERE 'Feature'.priority=1...`
Trace count filter	Specified as attribute within class and distinguished with results symbol 	Specified in either the WHERE clause, as joins and/or as nested queries depending on count 0: artifact must not be part of a nested query inner joining artifact and link tables 0..*: left join artifact and link tables 1..*: inner join artifact and link tables 2..*: inner join artifact and link tables, and a nested query constraining the relation count
Aggregation function	Specified as method within class and distinguished with function symbol 	Specified in the SELECT clause `SELECT COUNT(DISTINCT 'Feature'.id)...`
Scope: input-set of artifacts selectable by user	Specified by dot symbol at one class 	Specified as named parameter 'scope' in the WHERE clause `WHERE 'UseCase'.id IN (@scope)...`

user 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.

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, 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. Table 2 shows how the different features of VTML queries are translated into SQL queries.

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 TIM, 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. Furthermore, it is common practice to store different types of traceability links in separate tables.

6.3 Supporting the creation and validation of queries

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 TIM, the TIM can be used to validate the structural correctness of each query. Additionally, defined constraints can be validated for their syntactical correctness using regular expressions.

As traceability queries are built on top of a TIM, any change to the TIM can also have an effect on existing queries. While extensions to the TIM will have no effect on defined queries, deletions and modifications could invalidate a query if the required information becomes unavailable following the change. Due to the coupling between the TIM and its associated queries, the renaming of artifacts and properties in the TIM can be resolved automatically across all of the queries. For larger modifications and deletions, an automated detection of impacted queries will be possible. Whether such impacted queries can be evolved automatically or not, will be the subject of future work.

Our tool revalidates queries each time they are transformed into the executable format. The current prototype implementation checks consistency of a query with the TIM using a script that evaluates that each queried artifact type, each trace type and each property used in the query is also defined within the TIM. In the case of a mismatch, the script highlights contradictions within the query. As future work, we plan to evaluate a technique that generates an up-to-date UML profile for the definition of VTML queries based on the currently defined TIM for a project. That profile would support the creation of queries as well as checking them.

7 Evaluation

In this paper the expressiveness of VTML is evaluated through a case study, and its readability and writability are evaluated through a user study which compares the readability and writability of VTML to that of SQL.

7.1 Expressiveness

For purposes of this study, we define expressiveness of a TQL, as the ability to (1) model all desired trace queries, and (2) model trace queries easily without resorting to complicated work-arounds. We evaluated the expressiveness of VTML through developing an extensive set of traceability links for the iTrust project.

Project iTrust is a web-based medical application that provides healthcare personnel, patients, and other medical stakeholders with the means to manage their medical records and communications. iTrust is implemented in Java and JSP, and was initially developed at North Carolina State University under an NSF grant [21]. Though not an industrial project, iTrust is a full blown application developed and improved through eleven major versions. iTrust has been developed to comply to the US Health Insurance Portability and Accountability Act (HIPAA) of 1996. As such, it represents the kind of system which is expected to employ traceability. In particular, we selected iTrust because it reflects a well-documented development including substantial use cases, test cases and traceability.

Method To evaluate the expressiveness of VTML, we developed a representative set of trace queries for iTrust. Unfortunately, although numerous research papers, and training materials for software engineering support tools, describe various kinds of requirements traces, there is no accepted standard defining a set of typical or required traceability queries. This problem has also been identified by the traceability community and has been documented as one of the Grand Challenges of Traceability [22,23].

This lack of reference material creates a challenge for demonstrating the expressiveness of a TQL, as there is no existing standard against which the new language should be evaluated. We, therefore, conducted a survey of traceability research literature and online training materials for requirements management tools, and also drew on the industrial traceability knowledge of both authors, to identify different categories and different attributes of trace queries which should be representable in a TQL. Although these sources produce sometimes competing views and classification systems, we were able to extract the following coverage criterion for evaluation purposes:

Criteria #1: Basic traces The research literature reports multiple classification schemes for traceability queries. Ramesh identified four primary types of trace query which he referred to as *satisfaction*, *dependencies*, *evolution*, and *rationale* [1]. Brcina et al. identified five primary purposes for traceability and defined them as *verification*, *change impact analysis*, *software comprehension and reverse engineering*, *decision support*, and *system configuration and versioning* [24]. Pohl analyzed 30 different publications on traceability and constructed a taxonomy of traceability link types which included five top-level traceability link types as *requires*, *supports*, *obstructs*, *conflicts*, and *depends on*. Other researchers such as Jirapanthong have developed extensive classifications for specific traceability domains such as product lines [25]. However, a quick analysis of these trace categories shows that they differ only in the trace *target*, the trace *source*, and the link *semantics*. It, therefore, is not necessary to reproduce each of these trace categories to demonstrate the expressiveness of a TQL. Instead, it is simply necessary to show that VTML supports a trace from a source artifact to a target artifact across a link of a specific type.

Criteria #2: Link direction Jarke et al. [26] classified trace queries according to their direction with respect to requirements. He identified the four categories of *forward-to*, *backward-from*, *forward-from*, and *backward-to*. Other researchers have placed different types of artifacts at the center of the traceability process. For example, Hendrickson et al. [27] focus traces around architectural concerns instead of requirements. In practice, traces can originate from and be traced to any artifact in either direction. We, therefore, include link direction as one of our coverage criterion.

Criteria #3: Exists versus missing Our experience of working on industrial traceability projects and our analysis of online training materials for tools such as Rational Software Architect also show that traceability links can be classified according to whether related artifacts *exist*, i.e., a query requesting "all test cases related to requirement R", or are *missing*, i.e., "all requirements which have no associated test cases."

Criteria #4: Constraints Traces are often refined according to constraints placed on the attributes of the source or target artifacts, attributes of the link, or even the cardinality of the link [28]. For example, the query "return all test cases created in the past week" uses the test case creation date as a constraint, while the query "return all requirements with two or more test cases" uses cardinality of the link to constrain the results.

Criteria #5: Simple versus transitive As numerous researchers have shown [29], traceability links are often transitive in nature, such that a trace query involves chains of individual traceability links, composed together using various constraints upon links and elements.

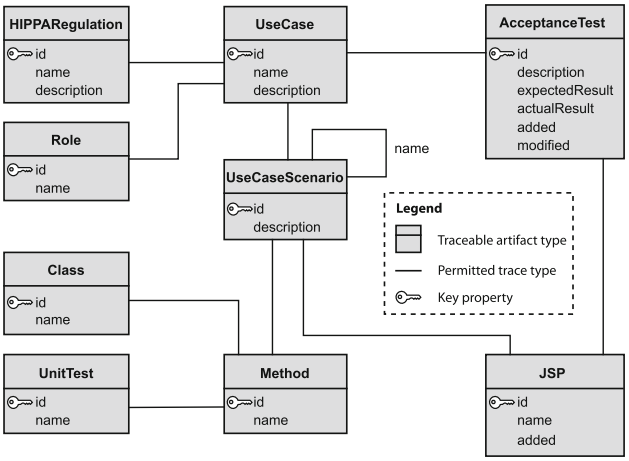
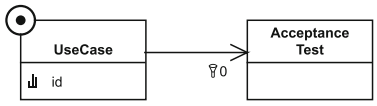


Fig. 7 Traceability information model for the iTrust project

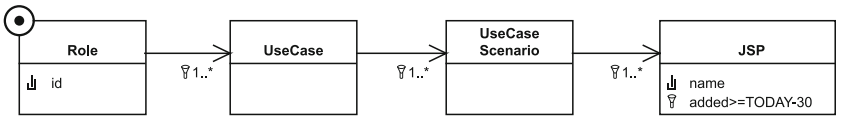
iTrust traceability information model The TIM for the iTrust system is shown in Fig. 7. The TIM was reverse engineered according to the iTrust artifacts and traceablity paths between them. iTrust includes an extensive, well-defined set of use cases, each of which is associated with one or more user roles (such as physician, nurse, patient etc). Each use case defines a number of different scenarios. Usage scenarios are implemented within methods, belonging to classes, and also in Java server pages forming the user interface. The system is also supported by an extensive set of automated unit test cases and also acceptance tests. A subset of the acceptance tests is automated and each one of these is related to exactly one Java server page. Our TIM reflects all these types of artifacts and the relations among them. Furthermore, it defines important attributes for each of the artifacts and the mapping the actual tool artifacts as described in Sect. 4.2.

iTrust trace queries As there is an almost endless combination of possible trace queries for the iTrust project, we developed a set of trace queries which provided good coverage of the different artifact types, and also good representation of each of the possible types of queries and constraints discussed in the previous paragraph. These queries are shown in Fig. 8. It should be noted that constraints upon the scope of the query, i.e., a constraint specifying that a trace applies only to a subset of the source artifact type, is not visible in our notation. This is because we model a single query, and the scope constraint is provided as a runtime parameter by the user.

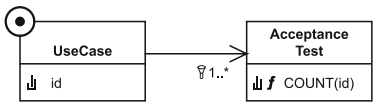
Analysis of expressiveness We experienced few difficulties in modeling trace queries for the iTrust project with two exceptions. The first case was query 8e which was stated



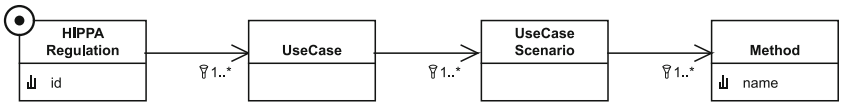
(a) Query: Return all use cases without acceptance test cases.



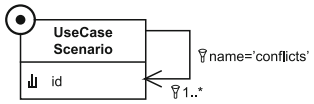
(b) Query: Return all JSP completed in the past 30 days per role.



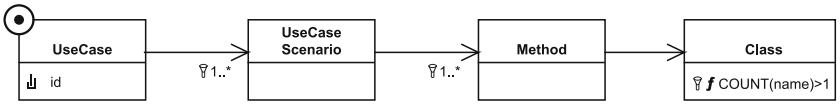
(c) Query: Return a count of test cases per use case.



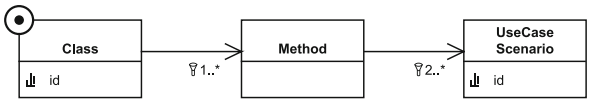
(d) Query: Find all source code methods that contribute to satisfying a given HIPAA regulation.



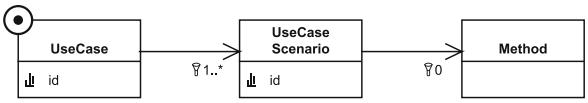
(e) Query: For a given use case scenario return all other use case scenarios for which it exhibits conflicts.



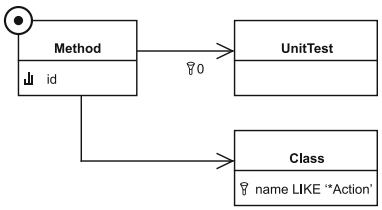
(f) Query: Return a list of use cases implemented across more than one class.



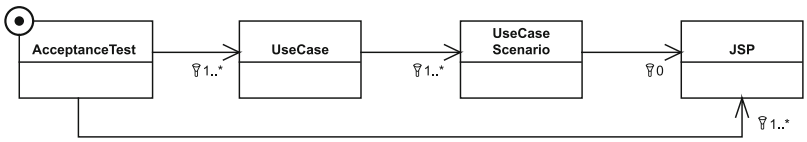
(g) Query: Return all classes that trace to more than two use case scenarios.



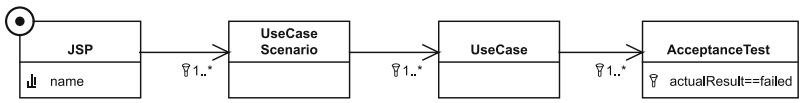
(h) Query: Find all use cases without implemented methods.



(i) Query: Find all methods that belong to an action class without associated unit tests.



(j) Query: Return acceptance tests, with missing traces between use case scenarios and JSPs based on redundant traceability paths.



(k) Query: Return all JSPs with failed acceptance tests.

Fig. 8 Visual traceability queries for the iTrust project

as "for a given use case scenario return all other use case scenarios for which it exhibits a conflict". This query required the ability to filter results according to an attribute of the link; however, this was not part of our original modeling notation [4], due in large part to the fact that in practice many traceability matrices do not differentiate between link types at the individual trace level. Our strategy for designing VTML is to be minimalistic so that the language contains only necessary features. As a result of recognizing this need in a real project, we extended the notation to include filtering on link attributes as depicted in Fig. 6. The second issue we experienced was in the case of query 8j which was designed to identify acceptance tests with missing traces between use case scenarios and JSPs based on a redundant traceability path that additionally connects acceptance tests and JSPs directly. The motivation for that query was that in the iTrust project some acceptance tests are related not only to use cases, but also to the JSP on which the test is performed.

Query (Figure)	Scope and coverage: HIPPA regulation	Role	Use case	Use case scenario	Acceptance test	Method	Class	JSP	Unit test	Query categories and attributes: Exists	Missing	Simple	Transitive	Link constraint	Artifact constraint	Cardinality const.	Forwards	Backwards	Trace evaluation	Aggregation
Return all use cases without acceptance test cases. (8a)			●		▪						●	●					●			
Return all JSPs completed in the past 30 days per associated role. (8b)		●	▪	▪				▪		●			●		●		●			
Return a count of test cases per use case. (8c)			●		▪					●		●					●			●
Find all source code methods that contribute to satisfying a given HIPAA regulation. (8d)	●		▪	▪		▪				●			●				●			
For a given use case scenario return all other use case scenarios for which it exhibits conflicts. (8e)				●						●		●		●						
Return a list of use cases implemented across more than one class. (8f)			●	▪		▪	▪			●			●		●		●			
Return all classes that trace to more than two use case scenarios. (8g)				▪		▪	●				●		●			●		●		
Find all use cases without implemented methods. (8h)			●	▪		▪					●		●				●			
Find all methods that belong to an action class without associated unit tests. (8i)						●	▪		▪		●	●			●		●			
Return acceptance tests, with missing traces between use case scenarios and JSPs based on redundant paths. (8j)			▪	▪	●			▪		●	●		●				●		●	
Return all JSPs with failed acceptance tests. (8k)			▪	▪	▪			●		●			●		●			●		

Fig. 9 VTML feature coverage of the iTrust queries (The *filled circles* used in the left part of the table mark the scope of each query, while the *filled squares* mark which other artifact types a query covers. The *filled circles* used in the right part of the table mark which coverage criteria a query meets)

The query identifies tests that are traced to a JSP directly, but not indirectly via the use case. Although our notation was certainly able to express this query, it took more time than the other queries to construct. We also expect it to be less intuitive for readability purposes. Nevertheless, now that we have modeled this type of query, it would be very simple to reproduce.

Figure 9 depicts the extent to which our iTrust queries covered the evaluation criterion we had previously defined. For example we included at least one trace query originating at each of the iTrust artifact types except for unit test cases. Furthermore, the trace queries were designed to provide broad representation of the trace query types we previously identified, i.e., simple trace queries between source and target artifacts, transitive queries, queries that checked for the existence of relations (i.e., exists vs. missing), and queries with a variety of constraint types.

Our experience writing these queries demonstrated that the language was sufficiently expressive to construct all of the desired trace queries for the iTrust project. We also found it quite intuitive to use. However, this issue is further explored through a user study described in the following section.

7.2 Understandability and usability

We designed an 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 VTML queries result in a more accurate and faster understanding of a query's purpose compared to equivalent techniques?
- Q2 Constructing: Does the use of VTML 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. 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 (see Table 3). The table reports for six relevant areas of experience on a self scoring of participants on a scale from 0 to 3 (0 meaning no experience, 3 meaning expert knowledge)

Table 3 Prior experience of subjects

	Score [0–3]		Years	
	Mean	SD	Mean	SD
SW/systems development	2.2	0.9	6.2	5.4
Requirements engineering	0.9	0.9	1.7	3.0
Model-based development	0.8	1.0	1.8	4.0
Requirements traceability	0.8	1.1	1.3	2.4
UML modeling	1.4	1.0	2.2	3.7
SQL querying	1.7	1.1	3.5	4.0

and the time of experience in years. 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 were 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:

1. Each subject completed a series of questions to describe her/his background and experience in the field of software and data engineering.
2. Each subject read a tutorial about the general purpose of software traceability, the use of a TIM, 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. Each participant was given a set of nine different queries. Each of these queries was 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 nine participants in SQL and nine in VTML.
4. Each subject was also asked to construct two queries, one written in SQL and one modeled in VTML. The assignment was random.
5. Each subject completed a questionnaire concerning his or her experience using both VTML and SQL to read and construct traceability queries.

Results Q1 Reading queries Table 4 shows that subjects viewing our visual notation responded on average (mean) 26–63% faster to the nine questions (R1–R9) than subjects viewing the same query in SQL notation, thereby reducing

Table 4 Time differences [s] for performing tasks

Task	VTML		SQL		Diff	*t* test
	Mean	SD	Mean	SD	VTML (%)	*p* value
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	

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 10 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 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.

Results Q2 Constructing queries Table 4 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 Fig. 10), 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.

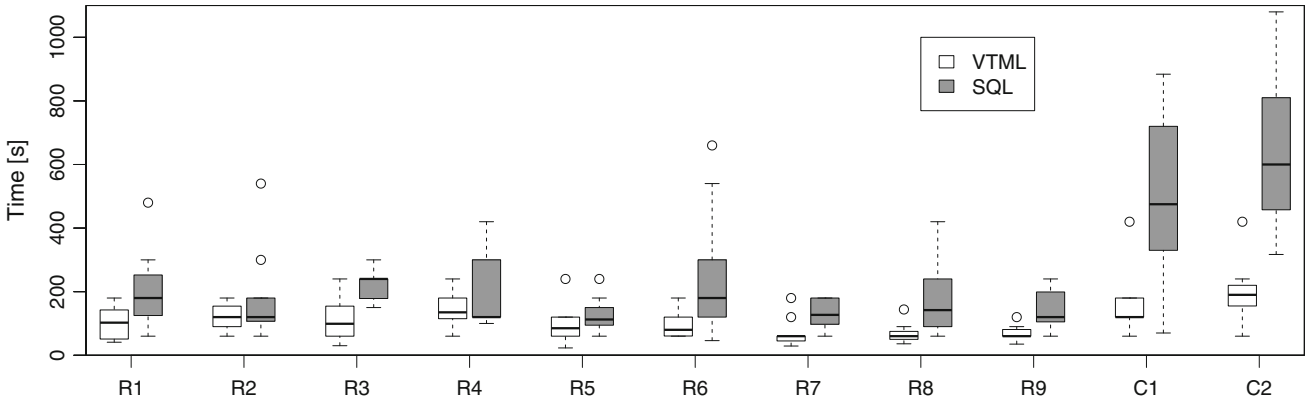


Fig. 10 Time required to understand (R1–R9) and construct queries (C1, C2)

7.3 Threats to validity

Important threats to the validity of the experiment are divided into four common categories as suggested by [30].

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.

We used a variety of nine different queries for the reading, but only two different queries for the writing partly due to time restrictions of our participants. Though carefully selected, there is a risk that the two chosen queries for the writing part may not be representative enough for a large population of possible queries. We will evaluate that issue in future studies.

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 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. We provide our experimental material as an electronic appendix (see Electronic Supplementary Material) giving other researchers the chance to get a clear understanding of what we did and to replicate our experiment. 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, constructing (writing), and expressiveness of queries as we believe these qualities to be the most important ones for a visual traceability modeling language. If a notation is easier to use and comprehend, then the measures of time should correspondingly show lower values and those of correctness higher 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.

VTML was evaluated in two different ways. First, the case study that we conducted demonstrated the ability to use VTML to express a broad set of traceability queries,

defined following an extensive literature study. VTML was found to be sufficient to express all of the desired types of trace queries for the iTrust project. Second, the user experiment we performed demonstrated that users were able to read and construct traceability queries more quickly using VTML than using SQL. When comprehending visual queries, users gave slightly more (5%) incorrect answers regarding the purpose of a query. An analysis of that issue suggests an initial learning curve during which the majority of false answers were given. 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 be used by stakeholders to create traceability links that help them perform useful tasks in an industrial settings.

Acknowledgments The authors would like to thank all participants of the experiment for their dedicated work. Furthermore, the authors would also like to thank the anonymous reviewers of an earlier version of this paper for their useful suggestions for improving the final paper. This work was partially funded by the National Science Foundation grant #CCF: 0810924.

References

1. Ramesh, B., Jarke, M.: Toward reference models of requirements traceability. IEEE Trans. Softw. Eng. **27**(1), 58–93 (2001)
2. Arkley, P., Riddle, S.: Overcoming the traceability benefit problem. In: Proceedings of the 13th International Requirements Engineering Conference, pp. 385–389. IEEE Computer Society, New York, ISBN: 0-7695-2425-7 (2005)
3. 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)
4. Mäder, P., Cleland-Huang, J.: A visual traceability modeling language. In: MoDELS, vol. 1, pp. 226–240 (2010)
5. Zloof, M.: Query-by-example: A database language. In: Query-by-Example: A Database Language, pp. 324–343. IBM Systems Journal (1977)
6. Post from HP Quality Center™ Support Forum. http://h30499.www3.hp.com/t5/ITRC-Quality-Center-Forum/traceability-query/m-p/4505026. Accessed 15 July 2011 (2009)
7. 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 (Washington, DC, USA). pp. 16–20, IEEE Computer Society, New York (2009)
8. Zhang, Y., Witte, R., Rilling, J., Haarslev, V.: An ontology-based approach for the recovery of traceability links. In: 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, Italy, October 1st 2006
9. Guerra, E., de Lara, J., Kolovos, D., Paige, R.: Inter-modelling: From theory to practice. In: Petriu, D., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 6394, pp. 376–391. Springer, Berlin (2010). doi:10.1007/978-3-642-16145-2
10. Kim, H.-J., Korth, H.F., Silberschatz, A.: Picasso: a graphical query language. Softw. Pract. Exp. **18**, 169–203 (1988)
11. Jaakkola, H., Thalheim, B.: Visual sql—high-quality er-based query treatment. In: Jeusfeld, M., Pastor, Ó (eds.) Conceptual Modeling for Novel Application Domains. Lecture Notes in Computer Science, vol. 2814, pp. 129–139. Springer, Berlin (2003). doi:10.1007/978-3-540-39597-3
12. Harald, Störrle: VMQL: A visual language for ad-hoc model querying. J. Vis. Lang. Comput. **22**(1), 3–29 (2011)
13. Wieringa, R.: An introduction to requirements traceability. Technical Report IR-389, Faculty of Mathematics and Computer Science, November 1995
14. 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 (2008)
15. 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), October 2003
16. 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, New York, September (2006)
17. 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, USA (1994)
18. Dömges, R., Pohl, K.: Adapting traceability environments to project-specific needs. Commun. ACM **41**, 54–62 (1998)
19. 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
20. Moody, D.L.: The 'Physics' of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)
21. iTrust: Role-Based Healthcare System. http://agile.csc.ncsu.edu/iTrust/ (2011)
22. Cleland-Huang, J., Czauderna, A., Dekhtyar, A., Gotel, O., Huffman Hayes, J., Keenan, E., Leach, G., Maletic, J., Poshyvanyk, D., Shin, Y., Zisman, A., Antonio, G., Berenbach, B., Egyed, A., Maeder, P.: Grand challenges, benchmarks, and tracelab: developing infrastructure for the software traceability research community. In: International Workshop on Traceability in Emerging Forms of Software Engineering. ACM Press, New York (2011)
23. Gotel, O., Cleland-Huang, J., Huffman Hayes, J., Zisman, A., Egyed, A., Grunbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J.: The Grand Challenges of Traceability 1.0. In: Cleland-Huang, J., Gotel, O., Zisman, A. (eds.) Software and Systems Traceability, pp. 343–409. Springer, January 2012

24. Brcina, R., Riebisch, M.: Defining a traceability link semantics for design decision support. In: ECMDA Traceability Workshop, pp. 39–48 (2008)
25. Jirapanthong, W., Zisman, A.: Xtraque: traceability for product line systems. Softw. Syst. Model. **8**(1), 117–144 (2009)
26. Jarke, M.: Requirements tracing. Commun. ACM **41**, 32–36 (1998)
27. Hendrickson, S.A., Dashofy, E.M., Taylor, R.N.: An (architecture-centric) approach for tracing, organizing, and understanding events in event-based software architectures. In: IWPC, pp. 227–236 (2005)
28. Schwarz, H., Ebert, J., Lemcke, J., Rahmani, T., Zivkovic, S.: Using expressive traceability relationships for ensuring consistent process model refinement. In: ICECCS, pp. 183–192 (2010)
29. de Leon, D.C., Alves-Foss, J.: Hidden implementation dependencies in high assurance and critical computing systems. IEEE Trans. Softw. Eng. **32**(10), 790–811 (2006)
30. Yin, R.K.: Case Study Research: Design and Methods. 3rd edn. Sage, Thousand Oaks (2003)

Author Biographies

Patrick Mäder received a Diploma degree in industrial engineering and a PhD degree (Distinction) in computer science from the Ilmenau University of Technology in 2003 and 2009, respectively. He worked as a consultant for the EXTESSY AG, Wolfsburg between 2003 and 2005. Currently, he is a postdoctoral fellow at the Institute for Systems Engineering and Automation (SEA) of the Johannes Kepler University, Linz. Dr. Mäder also has an active collaboration with the Software and Requirements Engineering Center at DePaul University, Chicago. His research interests include topics related to software engineering, with a focus on requirements traceability, and object-oriented analysis and design.

Jane Cleland-Huang received the PhD degree in computer science from the University of Illinois at Chicago in 2002. She is currently an associate professor in the School of Computing at DePaul University, Chicago, where she serves as the director of the Systems and Requirements Engineering Center. She also serves as the North American Director of the International Center of Excellence for Software Traceability. Her research interests emphasize the application of machine learning and information retrieval methods to tackle large-scale and safety-critical Software Engineering problems. Dr. Cleland-Huang serves on the Editorial Board for the Requirements Engineering Journal, and IEEE Software, and as Associate Editor for IEEE Transactions on Software Engineering. She has been the recipient of the US National Science Foundation Faculty Early Career Development Award, three ACM SIGSOFT Distinguished Paper Awards and 2006 IFIP TC2 Manfred Paul Award for Excellence in Software: Theory and Practice. She is a member of the IEEE Computer Society and the IEEE Women in Engineering.