Requirements Eng (2015) 20:215–232
DOI 10.1007/s00766-015-0224-4

RE 2014

TiQi: answering unstructured natural language trace queries

Piotr Pruski[1] · Sugandha Lohar[1] · William Goss[1] · Alexander Rasin[1] · Jane Cleland-Huang[1]

Received: 25 October 2014 / Accepted: 26 February 2015 / Published online: 18 March 2015

Abstract Software traceability is a required element in the development and certification of safety-critical software systems. However, trace links, which are created at significant cost and effort, are often underutilized in practice due primarily to the fact that project stakeholders often lack the skills needed to formulate complex trace queries. To mitigate this problem, we present a solution which transforms spoken or written natural language queries into structured query language (SQL). TiQi includes a general database query mechanism and a domain-specific model populated with trace query concepts, project-specific terminology, token disambiguators, and query transformation rules. We report results from four different experiments exploring user preferences for natural language queries, accuracy of the generated trace queries, efficacy of the underlying disambiguators, and stability of the trace query concepts. Experiments are conducted against two different datasets and show that users have a preference for written NL queries. Queries were transformed at accuracy rates ranging from 47 to 93 %.

✉ Jane Cleland-Huang
jhuang@cs.depaul.edu

Piotr Pruski
ppruski@gmail.com

Sugandha Lohar
slohar@cs.depaul.edu

William Goss
william.t.goss@gmail.com

Alexander Rasin
arasin@cdm.depaul.edu

[1] DePaul University, Chicago, USA

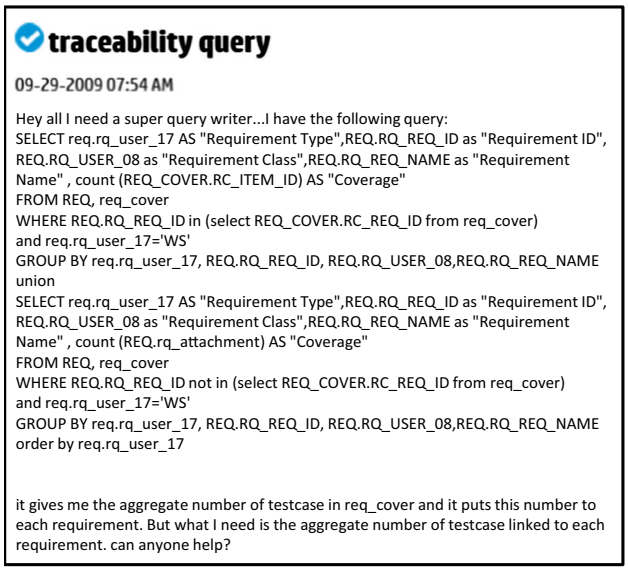
Keywords Traceability · Queries · Speech recognition · Natural language processing

1 Introduction

Most safety-critical domains, such as avionics, medical devices, and transportation [25], are regulated by certifying bodies. These bodies prescribe a set of software development practices and guidelines that include the need to demonstrate traceability between critical artifacts such as hazards, faults, mitigating requirements, design, code, and test cases [3, 33]. As a result, developers often invest significant cost and effort into creating trace links in order to meet certification or compliance requirements. Unfortunately, these trace links are often underutilized due to lack of tool support, poor understanding of the underlying trace schema, and lack of skills needed to formulate useful trace queries [11, 24, 25, 39].

The problem is exacerbated by the fact that trace queries are often quite complex and cut across many different artifacts. For example, a typical trace query could synthesize data about faults, mitigating requirements, code, test cases, and test logs, as well as the trace links that connect them [4]. This leads to significant complexity, especially as many software developers have only rudimentary skills at constructing structured queries using technologies such as SQL or XQuery [24]. This is anecdotally illustrated by the call for help in Fig. 1. A person attempts to formulate a trace query in SQL but is thwarted by the need to aggregate the number of test cases linked to each requirement—coupled with the complexity of joining not only data files but trace matrix files.

The goal of our research is therefore to make traceability information more readily accessible to project stakeholders



traceability query

09-29-2009 07:54 AM

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 Post from the Hewlett Packard help forum illustrating the challenges of modeling trace queries using SQL [31]

by allowing them to issue trace queries using either spoken or written natural language (NL) queries. NL query solutions started to emerge in the 1970s and 1980s [38], and while the field stagnated for many years as researchers refocused on developing NL interfaces for less structured sources of information [9], there are several factors which make the database interface problem a compelling one to revisit for traceability purposes today.

First, raw trace data are becoming increasingly available in software projects. There are a number of reasons for this including the fact that regulated industries insist on traceability [25], integrated development environments such as Jazz [5] produce links as a natural by-product of the development process, and advances in state-of-the-art tracing techniques have matured to the extent that they can be used to instrument the project environment in order to *capture* links or to *generate* trace links through the use of information retrieval techniques [1, 14]. Second, advances in speech recognition software make speech interfaces a viable option [12], and third, previous studies of NL queries for general databases have shown that natural language queries are simpler and more succinct to create than their SQL counterparts [38], suggesting that NL solutions could produce a viable solution to the trace usage adoption barrier.

In our previous paper presented at the International Requirements Engineering Conference, entitled "TiQi: Towards Natural Language Trace Queries" [32], we presented a solution for transforming NL trace queries into SQL. Prior to engaging in this project, we assumed that we could find and adopt an open-source general NL database query language and customize it for the traceability domain; however, as discussed in the related work section of this paper, an extensive search of available open-source solutions was unsuccessful. We therefore first constructed a generic NL database query mechanism and then customized it with vocabulary and concepts from the traceability domain. The result of this work is *TiQi*, which transforms spoken or written natural language trace queries into executable SQL statements.

In this paper, we extend our prior work by providing a more in depth description of the architecture, design, and heuristic rules adopted by TiQi and then empirically evaluating their use in the NL transformation process. We also extend the set of functions supported by TiQi to include more complex numbers, group-bys, comparisons, union queries, summation, count, as well as a mechanism for making more precise matches in the lexicon phase. To increase the difficulty level, we have updated both of the datasets used in experiments by adding additional data and a more extensive set of trace queries. Finally, we have replicated our previous experiment to evaluate the modified version of TiQi against the more challenging datasets, and report two entirely new experiments to evaluate the stability of our trace query domain model and the efficacy of TiQi's token disambiguation techniques. The extended set of sample queries and enlarged datasets provide a more rigorous evaluation of TiQi and highlight the difficulty of querying datasets which contain both structured and unstructured data.

The remainder of this paper is laid out as follows. Section 2 discusses the role of a Traceability Information Model (TIM) in the TiQi process, while Sect. 3 describes the underlying domain model. Section 4 presents the process TiQi uses to transform a NL query to SQL. Section 5 describes a series of studies that were conducted to evaluate TiQi and its underlying concepts. Section 6 describes threats to validity. Section 7 discusses related work, and finally, Sect. 8 concludes with a summary of our findings and a proposal for future work.

2 The role of the traceability information model in TiQi

In order for a person to issue a trace query, she or he needs to know what artifacts and supporting trace links are available for querying purposes. In a safety-critical system, these will typically include hazards, faults, mitigating requirements, design, code, and test cases. Certain artifact types will be connected by trace paths. Ideally, this information is documented in the form of a *Traceability Information Model* (TIM) [25] in which *artifact types* and their properties are represented as classes and attributes, and *permitted trace links* are represented as semantically typed

associations. In general, a TIM is used to strategically plan the traceability for a project, by serving as a guide for instrumenting the project environment to enable the creation, maintenance, and use of the planned trace links. It can also be reverse engineered from existing datasets using simple parsing tools [10, 33].

The TIM informs the user of available artifacts, attributes, and connecting trace paths. It therefore suggests, but does not constrain, the language a user might opt to use to express a NL trace query. For example, given the TIM in Fig. 2, a project stakeholder might issue a query such as "list all relevant regulatory codes not covered by at least one software requirement," or "return a list of hazards which have failed unit test cases associated with them." Several of these terms, including "regulatory code" and "software requirement," are taken straight from the TIM. On the other hand, the user might choose to describe artifacts in different ways, for example, by referring to the preliminary hazard tables as a FMECA (failure mode, effects, and criticality analysis) or to the software requirements as the SRS (software requirements specification). The goal is for TiQi to accommodate the full range of possible terminology.

From a physical perspective, as trace links often represent many-to-many dependencies, the trace paths (i.e., associations between artifact types) must be treated as first-class citizens in the underlying database schema. As a result, each artifact, and each association, is represented as a distinct table. Assuming a naming convention in which the trace matrix that establishes links between two artifacts *A* and *B* is called *TM_A_B* and captures each trace link as a pair of IDs taken from *A* and *B* respectively, then the first query could be specified in SQL as follows:

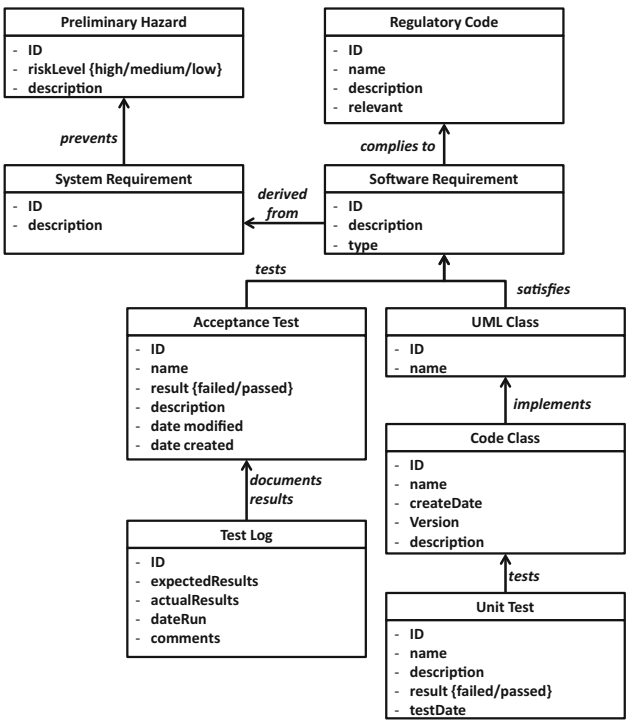


Fig. 2 Sample traceability information model

```
SELECT DISTINCT regulatory-code.*
FROM ((regulatory-code LEFT OUTER JOIN
 tm_regulatory-code_software-requirement ON
 (regulatory-code.id=tm_regulatory-code
 _software-requirement.regulatory-code-id))
 LEFT OUTER JOIN software-requirement ON
 (software-requirement.id=tm_regulatory-code
 _software-requirement.software-requirement-id))
WHERE regulatory-code.relevant='true'
AND software-requirement.id IS NULL;
```

In practice, software project artifacts are usually not stored in a single SQL database, but are instead stored in a variety of third party case tools and/or repositories [7]. For example, source code is typically stored in a version control system such as PerForce or Git, while requirements are stored in tools such as DOORS or Requisite Pro, and designs are stored as text alongside the requirements or as models in tools such as Enterprise Architect (EA) or PTC-creo. Furthermore, in large projects, the various tools and repositories are often distributed behind firewalls across organizational boundaries [7, 34]. While frameworks exist for tracing across an enterprise environment, the focus of this paper is upon the trace query transformation mechanism. We therefore extracted all data in advance from its native repositories and stored it in a SQL database. In project environments in which the query data are not centralized, there will be an extra step in the query execution process to dynamically retrieve the data needed to support each specific query [7].

In this paper, we also assume that all available trace links have been constructed in advance. However, TiQi could also adopt a just-in-time approach in which trace links are generated upon demand using trace retrieval tools such as Poirot [22], RETRO [14], or ADAMS [23].

3 The TiQi domain model

In a seminal discussion on the notion of general versus domain-specific NL query languages [35], Shwartz provided an interesting example from the petrochemical domain in which he compared four very similar queries: (1) show oil wells from 1970 to 1908, (2) show oil wells from 7000 to 8000, (3) show oil wells from 1 to 2000, and finally (4) show oil wells from 40 to 41 and 80 to 81. To an outsider, these queries appear very similar in nature. However, a domain expert would infer that the first query

refers to dates, the second to well depths, the third to map depths, and the fourth to well numbers. A general database system would have little hope of understanding these concepts unless units of measure were explicitly stated. Shwartz therefore claimed that conceptual domain-specific knowledge is critical for understanding the nuances of queries. Similar issues apply for querying software artifacts.

In order to build a domain model for TiQi, we need to understand the kinds of terminology a user might use to formulate a query. Following an analysis of an initial collection of sample trace queries, we identified three different categories of terms that could occur in a trace query. These were *project-specific*, *trace query language*, and *junk*.

3.1 Project-specific vocabulary

Project-specific terms are those terms which describe artifact types and attribute names, or which are found in the project's raw data (e.g., within a requirement or a test case). As depicted in Fig. 3a, these terms can be extracted directly from the TIM. For example, the TIM in Fig. 2 offers vocabulary items such as "preliminary hazard," "risk level," and "test log" (shown here following a simple processing step in which variable names are split into their constituent parts). The second knowledge source is the traceable data itself. These are the raw data (i.e., rows or records) such as hazard descriptions, types of software requirements, and code version numbers. In addition, project-specific vocabulary also includes synonyms of these terms. We discover these synonyms dynamically during the NL query transformation process by using word net [28]. We currently do not consider link labels such as "tests" or "implements" as we rarely see trace links labeled in practice, and furthermore, our experiences have shown that people use a wide array of terminology in their NLP queries to describe relationships. We will consider these in future work.

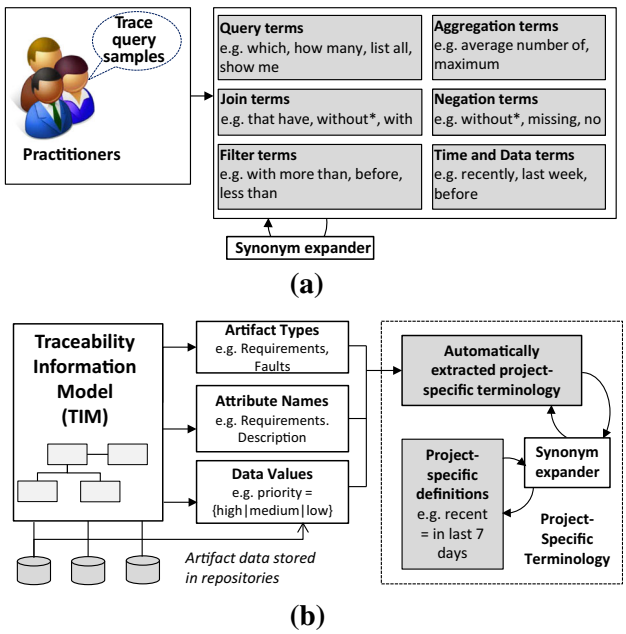


Fig. 3 TiQi process. **a** General query terminology is learned through analyzing a set of trace queries **b** Project-specific terminology is extracted from the Traceability Information Model (TIM) and the underlying data. Definitions for specific terms can be provided by the user

3.2 Trace query language

The second class of terms found in trace queries is terms that compose the query mechanism itself and form the "glue" which holds the pieces of the query together. For example, terms such as *show me*, *list all*, or *I'd like to see* are all synonyms for the SQL construct *SELECT*. Similarly, terms such as *associated with*, *related to*, or *with* are synonyms for various forms of *JOIN*. To discover an initial set of query terminology, we developed an online web collection tool which displayed a TIM and prompted the user for sample NL trace queries. We invited eight traceability experts to formulate three to five trace queries for each of three TIMs. This produced an initial dataset of approximately 100 trace queries. The trace experts were encouraged to use any words and phrases that they wished. We deliberately set this as an open-ended exercise because we did not want to constrain the vocabulary choice of the trace users. Table 1 shows eight of the sample queries collected during this exercise.

Prior work by Meng and Siu [27] proposed automated approaches for extracting grammars for a new domain. Their approach infers a grammar from an unannotated corpora of queries. While such approaches have been shown to be fairly effective, they require a very large corpora of sample queries and are therefore more appropriate for domains with thousands (or even hundreds of thousands) of users, such as airline travel inquiries. Given the limited number of sample queries available to us at this time, we chose to extract the vocabulary manually through systematically analyzing each of the queries to identify *task-related* terms, *join terms*, and *filter terms*. Each of the observed terms was then mapped onto a simple representative term. Based on the initial sample of 100 queries, we mapped 94 phrases and concepts, such as the mapping of *mitigates* to *prevents*. These terms and mappings have grown iteratively throughout our project as we have collected and/or constructed additional trace queries.

Table 1 Sample of the collected trace queries

1	How many high-level hazards are associated with the security camera?
2	Let me see the test log for all system requirements that fail their acceptance test
3	List all hazards for which the most recent unit test case has failed
4	Are there any orphaned UML classes?
5	List all preliminary hazards that have a high level of risk, and which are not covered by a software requirement of type "mitigating"
6	Show me all unaddressed low-level hazards
7	Retrieve all the usability-related software requirements with currently failed acceptance tests
8	What percentage of relevant regulatory codes have been fully addressed with passed acceptance tests?

3.3 Junk terms

Finally, we define junk terms as those terms that are not classified as being project-specific or trace query language. These terms are ignored by TiQi. However, a new trace query might contain a meaningful term which is misclassified as junk, simply because TiQi does not recognize it as either project-specific or trace query related. Terms misclassified in this way can produce incorrect trace results. In the long run, we need the ability to incrementally and continually grow the domain model. For purposes of the experiments in this paper, we added terms as needed. This issue is explored later in Sect. 5.

4 The TiQi transformation process

The overall TiQi process transforms a natural language trace query into a SQL statement and then executes the SQL. Its primary elements are depicted in Fig. 4. First, the rules manager and lexicon factory are loaded. The rules manager handles mapping and disambiguation rules, while the lexicon factory stores artifact names, attribute names, and data values, and provides the functions needed to retrieve data. The NL trace query is passed through four different processors responsible for pre-lexicon processing, disambiguation, post-lexicon processing, and SQL generation. As depicted in Fig. 5, a tokenized query, in progressive stages of refinement, is produced following each processing step.

In the following sections, we describe each of these steps and illustrate them with a running example of the query: *I'd like to see a list of all preliminary hazards for arm movements which are tested by recent unit tests*. The intermediate forms of the query are depicted in Fig. 5.

4.1 Speech to text

TiQi supports both spoken and written NL trace queries. Spoken queries are transformed into textual format as a preprocessing step [12, 15]. We investigated several tools for supporting the speech to text step, focusing mainly on

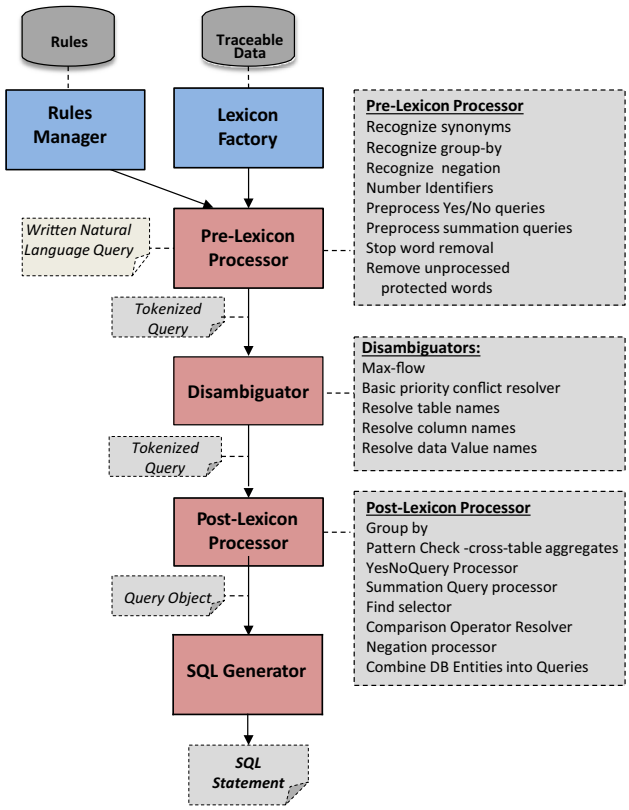


Fig. 4 TiQi transformation process

CMU Sphinx as a local speech processor, and Google Speech API as a web-based option. After training audio data and generating custom, domain-specific language data for Sphinx, the two applications were found to have similar accuracy rates. However, Google's Speech API was more portable, faster, and, ultimately, performed better on average when presented with new data; therefore, it was chosen as our speech processor.

4.2 Pre-lexicon processor

The pre-lexicon parser is primarily responsible for simplifying the query. It matches phrases found in the query to specific mapping rules and then replaces the phrases with the

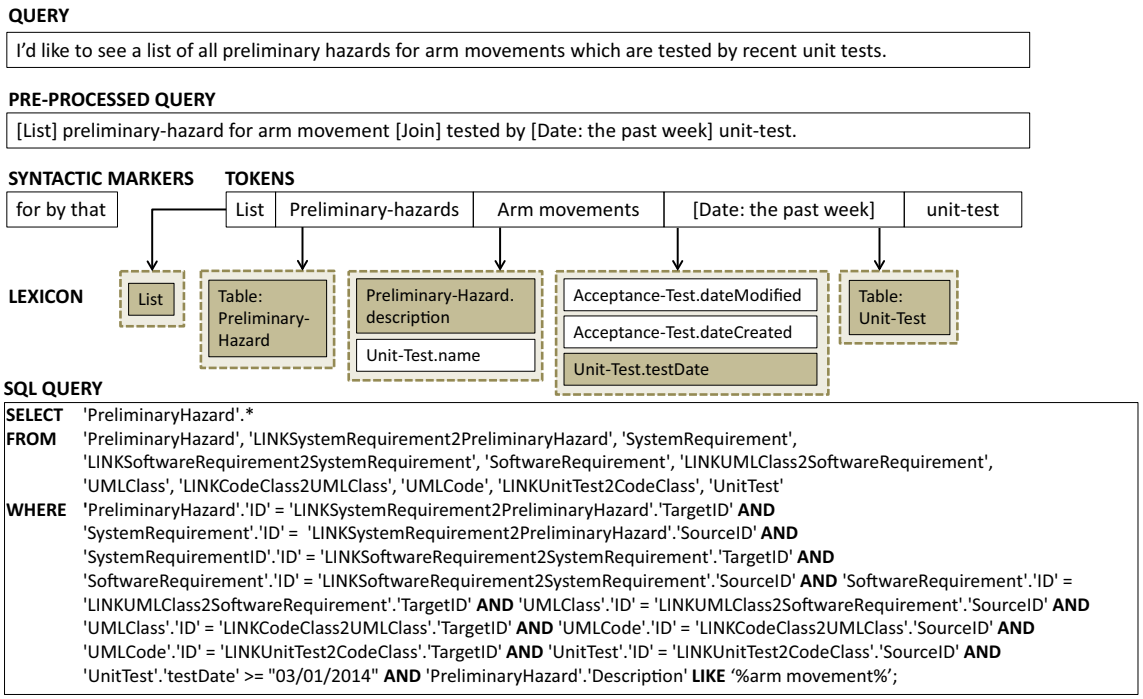


Fig. 5 Steps in the transformation process from a NL Query to an executable SQL query. The query is first simplified and then tokenized through mapping to the lexicon. When alternate mappings are possible, disambiguation occurs through applying a set of heuristic rules and executing the max-flow algorithm to optimize satisfaction of the results

mapped terms. Specific heuristics include phrase-based synonym matching, definition replacements, and the transformation of numbers, durations, times, and dates into standard forms. For example, phrases such as "I'd like to see" and "display every" were mapped to the term *list*, and meanings were defined for words such as "recently"—in this case defined as "within the past week." Such definitions must be provided by stakeholders at the project level. Initial mappings and definitions we created were based upon our analysis of the trace query samples we had collected.

We utilized the Stanford Parser to identify numbers, durations, times, and dates [36]. The parser returned labeled parts of speech which facilitated the extraction of relevant text and its subsequent transformation into a standardized format from which it was possible to generate SQL.

The tokenizer is responsible for identifying key terms in the simplified query. It performs this task by searching the lexicon for relation names, attribute names, attribute values, and link names. In our running example, *list* is tagged as a descriptive term. *High* is identified as a value associated with one of three possible attributes: PreliminaryHazard.riskLevel, UMLClass.name, or CodeClass.name. *riskLevel* is recognized as an attribute in the PreliminaryHazard relation. Similarly, other terms are all mapped to candidate relations, attributes, or values as depicted in Fig. 5.

4.3 Disambiguator

Many ambiguities arise as a result of the tokenization. For example, the term *class* could refer to either *Code Class* or *UML Class*. The primary task of the disambiguator is to attach each token to a single valid attribute and each attribute to a single valid relation. In cases where ambiguity exists, i.e., more than one match is viable, its job is to select the correct mapping.

Disambiguation is quite complex, and several different techniques have been used in other natural language solutions [8, 21]. TiQi adopts a toolbox of prioritized techniques and heuristics and uses them to resolve a variety of different kinds of ambiguities. The extent to which these ambiguities are correctly resolved directly impacts the correctness of the generated SQL queries. Based on our observations and analysis of sample queries and their SQL implementation, we developed an initial set of disambiguation rules. Rules were added systematically as each query was examined and were then refined and tested against other queries in the sample dataset. Each rule has the ability to filter the candidate results, but does not necessarily identify a single result; therefore, it is often necessary to apply rules sequentially until a single table, attribute, or value is identified. In future work, we plan to adopt a more probabilistic approach. We describe each of the disambiguation rules below.

4.3.1 Rule 1

No Competition If a token maps only to one table, one attribute, or one value element, then no ambiguity is present and that element is selected.

4.3.2 Rule 2

Max-Flow Popescu et al. [30] showed that a certain class of ambiguity could be definitively disambiguated through the use of the *max-flow* algorithm. Max-flow problems are set up as single-source and single-sink networks, and the goal is to maximize the flow through the network. For query disambiguation purposes, a graph is created to contain the candidate database values occurring in the NL trace query. The single source is the system, first-level nodes are created for each candidate value's table and column, and second-level nodes contain each value, which link to the output sink. Edges are established between the first-level nodes and their potential database values in the second level, and are assigned capacities of one. The max-flow algorithm guarantees to identify correct flow paths when conflicts occur, and it can identify when an ambiguous query that does not contain a conflict has been passed to the graph. The max-flow graph for our example query is shown in Fig. 6. All queries processed by our TiQi model are directed through the max-flow algorithm. In those cases in which it was able to resolve conflicts, the query is modified accordingly; otherwise, it is left unchanged. Max-flow, while powerful, only resolves a very small percentage of queries. Nevertheless, we retain it in our toolbox simply because it has a fast running time and guarantees to find either a **correct** solution or no solution. It therefore does no harm, while successfully resolving a small subset of conflicts.

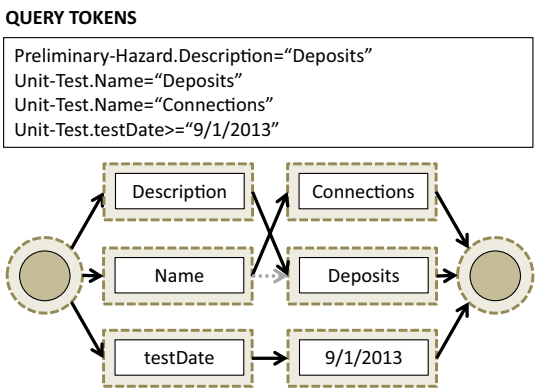


Fig. 6 Max-flow graph. *Solid black edges* indicate links, whereas *dashed gray edges* indicate that no flow exists

4.3.3 Rule 3

Pecking Order The pecking order disambiguator attempts to resolve cases in which a token could be mapped to various types of elements. If an item matches more than one element type, i.e., table, attribute, and/or a value, then it is assigned first to a table, then to an attribute, and only in the final case to a value. This particular disambiguator means that if a table and attribute (or value) share the same name, then the table will always be chosen. We discuss the ramifications of this in Sect. 5.

4.3.4 Rule 4

Compounding Evidence When an attribute or value could potentially be associated with multiple candidate tables or multiple candidate attributes, a likelihood weighting is computed based on the number of potential attribute and value matches to each table. The table with the highest total weighting (i.e., the strongest evidence) is selected. For example, it is unclear whether the term *failed* refers to "UnitTest.result" or "AcceptanceTest.result." However, the direct and unambiguous mapping of the additional query term *unitTest* to the relation *unitTest* provides compounding evidence for selecting *UnitTest.Result*.

4.3.5 Rule 5

Neighbors first If a token k_1 has been mapped to table T_1, and there are options to map a second token k_2 to either tables T_2 or T_3, and the distance from T_2 and T_3 to T_1 , respectively, is measured, and the closest table is chosen. Treating the TIM as a directed acyclic graph (DAG), distance is defined as the number of edges between two artifact types (i.e., tables).

4.3.6 Rule 6

Smaller is Better If a token matches a value found in more than one attribute, then attributes containing fields with fewer words are selected over those containing fields with more words. For example, if the token "critical" is found in a query and there are two options to map it as a value associated with a "status" attribute (with an average word count of one word per record) versus a "description" attribute (with an average word count of 20 words per record), then it is mapped to the "status" attribute.

4.3.7 Rule 7

Textual Proximity Finally, if a token cannot be otherwise disambiguated, we attempt to apply a second distance measure based on proximity in the actual query text. If a

token k_1 could be mapped to either table T_1 or T_2, then the distance between their source terms in the textual trace query is measured. For example, given the query "Show me all **test**ed elements of the **design** that address **fail**-over," assume that we need to decide whether to map "fail" to the design or test-cases table. The textual proximity rule would select *design* because it has closer proximity to the word *fail* in the sentence than *test* does.

Unless rules 1 and 2 result in complete disambiguation, we cannot guarantee that the query will be correctly interpreted. Finally, if all disambiguators have been run and ambiguities still exist, we currently randomly select one of the candidate mappings. This rarely happens; however, we plan to address it in future work by ranking multiple SQL outcomes and integrating human-in-the-loop disambiguation techniques.

Furthermore, statistical inferencing techniques can also be used in domains for which a large number of queries are available [30]. In these circumstances, it would be possible to analyze the use of phrases and terms versus the underlying intent of the query and then to infer the meaning of a query within some confidence interval. We were unable to implement statistical inferencing in our prototype implementation of TiQi as we do not yet have a sufficient quantity of trace queries; however, this approach will be integrated in future versions.

4.4 Post-Processor

The post-processor performs final transformations converting tokens such as GroupBy, Yes–No query processing, aggregations, comparisons, and joins into database entities. It takes a tokenized query structure as an input and produces a SQL query object. The SQL query object contains a collection of database entity objects for each of the standard SQL query clauses—SELECT, FROM, WHERE, and GROUP BY. Individual database entity objects refer to a single element (column or table) within the underlying relational database. The SELECT clause set contains columns that the query is returning to the user, FROM clause contains tables that the query accesses, WHERE clause contains predicates (column–predicate–constant) and join conditions (column–predicate–column), and GROUP BY clause represents the aggregation columns. The SQL query object also keeps track of additional query information such as whether an OUTER JOIN is necessary.

4.5 SQL query generation

Once conflicts have been resolved, every token in the original query is mapped to one, and only one target artifact. However, trace queries are often described only in terms of their end points even though the actual traceability path flows through a set of intermediate artifacts. To create an executable query, we need to explicitly identify this path and to inject appropriate WHERE clauses into the SQL query. We conduct a modified breadth-first search to identify the complete set of trace paths between source and target artifacts—excluding paths with loops. It is important to note that while all of our current TIMs have only a single path between each pair of artifacts, there are sometimes valid reasons for redundant paths to appear in a TIM [25]. For example, a small set of critical requirements might be traced via state transition diagrams to code, while the remaining requirements are traced directly to code. The breadth-first search is feasible because of the relatively small size of the TIM. In TIMs with no redundant trace paths, a faster approach, such as Dijkstra's shortest path algorithm, can be used [6]. In the case that multiple candidate paths are found, TiQi needs to merge results; however, this feature is not yet implemented. Once the path or paths have been identified and results merged, the formal SQL query is generated as depicted in Fig. 5.

4.6 Query object to SQL conversion

The previous query example represents one of the simpler query types. In Fig. 7, we walk through a more complex example in which the initial NL query must be expressed through SQL aggregation. After a CountQuery token ("How many") is identified, the non-numeric column (Fault.contributingFault) is mapped to the table of origin (Fault). While aggregate functions could be applied directly to the numeric columns (e.g., numberOfFaults), for non-numeric attributes the query needs to count the number of matched rows in the table. TiQi further adds a subquery to include a count of hazards without any associated faults in the result (emulating OUTER JOIN) because "at most 2" includes the possibility of zero faults.

Table 2 summarizes the different types of query structures and the corresponding SQL form. TiQi adds necessary JOINs to the final SQL code, including the connector tables (e.g., TM_T2_T3) which are not queried by users directly. Note that we simplify join code in Table 2 by writing "T2 JOIN TM_T2_T3" instead of "T2, TM_T2_T3 WHERE T2.ID = TM_T2_T3.T2ID," which is what TiQi actually produces. The aggregate function is chosen based on the attribute's data type—for example, fault count may be expressed either as COUNT (FaultDescription) or as SUM (FaultCount). TiQi substitutes an IF operator for yes/no questions because SQL is not well suited for expressing such queries. For example, given a question "Are there any TestCases in the repository?", standard SQL would only be able to compute the COUNT of TestCases—hence, we need to add an IF condition to give a yes/no answer. Figure 7 provides an example where

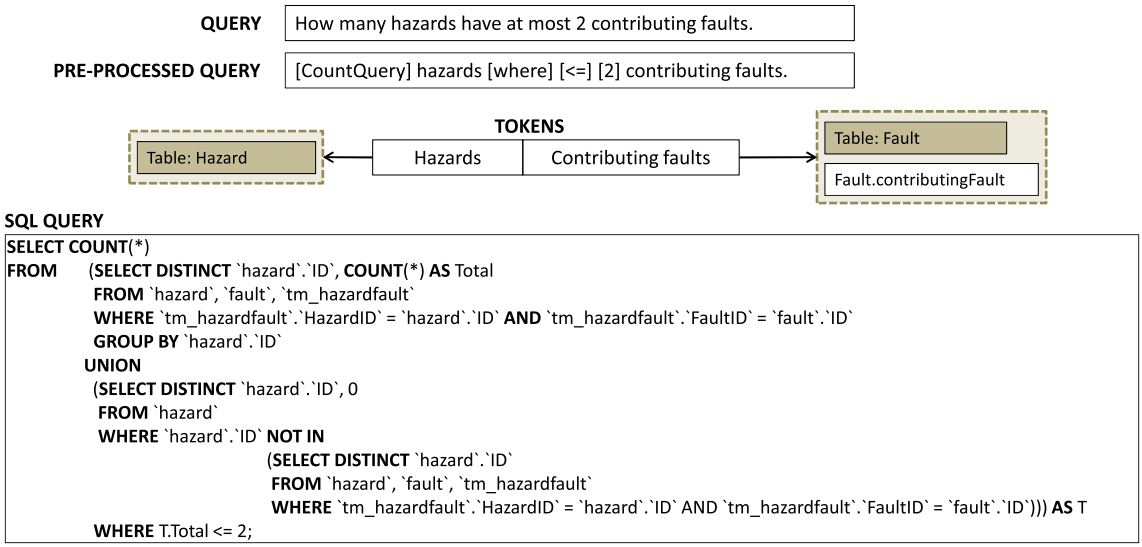


Fig. 7 Step-by-step transformation of an aggregation NL query to an executable SQL query. When an aggregate function is applied to a non-numeric column, it is mapped to the table of origin

Table 2 Heuristics for NL to SQL conversion

List all attributes from Table 1	SELECT Table1.* FROM Table1
List b, c and d attributes from T2 with T3. a = 5	SELECT T2.b, T2.c, T2.d FROM T2 JOIN TM_T2_T3 JOIN T3 WHERE T3. a = 5
Count T4.IntAttr	SELECT SUM(T4.IntAttr) FROM T4
Count T5.StrAttr	SELECT COUNT(T5.StrAttr) FROM T5
Count T6.StrAttr for T7.b	SELECT T7.b, COUNT(T6.StrAttr) FROM T6 JOIN TM_T6_T7 JOIN T7 GROUP BY T7.b
Is the count of T8.IntAttr > 100?	SELECT IF ((SELECT SUM (T8.IntAttr) AS Total FROM T8 HAVING Total > 100), 'Yes', 'No') AS Answer
Count T9 for which count T10.StrAttr is at most 3	SELECT COUNT(*) FROM (SELECT T9.ID, COUNT(*) AS Total FROM T9 JOIN TM_T9_10 JOIN T10 GROUP BY T9.ID) UNION (SELECT T9.ID, 0 FROM T9 WHERE T9.ID NOT IN (SELECT T9.ID FROM T9 JOIN TM_T9_T10 JOIN T10)) WHERE Total ≤ 3

an OUTER JOIN query would be needed—because "at most 2" contributing faults includes the possibility of 0 contributing faults. Such a query would typically be expressed using an OUTER JOIN by explicitly including hazards that do not match any contributing faults in query output. However, because automated generation of OUTER JOIN syntax is difficult, we chose to achieve the same result through a UNION of two different queries—one with an INNER JOIN and another that explicitly finds hazards which do not match any contributing faults.

5 Evaluating TiQi and its underlying concepts

We conducted four different studies to evaluate TiQi. The first was designed to explore user preference for spoken versus written natural language trace queries and to evaluate their correctness. The second directly evaluated TiQi's ability to accept a range of natural language queries and to transform them into correct SQL statements which returned the desired trace information. The third evaluated the efficacy of the disambiguators. We release two datasets used in these experiments, including TIMs, databases, and queries at http://re.cs.depaul.edu/tiqi/dataset2.html. Finally, the fourth study analyzed the distribution of new trace query terms across randomly ordered trace queries in order to investigate the stability of the domain model. The first study is reported as is from our previous conference paper [32]. The second evaluates the feature-enriched TiQi solution against more challenging datasets, and the third and fourth studies are entirely new.

5.1 Study # 1: spoken versus written queries

We designed a series of experiments to evaluate the extent to which users could effectively create trace queries using SQL, Speech, and written NL. We also explored their preferences for these three techniques. To this end, we measured both the time it took users to create various kinds of queries and also the quality of the resulting query.

One of the challenges in designing this experiment was in deciding how to elicit similar queries from users for each of the three query specification techniques. We could not simply describe all the queries in words, because this would suggest the actual solution for the NL queries. Similarly, if we prompted for SQL queries with words, and NL queries using SQL, our experiment would end up evaluating readability of one query style in tandem with writability of another query style. Furthermore, we anticipated that the NL queries would deteriorate into "verbal SQL" which was clearly not our intent. We therefore constructed nine concrete problem scenarios and asked participants to design a supporting trace query. A web-based tool was developed which displayed a TIM and a related problem scenario to the user, and prompted the user to enter or to record a trace query using one of the three query methods. The TIM contained standard artifacts including requirements, code, UML classes, acceptance tests, and test logs which any practicing Software Developer would be familiar with. None of the prompts required domain knowledge.

For example, Prompt-1 (P1) stated that "The safety officer is worried that an important requirement *R*136 is not correctly implemented. The developer tells him that it is not only implemented but has also passed its acceptance tests. The security officer runs a trace query to confirm this. What is his query?" This prompt is designed to elicit a trace query which either (1) lists all requirements which have not passed their most recent acceptance tests, (2) counts the number of requirements with failed acceptance tests, or (3) simply lists the test status of all requirements. All three of these queries incorporate information from two distinct tables in the TIM.

To reduce the bias introduced by the order in which various query types were elicited, we adopted an interleaved experimental design in which each subsequent participant was assigned to one of three groups (A, B, or C). All three groups received the prompts in the same order; however, the type of query, i.e., NL-Speech, NL-Text, or SQL, was presented in three different orders as shown in Table 3. Subsequently, by the end of the experiment, each prompt had been addressed approximately an equal number of times using each of the three techniques, and the ordering of the techniques was varied across the three groups.

Twenty-one people participated in our study. Eleven of them classified themselves as traceability experts. Of these, four used traceability in their workplace and seven were traceability researchers, well versed in creating and using trace links. The remaining participants were IT professionals, either architects, developers, or project managers, who understood the tenets of traceability but did not use it in the workplace. We provided a basic web-based tutorial on traceability which all participants were required to view before participating in the study.

5.1.1 Query creation time

The first research question evaluated whether users could specify NL queries more quickly than SQL ones. We hypothesized (H1) that the time taken to create natural language trace queries was less than that taken to create SQL trace queries. We recorded the time each participant took to view the scenario prompt and specify the query. Results were aggregated for all participants. The average time taken to create a query for each scenario using each of the three techniques was first computed, and then, the mean query creation time was calculated for each technique.

Results are displayed in Fig. 8 as a Box and Whisker graph and show that NL-Speech queries were the fastest with a mean of 56 seconds, NL-Text came next at 4 min and 47 s, and SQL was the slowest at 6 min and 2 s. A t-test failed to reveal a statistically reliable difference between the means for speech vs. written NL-text at Sig(2-tailed) value of 0.308 ($p < 0.05$). However, a second t test was conducted to compare the mean query creation time for SQL versus NL-Text and for SQL versus NL-Speech and in both cases found statistically significant differences between the two means at Sig(2-tailed) value of 0.0001 ($p < 0.05$).

We also observed that there was a large distribution in the time people took to create NL-Text queries. Although more than 75 % of the queries were created in approximately 6 min or less, a few queries took much longer. In these cases, the user had tried to describe the syntax of the SQL query instead of expressing it using natural language.

5.1.2 Query correctness

The second research question evaluated the extent to which each of the three techniques produced correct trace queries. We hypothesized that the correctness of both spoken and

Table 3 Experimental design showing the query types elicited by prompt for three different groups of participants

	P1	P2	P3	P4	P5	P6	P7	P8	P9
Group A	SQL	Txt	Sp	SQL	Txt	Sp	SQL	Txt	Sp
Group B	Sp	SQL	Txt	Sp	SQL	Txt	Sp	SQL	Txt
Group C	Txt	Sp	SQL	Txt	Sp	SQL	Txt	Sp	SQL

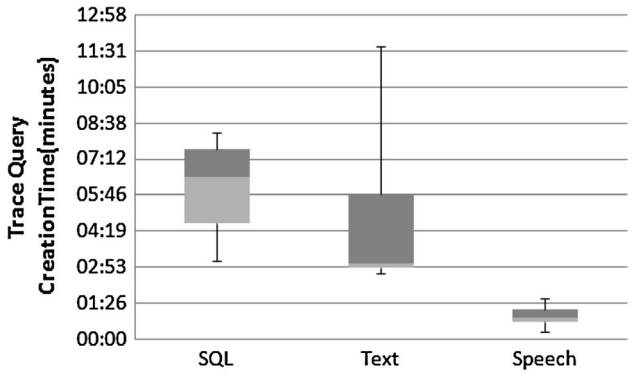


Fig. 8 Trace query creation time for each technique

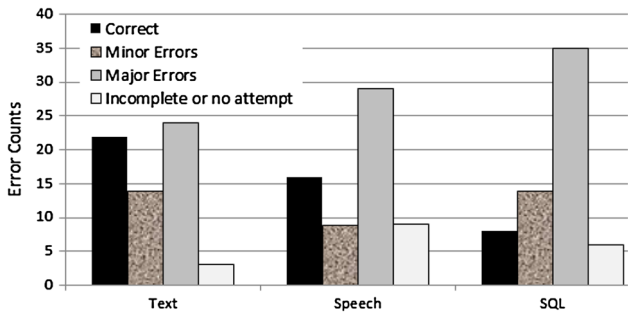


Fig. 9 Correctness of speech, text, and SQL queries

written natural language trace queries would be at least as good as that of SQL trace queries.

We evaluated the quality of the generated trace queries using a subset of the categories proposed by Jarke et al. [18] for evaluating structured natural language versus SQL in general database queries. Queries were classified into four categories of *correct*, *minor* substance error, *major* substance error, and *incomplete*. Our categories are listed below.

Correct: Statements are formulated correctly and, if executed, will return the correct results (barring other failures).

Minor Error: The query is structurally correct and all major elements are present (i.e., necessary joins, filters, etc), but there are minor syntax errors.

Major Error: The query is not structurally correct or major elements are missing (i.e., necessary joins, filters, etc).

Incomplete: Either the solution is incomplete or no attempt was made to create a query.

Each query was classified into these categories by two members of the team. For each of the three query techniques, we calculated the percentage classified into each category. As reported in Fig. 9, only eight SQL queries were categorized as completely correct compared with 16 for Speech and 22 for Text. Both SQL and Text had 14 minor errors, while Speech had only 9. Merging the two categories of *correct* and *minor error* into a "basically correct" group results in 36 basically correct for Text, 25 for Speech, and only 22 for SQL. However, it is worth pointing out that SQL is far less forgiving of minor errors than the two natural language approaches. At the other end of the spectrum, all three techniques had at least 24 of their queries categorized as having major substance errors. We provide insights into this when we discuss the users' feedback in Sect. 5.1.3.

5.1.3 User perception

We designed a posttest survey to discover the participants' perspective on the three techniques. In particular, we were interested in their preferred query technique and also in the perceived difficulty of the three approaches.

The first question asked each participant to rate the three approaches of creating trace queries in terms of difficulty on a scale of 1 to 5, with 1 being the easiest and 5 being the most difficult. Results are reported in Fig. 10 and show that in general more people found it difficult to formulate queries in SQL than in either Speech or Text. In fact, 15 of the 21 participants assigned SQL a score of four or higher on the difficulty scale, in comparison with only three for speech and one for text. Conversely, 12 people assigned a difficulty score of two or lower to Speech, 12 to text, but only two to SQL. We can therefore clearly see that in general people found both Speech and Text queries simpler to formulate than SQL ones, with a slight preference for text.

The second question asked each participant to select their preferred query technique. Overall results showed that 14 preferred NL-Text, four preferred SQL, and only three preferred speech. We also examined preferred query techniques for traceability experts versus non-experts. Interestingly, out of the 10 non-experts seven preferred Text and three preferred SQL. None of them registered a preference for Speech. In the case of the traceability experts,

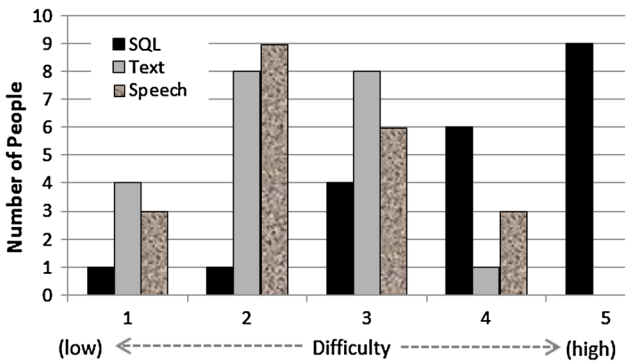


Fig. 10 User perceptions of the difficulty of issuing SQL, speech, and textual trace queries

three people out of 11 said that they preferred NL-speech, seven preferred NL-Text, and only one preferred SQL. These results are quite interesting and show strong support for the use of NL trace queries.

Feedback from participants highlighted several issues. Some people preferred written NL trace queries over spoken ones because they felt that it gave them time to plan the query more effectively. They were also concerned that the speech recognition software might not recognize their voice efficiently and that this would impact the correctness of the query. On the other hand, other participants liked the spoken approach as they felt that they did not need to worry about the finding the correct syntax to frame a query. Many participants commented that creating trace queries in SQL was very complex, especially because they needed to carefully handle the joins in order to establish links through the underlying trace matrices. On the other hand, those who preferred SQL over NL approach felt that with SQL they could state the queries more precisely without worrying about potential ambiguities.

5.2 Study # 2: natural language to SQL transformations

The second experiment we conducted focused on the ability of TiQi to transform written NL queries to executable SQL and to return correct results. We identified four different categories of queries that we classified as *solvable*, *ambiguous*, *unsupported*, and *intractable*. An *intractable* trace query is one which is unsolvable with respect to the TIM. For example, queries such as "Is the design flexible enough to accommodate new system requirements?" or "Does the code follow proper object-oriented programming practices?" address interesting software engineering questions but are not supported by underlying trace data and are therefore outside TiQi's scope. Similarly, nonsense queries such as "Dude, what's up?" are also intractable. An *ambiguous* query is one which has more than one reasonable interpretation (even to a human). An *unsupported* query is a query that we anticipate TiQi being able to answer, but for which we have not yet developed the functionality. Finally, a *solvable* query is one which is neither intractable, unsupported, nor ambiguous. It is considered correctly solved only if TiQi returns the correct information to the user.

For this experiment, we used the prototype TiQi tool we developed in Java and a web-based testing harness to evaluate the results. The current implementation of TiQi includes features such as path finding, token disambiguation, date and time recognition and standardization, definitions and synonyms, simple negation of attributes, group-by, aggregation, unions, summation, and the transformation of structured, tokenized sentences to SQL. Results were evaluated by executing the generated SQL trace queries and then inspecting both the returned data and the original SQL statements. These inspections were performed by two members of our team. In cases for which a textual query could be interpreted in more than one way, we accepted any valid interpretation of the query.

Experiments were conducted against the following two datasets—both of which are available for download at http://re.cs.depaul.edu/tiqi/dataset2.html.

5.2.1 *Isolette data set*

The Isolette dataset includes eight different artifacts, namely hazards, faults, environmental assumptions, system requirements, design, code, test cases, and test results connected through six unique trace paths. Together, these artifacts contain a total of 26 attributes in addition to IDs. The actual data were primarily extracted from a documented case study about a safety-critical Isolette system [20]. One hundred and five NL trace queries were developed. Approximately half the queries were created by five different software engineers and included a variety of non-trivial queries, many of which could only be addressed by traversing multiple artifacts and trace matrices. The remaining queries were developed by our research team. The benefit of this approach was that we were able to create a more challenging and diverse set of trace queries which served as test cases for the various transformation functions provided by TiQi.

Results were evaluated by running each trace query through TiQi and then reviewing both the generated SQL and the produced data result. Each result was marked as either correct or incorrect. We also report results for the subset of queries for which all necessary TiQi functions have been implemented. We refer to these as *supported* queries.

Of the 105 queries, we did not consider any to be *intractable* although several were *ambiguous* even to a human. Eighty-seven individual queries were transformed correctly, and of the remaining 18, seven were due to failures of the transformation algorithm and 11 were due to unsupported constructs. These results are reported in Table 4.

5.2.2 *Easy-clinic*

The Easy-Clinic TIM includes seven different artifacts, namely HIPAA goals, requirements, design, code, unit and acceptance test cases and test logs connected through six unique trace paths. Together, these artifacts contain a total of 17 attributes in addition to IDs. Functional requirements, code classes, and test cases were derived directly from Easy-Clinic data (available from CoEST.org). HIPAA

Table 4 Results for Isolette dataset

All queries	Correct	Incorrect	% Correct
Supported queries	87	7	92.6
All queries	87	18	82.9

Table 5 Results for Easy-Clinic dataset

	Correct	Incorrect	% Correct
All data			
Supported queries	48	37	56.4 %
All queries	48	52	48.0 %
Reduced text*			
Supported queries	62	22	73.8 %
All queries	62	38	62.0 %

* The reduced text subset of the data included ALL records, but fields with lengthy descriptions such as test cases and code classes were reduced to titles only

goals were taken from the US HIPAA Technical Safeguards. All other artifacts were created by one of the researchers for purposes of the project. As Easy-Clinic contains a mix of English and Italian terms, we translated Italian terms to English. Hundred NL trace queries were developed using the same protocol as the Isolette dataset (accuracy results are summarized in Table 5).

Results for the Easy-Clinic dataset are less stellar than for Isolette. Out of 100 queries, TiQi transformed only 48 correctly at an accuracy rate of 48.0 %. Of the 52 incorrect queries, 16 of them were due to unsupported constructs and the remaining 36 due to failures in the TiQi algorithm. An analysis of the queries that failed for algorithmic reasons revealed that many of the problems were caused by unresolvable ambiguities introduced as a result of long textual descriptions in test cases and code which had been added to the dataset for these experiments. To test our informal hypothesis that the lengthy data fields were causing the problem we created a modified version of Easy-Clinic (named "Reduced Text") and re-executed the queries. For this reduced dataset, TiQi correctly transformed 62 of the 100 queries at accuracy of 62 %. Further analysis showed that 46 of the queries were correct in both cases (i.e., for both the full data and reduced text datasets), two were correct only when all data were present, 16 were correct only with the reduced text, and 21 were never correct. These results confirmed our hypothesis that TiQi currently does not perform well on textually rich data. To address this problem, future versions of TiQi need to integrate techniques for summarizing lengthy text fields so that only the most important terms are retained. However, this solution was out of scope for the current paper.

5.3 Study # 3: disambiguator efficacy

Our third study explored the efficacy of TiQi's ambiguity resolution process. This is invoked only if ambiguities exist. TiQi performs disambiguation at four different levels which we refer to as *basic*, *table*, *attribute*, and *value*. Each level of resolution calls upon one or more of the disambiguators described in Sect. 4.3 of this paper.

Basic resolution occurs when a query token is potentially mapped to different database entities, i.e., to tables, attributes, or actual values. For example, the token *Hazard* could be mapped to the table name or to an attribute. The *pecking order* disambiguator maps it to the table name. The task is to identify either a table name or an attribute within a table.

Table resolution occurs when a query token maps to two or more potential tables. In these circumstances, the *compounding evidence* disambiguator is called, and then, if a conflict remains, the *neighbors first* is called. The task is to identify the correct table.

Attribute resolution occurs when a query token maps to two or more column names. In this case, *compounding evidence* and *neighbors first* disambiguators are again invoked. The task is to identify the correct attribute.

Value resolution occurs when a query token maps to words contained in two or more distinct attributes. The task is again to identify the correct attribute, but this time based upon its associated data. Resolution involves first invoking the *smaller is better* disambiguator, and then, if necessary, the *word proximity* disambiguator.

In this experiment, we evaluated the frequency with which each of these four resolution "engines" was invoked, whether the resulting query was processed correctly, and finally what happened to the relevant queries if the engine were turned off.

We report results in Table 6. The first three columns depict the resolution level, disambiguators used, and the dataset. The next three depict the number of queries for which the resolution was applied and returned correct or incorrect results, respectively. The next column marked "only correct if triggered" represents the case in which queries which were correct when the resolution was used and were incorrect without it. Finally, the remaining column, labeled "Only correct if omitted" reports the number of queries which were harmed by the use of the resolution (i.e., correct without it but incorrect with it).

These results show that the basic resolver was activated in the majority of queries for both datasets (i.e., 86/105 for Isolette and 75/101 for Easy-Clinic), followed by data-value level and then table-level disambiguation. Attribute-level disambiguation occurred infrequently, and in the case of Easy-Clinic, it failed to resolve any ambiguities. These findings suggest that future efforts should be focused on

Table 6 Efficacy of the various resolution levels. Numbers represent the number of queries

Resolution Level	Disambiguators	DataSet	Queries Applied			Only Correct if triggered	Only Correct if omitted
			Total	Correct	Incorrect		
Basic	Pecking order	Isolette	86	70	16	29	0
		Easy-Clinic	75	45	30	16	5
Table level	Compound Evid.	Isolette	18	15	3	6	0
	Neighbors 1st	Easy-Clinic	12	9	3	6	0
Attribute level	Compounding Evid.	Isolette	2	2	0	0	0
	Neighbors 1st	Easy-Clinic	3	0	3	0	0
Data value level	Smaller is better	Isolette	44	39	5	31	0
	Term Prox.	Easy-Clinic	52	26	26	23	0

improving TiQi's ability to differentiate between different types of artifacts (i.e., the basic resolver) and between different data values. The reason that the attribute-level resolver was rarely triggered is that attribute names which repeat across tables tend to be more generic in nature such as *ID* and *description*, while the trace queries created by our users tended to only reference more unique attribute names. It is also interesting that the only disambiguator which actually harmed the query (i.e., "only correct if omitted") was the *pecking order*. An analysis of the failed queries suggests that the problem was caused by the fact that *pecking order* fails if a table and attribute share the same name—and the correct choice should have been the attribute. The disambiguator is overly strict, and the problem will be addressed in future work through exploring probabilistic solutions. Care needs to be taken in interpreting these results because generating a correct query requires a choreography of resolution techniques.

5.4 Study # 4: trace query concepts

The final study was designed to explore the stability of concepts in the trace query domain model. The underlying research question addresses the question of whether our collection of trace queries is sufficient for constructing a relatively complete model. We recognized that this is unlikely and so designed the experiment to understand the growth and stability of concepts in relation to the number of sample trace queries.

First, the concepts such as *give me*, *associated with*, and *without* found in each query were tagged. A random ordering of queries was then generated for each of the two query sets (i.e., queries collected for Isolette and for Easy-Clinic), and the first occurrence of each concept was identified. For example, the term *associated with* might have first occurred in the ninth query in the randomly ordered list. We then divided the ordered queries into blocks of 10 queries and counted the number of concepts

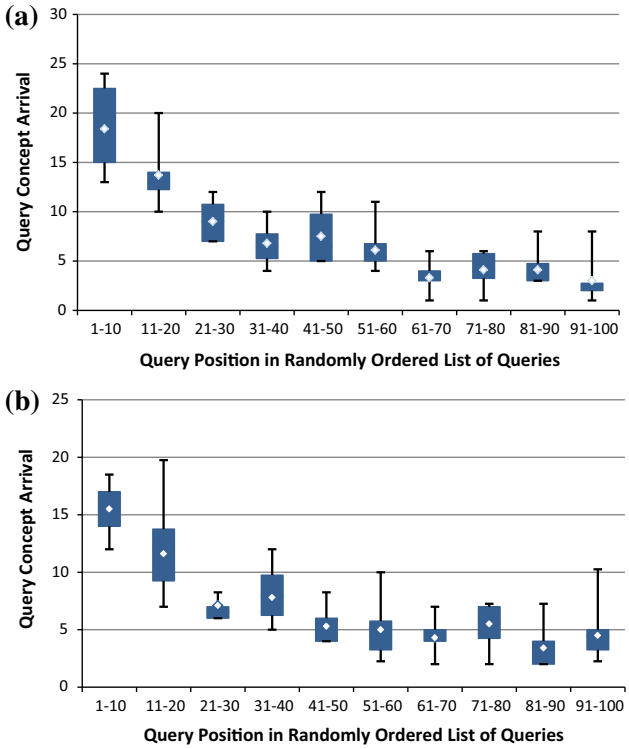


Fig. 11 Arrival query terms and stability of the domain model **a** Isolette data and **b** Easy-Clinic data

which appeared for the first time in each block. This process was repeated 10 times with different random orderings of the queries. Results are reported in Fig. 11 and show that there is an initial rapid decrease in the number of new concepts as the size of the query set increases. On the other hand, new concepts continue to arrive even in the last blocks of sample queries.

We also examined the similarity and differences between query concepts in the two query sets and found only about 50 % overlap. In several cases, the query concepts were similar to each other, while in other cases we noticed

that different artifacts in the TIM triggered the use of different query terms.

These results suggest that while the model is expected to eventually stabilize, we need to significantly increase our sample size. Furthermore, we need to integrate techniques for resolving unknown terms and dynamically growing our domain model.

6 Threats to validity

External validity evaluates the ability of the approach to generalize well out of samples used in the experiments. One threat to validity is introduced by the fact that we only evaluated our approach against two different TIMs and a limited set of trace queries. While our TIMs were non-trivial in size and complexity, they did not contain multiple hierarchies of systems and subsystems which is common in some large system engineering projects. On the other hand, the TIMs represented a variety of artifact types and attributes and were representative of numerous software projects. A second threat is introduced by the fact that our vocabulary and rules were built from a relatively small sample of trace queries. However, we have performed an analysis showing the growth rate of query concepts associated with new queries. It is clearly important to extend the sample size in the future so that we can construct a more robust transformation process. Furthermore, the fact that Isolette which contains simpler data (i.e., fewer long descriptions) outperformed Easy-Clinic suggests the need to further improve TiQi's ability to process free-form text. Finally, TiQi is a work in progress and certain features, such as complex negation, are not yet implemented. While our TiQi model is designed to accommodate complex queries, further evaluation is needed before we can claim its ability to support a broad collection of trace queries.

Construct validity refers to whether the dependent and independent variables are suitable for evaluating the hypothesis, and therefore of answering the stated research questions. In our user study, the primary independent variable was the query method, i.e., spoken NL, written NL, or SQL, while, in two of the evaluations, the dependent variables were based on a rubric and involved human assessment. To mitigate bias, we carefully defined rules for placing queries into each of the categories and these rules were systematically followed. Evaluating the correctness of each query was relatively simply, and we cross-checked results by examining both the generated SQL and also the produced data. One of the researchers is a database instructor and well experienced in assessing SQL. Finally, our evaluation of the disambiguators looked only at individual disambiguators and in certain cases, pairs of disambiguators.

Finally, internal validity reflects the extent to which a study minimizes systematic error or bias, so that a causal conclusion can be drawn. We mitigated systematic error in our user study by randomly assigning participants to groups and then adopting an interleaving approach in which different groups were given tasks in different orders for each of the three TIMs; however, our study included only 21 participants who may not have been fully representative of the general IT community. Similarly, in our evaluation of growth in the domain model, we repeated each analysis 10 times on various random orderings of the queries.

7 Related work

Related work falls under the two areas of trace query techniques and natural language database queries. We discuss each of these.

7.1 Trace query techniques

Trace queries can be issued in a number of different ways. Maeder et al. [24] developed the visual trace modeling language (VTML) which represents queries as a set of filters applied to a structural subset of the TIM. A VTML query is composed of a connected subset of the artifacts and trace types defined in the TIM, as well as a set of associated filter conditions. These filters are used to eliminate unwanted artifacts and to define the data to be returned by the trace query. Studies conducted with human analysts showed that VTML queries are easier to read and to write than SQL queries. One of their primary advantages is that they allow the information of the trace matrices to be abstracted away, so that the human user can specify most queries in terms of visible elements of the TIM. Störrle [37] presented the visual model query language (VMQL), which is similar to, but less expressive than VTML.

Maletic and Collard [26] 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. [40] 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. [13] build models that describe how modeling languages are interrelated. They transform

existing traceability data within a model into a different representation (e.g., a table) against which standard trace queries can be issued. Their approach also requires advanced skills.

In addition to textual approaches such as SQL, graphical query languages have been proposed and commercially offered in the database domain for a long time. The PICASSO approach by Kim et al. [19] represents one of the earliest graphical query languages and was built on top of the universal relation database system System/U. Visual SQL by Jaakkola et al. [16] translates all features of SQL into a graphical representation similar to entity relationship models and UML class diagrams. Furthermore, a variety of commercial and open-source tools provide graphical support for the specification of queries (e.g., Microsoft Visual Studio™, Microsoft Access™, Active Query Builder, and Visual SQL Builder).

However, all of these techniques, whether designed specifically for tracing purposes, or for more general database queries require some degree of technical expertise. In contrast, TiQi is designed to accept queries from stakeholders who have no formal technical training in either UML or SQL. The underlying trace query domain model and the TiQi engine is designed to translate higher-level concepts and domain-specific jargon into executable trace queries.

7.2 Natural language database queries

There are several commercial products that claim to provide NL database interfaces. For example, PrimeQue allows a user to interact with a database by following a series of prompts. The user first builds a semantic map of the dataset, and the tool then uses this map to help the user construct a NL query using a query-by-example style interface [41]. However, such solutions severely constrain the language of the query and therefore do not support truly natural language queries. Other general NL query techniques are primarily limited to the language extracted from the structure of the database, i.e., its table names and attribute names, and therefore are also rather restrictive [17]. Recently, tools such as *EasyAsk Quiri* and Microsoft's *Power BI for Office 365* have demonstrated the viability of using speech interfaces to issue business intelligence queries against database schemas.

C-Phrase, developed by Minock et al. [29], allows users to specify queries using a natural language front end and then to execute those queries against a relational database. C-Phrase provides support for customization for a specific domain by following a *name-tailor-define* cycle. *Naming* involves providing names to classes, attributes, and join relationships. For tracing purposes, these are already specified in the TIM. *Tailoring* allows domain-specific patterns to be specified and matched. For example, if we applied this to the traceability domain, then the phrase "Is safe for use" might be associated with "all hazards mitigated" or "all mitigating requirements addressed." This phase involves analyzing domain-related phrases and identifying matches between them. Finally, in the *define* step, the user provides definitions of terms, for example, the word "recent" might be defined to mean "in the past week." While C-Phrase provides support for this process, it is generally understood that customizing a model for a specific domain can be extremely time-consuming [2]. Nevertheless, we believe the effort is worthwhile for the tracing domain, simply because, once created, it will be applicable across an enormously wide spectrum of software and systems engineering projects. Many of the customization concepts from C-Phrase were adopted in TiQi. However, the C-Phrase tool is no longer compilable due to obfuscation issues.

Another interesting approach proposed by Meng [27] integrates information from the enterprise, database values, use words, and query cases. The novelty of this approach is that it directly stores entire sample queries and then, instead of attempting to analyze the individual parts of the NL query and to formulate an executable SQL query, it simply finds a match in the large database of stored queries and issues the associated stored query. Query cases are somewhat similar to the phrases identified in Minock's "tailor" stage. The approach we adopted for TiQi integrates concepts from both Minock's and Meng's techniques.

8 Conclusion

In this paper, we have extended our previous work on TiQi as a solution for transforming natural language trace queries into executable SQL [32]. We have demonstrated the potential that TiQi has for making traceability data accessible through allowing project stakeholders to express trace queries using their own words. Our first experiment showed that users have a preference for written rather than spoken queries; however, our approach is able to support both techniques. We expect that as we continue the inevitable move toward mobile devices, speech options for interacting with software engineering tools such as TiQi will become increasingly attractive.

The quality of trace links generated against the two datasets showed rather mixed results. For Isolette, TiQi was able to return high accuracy by correctly generating dataset and show high accuracy 92.6 % of the supported queries and 82.9 % of unsupported all queries. Our current development process is expected to deliver the missing features needed to fill this gap. For Easy-Clinic, results were

less than stellar. TiQi was only able to correctly generate 73.8 % of supported queries and 62.0 % overall. Furthermore, these results were only achieved after the textually rich fields were reduced to include only header data.

However, these results suggest opportunities for further research. First, we will investigate techniques for merging solutions for querying structured and unstructured text. There is a vast body of work in the area of natural language information retrieval which we intend to explore and to integrate into our more structured approach. Second, we will investigate different orderings of our disambiguators and more sophisticated techniques based on probabilistic and machine learning approaches. Third, we will explore visualization techniques to engage users in the disambiguation process and to provide full accountability for how TiQi interpreted the NL query and how the SQL transformation was achieved. Without such techniques in place, TiQi users could not be confident in the results.

Finally, in future work, we plan to collect trace queries from practitioners working in a wide variety of software engineering domains so that we can construct a more extensive trace query domain model, with a more complete set of disambiguators and transformation rules.

Acknowledgments The work described in this paper was partially funded by NSF Grant CCF-1319680.

References

1. Ali N, Guéhéneuc Y-G, Antoniol G (2013) Trustrace: mining software repositories to improve the accuracy of requirement traceability links. IEEE Trans Softw Eng 39(5):725–741
2. Androutsopoulos I, Ritchie G (2000) Database interfaces. In: Handbook of natural language processing, Marcel Dekker Inc. pp 209–240
3. Cleland-Huang J, Gotel O, Hayes JH, Mäder P, Zisman A (2014) Software traceability: trends and future directions. In: Proceedings of the on future of software engineering, FOSE 2014, Hyderabad, India, May 31–June 7, 2014, pp 55–69
4. Cleland-Huang J, Heimdahl MPE, Hayes JH, Lutz RR, Maeder P (2012) Trace queries for safety requirements in high assurance systems. In: Proceedings of requirements engineering: foundation for software quality–18th international working conference, REFSQ 2012, Essen, Germany, March 19–22, 2012, pp 179–193
5. Collaborative lifecycle management: design, test, analyze, develop, deliver, integrated by design, Accessed 9/6/2013
6. Cormen T, Leiserson C, Rivest R, Stein C (eds) Introduction to algorithms (2nd ed.). MIT Press and McGrawHill (2001)
7. Czauderna A, Cleland-Huang J, Çinar M, Berenbach B (2012) Just-in-time traceability for mechatronics systems. In: Second IEEE international workshop on requirements engineering for systems, services, and systems-of-systems, RESS 2012, Chicago, IL, September 25, 2012, pp 1–9
8. Frost RA, Amour BS, Fortier RJ (2013) An event based denotational semantics for natural language queries to data represented in triple stores. In: 2013 IEEE seventh international conference on semantic computing, Irvine, CA, September 16–18, 2013, pp 142–145
9. Göker MH, Thompson CA, Arajärvi S, Hua K (2007) Connecting people with questions to people with answers. KI 21(4):23–26
10. Gotel O, Cleland-Huang J, Huffman Hayes J, Zisman A, Egyed A, Grnbacher P, Dekhtyar A, Antoniol G, Maletic J, Mder P (2012) Traceability fundamentals. In: Cleland-Huang J, Gotel O, Zisman A (eds) Software and systems traceability. Springer, London, pp 3–22
11. Gotel OCZ, Finkelstein A (1994) An analysis of the requirements traceability problem. In: Proceedings of the first IEEE international conference on requirements engineering, ICRE '94, Colorado Springs, Colorado, April 18–21, 1994, pp 94–101
12. Gouvêa E, Moreno-Daniel A, Reddy A, Chengalvarayan R, Thomson DL, Ljolje A (2013) The at&t speech API: a study on practical challenges for customized speech to text service. In: INTERSPEECH 2013, 14th annual conference of the international speech communication association, Lyon, France, August 25–29, 2013, pp 2071–2073
13. Guerra E, de Lara J, Kolovos D, Paige R (2010) Inter-modelling: from theory to practice. In: Petriu D, Rouquette N, Haugen Ø (eds) Model driven engineering languages and systems, volume 6394 of lecture notes in computer science. Springer, Berlin, pp 376–391. doi:10.1007/978-3-642-16145-2
14. Hayes JH, Dekhtyar A, Sundaram SK, Holbrook EA, Vadlamudi S, April A (2007) Requirements tracing on target (retro): improving software maintenance through traceability recovery. ISSE 3(3):193–202
15. Huang X, Baker J, Reddy R (2014) A historical perspective of speech recognition. Commun ACM 57(1):94–103
16. Jaakkola H, Thalheim B (2003) Visual sql: high-quality er-based query treatment. In: Jeusfeld M, Pastor Ó (eds) Conceptual modeling for novel application domains, volume 2814 of lecture notes in computer science. Springer, Berlin, pp 129–139. doi:10.1007/978-3-540-39597-3
17. Jarke M, Krause J, Vassiliou Y (1986) Studies in the evaluation of a domain-independent natural language query system. In: Cooperative interfaces to information systems. Springer, pp 101–130
18. Jarke M, Krause J, Vassiliou Y, Stohr EA, Turner JA, White NH (1985) Evaluation and assessment of a domain-independent natural language query system. IEEE Database Eng Bull 8(3):34–44
19. Kim H-J, Korth HF, Silberschatz A (1988) PICASSO: a graphical query language. Softw Pract Exp 18:169–203
20. Lempia DL, Miller SP (2009) Requirements engineering management handbook. National Technical Information Service (NTIS)
21. Li Y, Yang H, Jagadish HV (2006) Term disambiguation in natural language query for XML. In: Flexible query answering systems, 7th International Conference, FQAS 2006, Milan, Italy, June 7–10, 2006, Proceedings, pp 133–146
22. Lin J, Lin CC, Cleland-Huang J, Settimi R, Amaya J, Bedford G, Berenbach B, Khadra OB, Duan C, Zou X (2006) Poirot: a distributed tool supporting enterprise-wide automated traceability. In: 14th IEEE international conference on requirements engineering (RE 2006), 11–15 September 2006. Minneapolis/St. Paul, Minnesota, pp 356–357
23. Lucia AD, Fasano F, Oliveto R, Tortora G (2010) Fine-grained management of software artefacts: the ADAMS system. Softw Pract Exp 40(11):1007–1034
24. Mäder P, Cleland-Huang J (2013) A visual language for modeling and executing traceability queries. Softw Syst Model 12(3):537–553
25. Mäder P, Jones PL, Zhang Y, Cleland-Huang J (2013) Strategic traceability for safety-critical projects. IEEE Softw 30(3):58–66
26. Maletic JI, Collard ML (2009) 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, IEEE Computer Society, pp 16–20
27. Meng HH, Siu KC (2002) Semiautomatic acquisition of semantic structures for understanding domain-specific natural language queries. IEEE Trans Knowl Data Eng 14(1):172–181
28. Miller GA, Fellbaum C (2007) Wordnet then and now. Lang Resour Eval 41(2):209–214
29. Minock M (2010) C-phrase: a system for building robust natural language interfaces to databases. Data Knowl Eng 69(3):290–302
30. Popescu A-M, Etzioni O, Kautz HA (2003) Towards a theory of natural language interfaces to databases. In: IUI, pp 149–157
31. Post from HP Quality Center Support Forum (2009) http://h30499.www3.hp.com/t5/ITRC-Quality-Center-Forum/traceability-query/m-p/4505026., Accessed 1/18/2015
32. Pruski P, Lohar S, Aquanette R, Ott G, Amornborvornwong S, Rasin A, Cleland-Huang J (2014) Tiqi: Towards natural language trace queries. In: IEEE 22nd international requirements engineering conference, RE 2014, Karlskrona, Sweden, August 25–29, 2014, pp 123–132
33. Rempel P, Mäder P, Kuschke T, Cleland-Huang J (2014) Mind the gap: assessing the conformance of software traceability to relevant guidelines. In: 36th international conference on software engineering (ICSE)
34. Rempel P, Mäder P, Kuschke T, Philippow I (2013) Requirements traceability across organizational boundaries - a survey and taxonomy. In: Proceedings of requirements engineering: foundation for software quality: 19th international working conference, REFSQ 2013, Essen, Germany, April 8–11, 2013, pp 125–140
35. Shwartz SP (1982) Problems with domain-independent natural language database access systems. In: Proceedings of the 20th annual meeting on association for computational linguistics, ACL '82, Stroudsburg, PA. Association for Computational Linguistics,pp 60–62
36. Silveira N, Dozat T, de Marneffe M, Bowman SR, Connor M, Bauer J, Manning CD (2014) A gold standard dependency corpus for english. In: Proceedings of the ninth international conference on language resources and evaluation (LREC-2014), Reykjavik, Iceland, May 26–31, 2014, pp 2897–2904
37. Störrle H (2011) VMQL: a visual language for ad-hoc model querying. J Vis Lang Comput 22(1):3–29
38. Vassiliou Y, Jarke M, Stohr E, Turner J, White N (1983) Natural language for database queries: a laboratory study. MIS Q 7(4):47–61
39. Winkler S, von Pilgrim J (2010) A survey of traceability in requirements engineering and model-driven development. Softw Syst Model 9(4):529–565
40. Zhang Y, Witte R, Rilling J, Haarslev V (2006) 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
41. Zloof M. Query by example. In: Proceedings of the NCC. AFIPS