TiQi: Towards Natural Language Trace Queries

Piotr Pruski, Sugandha Lohar, Rundale Aquanette, Greg Ott, Sorawit Amornborvornwong,
Alexander Rasin, and Jane Cleland-Huang
School of Computing
DePaul University, Chicago, IL, 60604, USA
ppruski@gmail.com, slohar@cs.depaul.edu,arasin@cs.depaul.edu, jhuang@cs.depaul.edu

Abstract—**One of the surprising observations of traceability in practice is the under-utilization of existing trace links. Organizations often create links in order to meet compliance requirements, but then fail to capitalize on the potential benefits of those links to provide support for activities such as impact analysis, test regression selection, and coverage analysis. One of the major adoption barriers is caused by the lack of accessibility to the underlying trace data and the lack of skills many project stakeholders have for formulating complex trace queries. To address these challenges we introduce TiQi, a natural language approach, which allows users to write or speak trace queries in their own words. TiQi includes a vocabulary and associated grammar learned from analyzing NL queries collected from trace practitioners. It is evaluated against trace queries gathered from trace practitioners for two different project environments.**

Index Terms—**Traceability, Queries, Speech Recognition, Natural Language Processing**

I. Introduction

Software traceability is a critical component of the software engineering process, especially in regulated industries such as avionics, medical devices, and transportation [14]. As a result, developers often invest significant cost and effort creating trace links in order to meet certification or compliance requirements. Unfortunately, despite the significant investment, trace links are often underutilized for supporting activities such as impact analysis, coverage analysis, and test regression selection. There are many possible reasons for this. In some cases, trace links are created late in the software development life-cycle process and are therefore not available for use during earlier development phases [14]. In other cases, lack of tool support, poor understanding of the underlying trace schema, or lack of skills needed to formulate useful trace queries all contribute to making trace links inaccessible to project stakeholders [25], [12]. In particular, there are several documented examples of practitioners who have struggled to generate adequate trace queries [12], either because they lack proficiency in SQL or XQuery, or simply because the available trace information is ill-defined and inaccessible [14].

The problem is exacerbated by the fact that trace queries are often quite complex and cut across many different artifacts. For example, the "producibility" report commissioned by the US Department of Defense [19], and released in 2010, identified traceability as a critical challenge that needs to be addressed in order to provide better support for ongoing evolution of software across an extensive set of artifact types that include requirements, architectural documents, design, code, and test cases. Our goal is to make traceability information easily accessible to project stakeholders by developing a natural language (NL) interface and supporting tool which allows users to express complex queries using both written and spoken NL.

The idea of NL database queries is far from new. NL query solutions have been developed at differing levels of success from as far back as the 1970s and 80s [24]. While the field stagnated for many years as researchers refocused on developing NL interfaces for less structured sources of information [4], there are several factors which make the database interface problem a compelling one to revisit for traceability purposes today.

First, raw trace data is becoming increasingly available in software projects. There are a number of reasons for this including the fact that regulated industries insist on traceability [14], integrated development environments such as Jazz produce links as a natural byproduct 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 [6], [1]. Second, the current ubiquitous move towards mobile computing creates a compelling reason to invest in more flexible and accessible trace usage methods, especially voice activated ones. Third, advances in speech recognition software make speech interfaces a viable option, and finally, 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 [24], suggesting that NL solutions could produce a viable solution to the trace usage adoption barrier.

While a variety of NL approaches exist for issuing general database queries, it is widely accepted that for an NL query language to be effective, it must be supported by a domain specific model [16]. To the best of our knowledge nobody has yet attempted to discover the vocabulary and grammar of queries in the traceability domain or to build an associated NL interface. To address this gap, we present TiQi, a framework and tool supported by a traceability domain model and algorithms capable of transforming naturally worded trace queries into executable SQL statements. The primary contributions of this paper are therefore twofold. First we make advances in

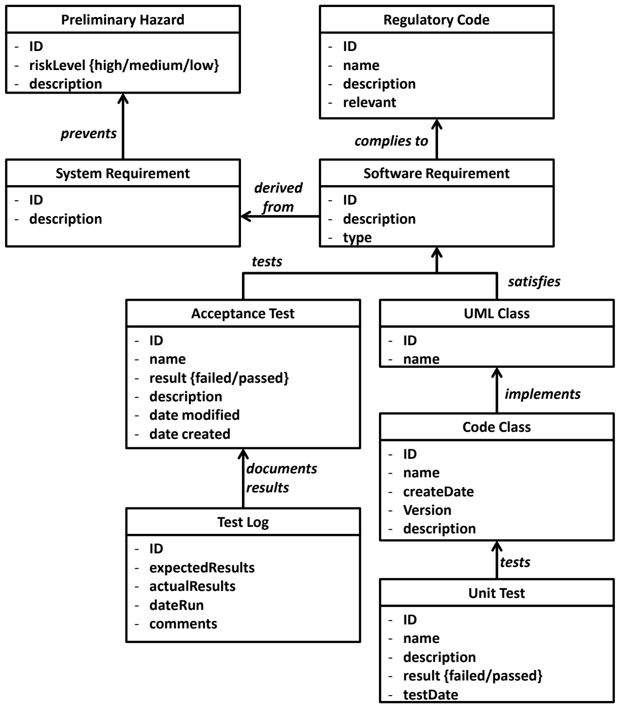


Fig. 1: A Sample Traceability Information Model

learning the vocabulary of a domain model for supporting natural language trace queries, and second we design and implement a NL processing engine capable of transforming NL trace queries into relevant SQL statements.

The remainder of this paper is laid out as follows. Section II discusses existing approaches for issuing trace queries and presents existing NL database query techniques. Section III presents our technique for constructing a domain model, and Section IV describes our TiQi process for transforming a NL query to SQL. In Section V we describe a series of experiments that were conducted to evaluate TiQi. Section VI describes threats to validity, and finally Section VII discusses our findings and proposes future enhancements to TiQi.

II. Towards Natural Language Trace Queries

In practice, trace queries are issued against an available dataset of artifacts and trace links. Ideally these are documented in a *Traceability Information Model* (TIM) [14] where *artifact types* and their properties are represented as classes and attributes, and *permitted trace links* are represented as semantically typed associations. A TIM can be 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 [21].

Furthermore, the TIM provides useful support for trace queries, as it clearly depicts the artifacts and trace links available for querying purposes. For example, the TIM in Figure 1 allows a project stakeholder to 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."

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:

```
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;
```

Furthermore, a typical software project stores artifacts in a variety of file formats, databases, and proprietary case tools. We therefore cannot assume that artifacts and trace links are nicely stored in a database, and neither can we assume that a complete and correct set of trace links are available. Instead, a traceability environment is more likely to involve a layered approach [13] in which the TIM serves as a logical schema against which all traceability queries are specified; however the elements of the TIM are mapped to physical sources, and executing a query requires the additional step of dynamically retrieving data from those sources in order to populate the database. Tools such as Poirot and RETRO [6] which utilize trace-retrieval techniques to generate candidate trace links upon demand could also potentially be used to service the SQL trace queries generated by TiQi.

A. Related Work: Trace Query Techniques

Trace queries can be issued in a number of different ways. Maeder et al. developed the Visual Trace Modeling Language (VTML) which represents queries as a set of filters applied to a structural subset of the TIM [12]. A VTML query is therefore 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 or to define the data to be returned by the trace query. Studies conducted with human analysts showed that VTML queries were easier to read and to write than SQL queries. One of their primary advantages is that they allowed the information of the trace matrices to be abstracted away, so that the human user could specify most queries in terms of visible elements of the TIM. Störrle [23] presented the Visual Model Query Language (VMQL), which is similar to, but less expressive than VTML.

Maletic and Collard [15] 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. [26] 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. [5] build models that describe how modeling languages are inter-related. 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.

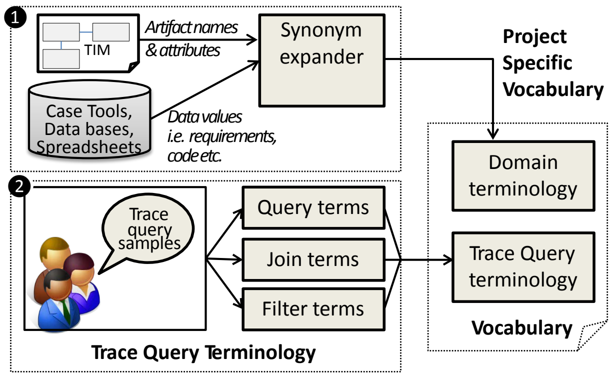
In addition to textual approaches such as SQL, graphical query languages have been proposed and commercially offered in the database domain for a long time. The PICASSO approach by Kim et al. [10] represents one of the earliest graphical query languages and was built on top of the universal relation database system System/U. Visual SQL by Jaakkola et al. [7] 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^{TM}, Microsoft Access^{TM}, 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.

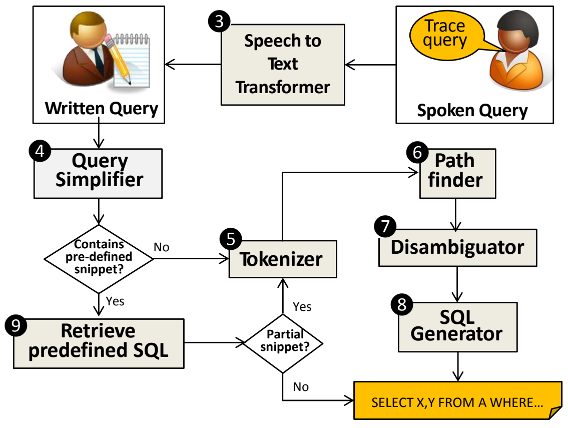
B. Related Work: Natural Language Queries

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 [27]. 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, attribute names etc, and therefore are also rather restrictive [8]. 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.

In a seminal discussion on the notion of general versus domain specific NL query languages [22], Shwartz provided an



(a) Domain specific vocabulary is extracted from the Traceability Information Model and associated data, while traceability jargon is identified in advance and reused across projects.



(b) A Natural Language Query initiated either as speech or in written form is transformed into SQL

Fig. 2: The TiQi Process

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 depth, the third to map depth, and the fourth to well number. A general database system would have little hope of understanding these concepts unless units of measure were explicitly stated. Shwartz therefore claims that conceptual domain-specific knowledge is critical for understanding the nuances of queries.

C-Phrase, developed by Minnock et al., allows users to specify queries using a natural language front end and then to execute those queries against a relational database [18]. 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, we were unable to deploy the C-Phrase tool due to obfuscation issues.

Another interesting approach proposed by Meng [17] 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 merges concepts from both Minock's and Meng's techniques.

III. Building a Domain Model

The domain model for a project is derived from three different sources as depicted in Figure 2a. The first is the structure of the TIM, the second is the underlying traceable data, and the third is the language used to express queries in the domain. The terminology in the TIM and the data compose the project specific vocabulary formulated from artifact names and attribute names. In our current model we do not assume that links will be labeled as we rarely see this in practice; however adding this additional information in future versions of TiQi could enhance its accuracy.

A. Project Specific Vocabulary

To identify the basic vocabulary needed to tokenize a NL query into a set of relations, attributes, values, query terms, filters, join commands, negations, and aggregations, a project specific vocabulary is constructed by parsing the text in the TIM and its underlying artifacts. This produces a list of relation names, attributes, and link names extracted from the meta-data of the TIM, and also a list of values extracted from the data in each of the tables. Each value is associated with one or more attributes. In the sample TIM presented in Figure 1, vocabulary items would include terms such as *preliminary hazard* extracted as a table name, *dateRun* extracted as an attribute, *medium* or *armController* extracted as attribute values, and *tests* or *prevents* extracted as link types. These vocabulary terms can be augmented by synonyms identified using a tool such as WordNet.

B. Domain Concepts and Jargon

To discover an initial set of terms, vocabulary, and query jargon used in the traceability domain, we developed an online

TABLE I: A 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?

web-collection tool which displayed a series of TIMs and then asked trace users to think of five useful trace queries for each TIM and to formulate associated NL queries. Participants were encouraged to use any words and phrases that they wished in order to formulate their queries. We deliberately set this as an open-ended exercise because we did not want to constrain the vocabulary choice of the trace users. We included 8 trace experts in the data collection exercise and collected a total of 100 trace queries issued against two different TIMs. Table I shows eight representative queries collected during this exercise.

Prior work by Meng and Siu [17] 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. 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 were then mapped into a simple representative term. This step was similar to Minnock's *tailor* step [18]. Based on the initial sample of 100 queries we mapped 94 phrases and concepts, such as the mapping of *mitigates* to *prevents*.

Based on this simple exercise we identified three different categories of queries that we classified as *solvable*, *ambiguous*, 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 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). Finally, a *solvable* query is one which is neither intractable nor ambiguous. It is correctly solved only if TiQi returns the correct information to the user.

IV. The TiQi Model

The overall TiQi process transforms a natural language trace query into an executable SQL statement. Its primary elements are depicted in Figure 2b. Steps 3-9 are executed each time an NL trace query is issued. In the following sections we describe each of these steps and illustrate them with the 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 Figure 3.

A. Speech to Text

Our TiQi approach includes the option of speaking or writing the NL trace query. All spoken queries are transformed into textual format as a preprocessing step. We investigated several tools for supporting the speech to text step, focusing mainly on 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 had similar accuracy rates. However, Google's Speech API was more portable, faster and, ultimately, performed better on average when presented with new data, so it was chosen as our speech processor.

B. Query Simplification

The query simplification process parses and tokenizes the query. This process basically involves matching phrases found in the query to specific mapping rules, and then replacing the phrases with the mapped terms. Specific heuristics included 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", which was defined to mean "within the past week". Such definitions must be defined by stakeholders at the project level. Our initial mappings were based upon our analysis of the initial collection of trace queries.

We utilized the Stanford Parser to identify numbers, durations, times, and dates. 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.

C. Tokenizer

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 Figure 3.

Clearly there are many ambiguities that arise as a result of the tokenization. For example, it is unclear whether the term *class* means *Code Class* or *UML Class*. Disambiguation is deferred to a later step of the process. In the case that a word receives no assignment i.e. the word is not found in the vocabulary of the domain either as a basic term, a synonym, or as a mappable definition, then the query is marked as intractable.

D. Path Finder

In many cases, trace queries will be specified only in terms of the source and target artifacts, even though the actual traceability path flows through a set of intermediate artifacts. To create an executable query, we need to explicitly identify the actual path of the trace query.

To accomplish this we conduct a depth-first search to identify the complete set of trace paths between source and target artifacts. It is important to note that while all of our TIMs allowed only a single path between each pair of artifacts, there are sometimes valid reasons for redundant paths to appear in a TIM [14]. 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 depth-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 [3]. In the case that multiple candidate paths are found, the *disambiguator* must determine which is the correct one.

E. Disambiguator

The primary task of the disambiguator is to attach each value 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 either select the correct mapping or to seek clarification from the user.

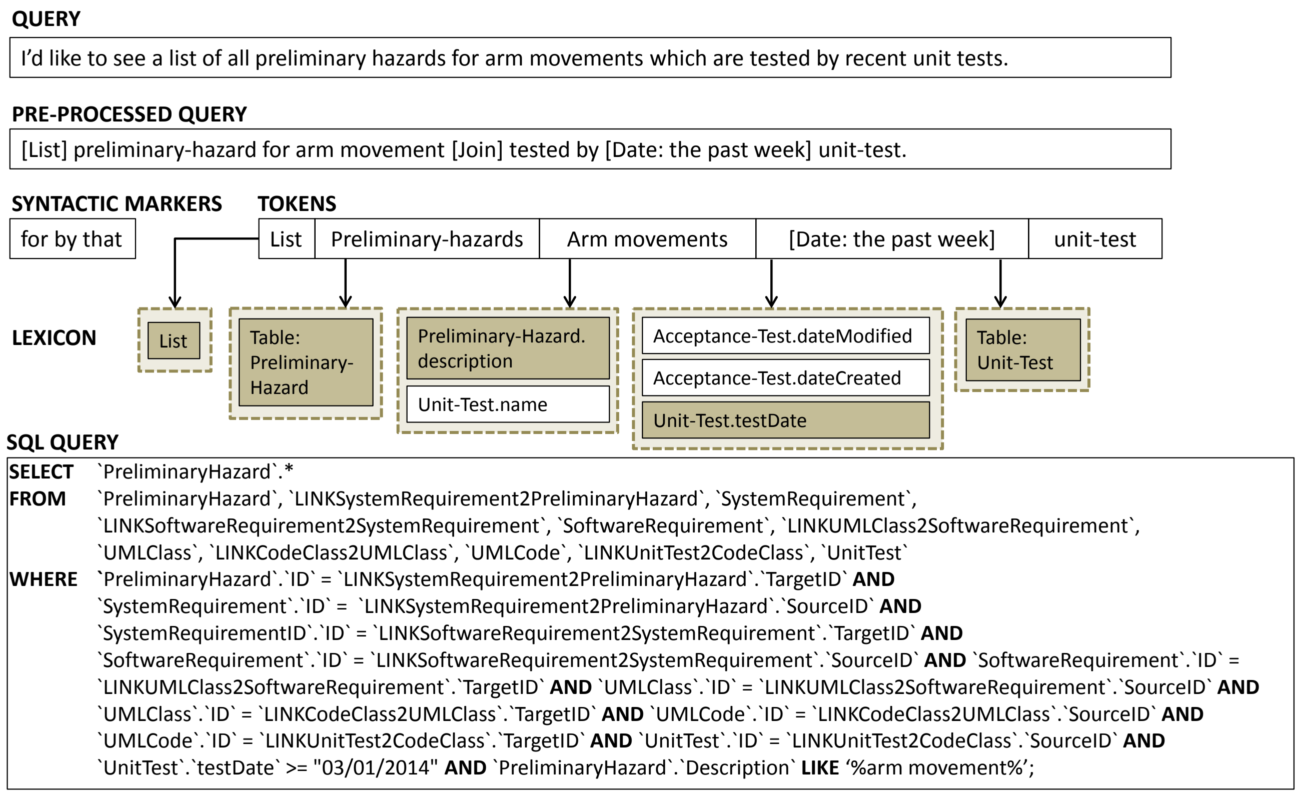
Disambiguation is quite complex, and several different techniques have been used in prior work. TiQi therefore adopts a toolbox of prioritized techniques and heuristics and uses them to resolve a variety of different kinds of ambiguities. The extent to which these ambiguities are correctly resolved directly 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 heuristic 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 list the primary rules here.

1) Rule 1: **No Competitition***:* If a token maps only to one table, one attribute, or one value element, then that element is selected.

2) Rule 2: **Max-Flow***:* Popescu et al. showed that a certain class of ambiguity could be definitively disambiguated through use of the *Max-flow* algorithm [20]. Maximum flow problems are setup as single-source and single-sink networks, and the goal is to maximize the flow through the network. For query


QUERY
I'd like to see a list of all preliminary hazards for arm movements which are tested by recent unit tests.
PRE-PROCESSED QUERY
[List] preliminary-hazard for arm movement [Join] tested by [Date: the past week] unit-test.
SYNTACTIC MARKERS
TOKENS
for by that
List
Preliminary-hazards
Arm movements
[Date: the past week]
unit-test
LEXICON
List
Table: Preliminary-Hazard
Preliminary-Hazard. description
Unit-Test.name
Acceptance-Test.dateModified
Acceptance-Test.dateCreated
Unit-Test.testDate
Table: Unit-Test
SQL QUERY

```
SELECT  `PreliminaryHazard`.*
FROM    `PreliminaryHazard`, `LINKSystemRequirement2PreliminaryHazard`, `SystemRequirement`,
        `LINKSoftwareRequirement2SystemRequirement`, `SoftwareRequirement`, `LINKUMLClass2SoftwareRequirement`,
        `UMLClass`, `LINKCodeClass2UMLClass`, `UMLCode`, `LINKUnitTest2CodeClass`, `UnitTest`
WHERE   `PreliminaryHazard`.`ID` = `LINKSystemRequirement2PreliminaryHazard`.`TargetID` AND
        `SystemRequirement`.`ID` =  `LINKSystemRequirement2PreliminaryHazard`.`SourceID` AND
        `SystemRequirementID`.`ID` = `LINKSoftwareRequirement2SystemRequirement`.`TargetID` AND
        `SoftwareRequirement`.`ID` = `LINKSoftwareRequirement2SystemRequirement`.`SourceID` AND `SoftwareRequirement`.`ID` =
        `LINKUMLClass2SoftwareRequirement`.`TargetID` AND `UMLClass`.`ID` = `LINKUMLClass2SoftwareRequirement`.`SourceID` AND
        `UMLClass`.`ID` = `LINKCodeClass2UMLClass`.`TargetID` AND `UMLCode`.`ID` = `LINKCodeClass2UMLClass`.`SourceID` AND
        `UMLCode`.`ID` = `LINKUnitTest2CodeClass`.`TargetID` AND `UnitTest`.`ID` = `LINKUnitTest2CodeClass`.`SourceID` AND
        `UnitTest`.`testDate` >= "03/01/2014" AND `PreliminaryHazard`.`Description` LIKE '%arm movement%';
```

Fig. 3: 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.

disambiguation purposes, a graph is created to contain the candidate database values spoken by the user. The single-source is the system, first level nodes are created for each candidate value's table and column, 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 Figure 4. 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.

3) Rule 3: **Pecking Order***:* If a name of 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.

4) Rule 4: **Compounding Evidence***:* When an attribute or value could potentially be associated with multiple candidate tables, for example if two tables share attributes of the same name, then a total 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 term *unitTest* to the relation *unitTest* provides compounding evidence for selecting *UnitTest.Result*.

5) Rule 5: **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 1 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.

6) Rule 6: **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, the distance from T_2 and T_3 to T_1 is measured and the closest table chosen.

Unless rules 1 and 2 result in complete disambiguation of the tokens, we cannot guarantee that the query will be correctly interpreted. However, statistical inferencing techniques can be used in domains for which a large number of queries are available [20]. 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

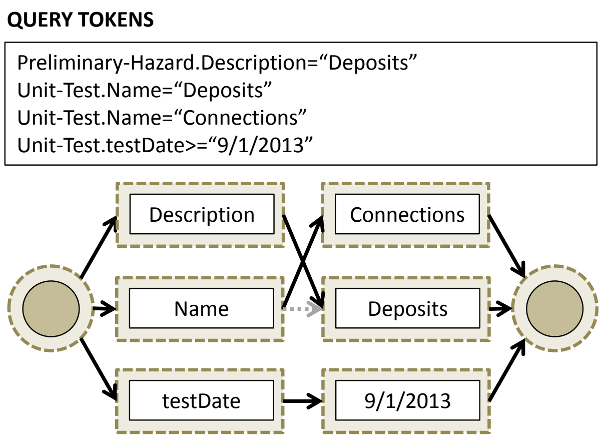

QUERY TOKENS
Preliminary-Hazard.Description="Deposits"
Unit-Test.Name="Deposits"
Unit-Test.Name="Connections"
Unit-Test.testDate>="9/1/2013"
Description
Connections
Name
Deposits
testDate
9/1/2013

Fig. 4: The Max-flow Graph. Solid black edges indicate links, whereas dashed grey edges indicate that no flow exists.

as we do not yet have sufficient trace queries; however, this approach will be integrated in future versions of TiQi.

F. SQL Generation

Once the incorrect results have been trimmed in the disambiguator, and the conflicts have been resolved, every token in the original query is mapped to one, and only one target artifact. From this, the formal SQL query can be generated as depicted in Figure 3.

V. Evaluating TiQi

We conducted two different experiments. The first was designed to explore user preference for spoken versus written natural language queries and to evaluate the correctness of the queries. 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. We release both datasets used in these experiments, including TIMs, databases, and queries at http://re.cs.depaul.edu/tiqi/dataset.html.

A. Comparison of Techniques

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 the 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 the user 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 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 very 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.

TABLE II: 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

For example, Prompt-1 (P1) stated that "The safety officer is worried that an important requirement $R136$ 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 require a trace query that incorporates 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 II. 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.

21 people participated in our study. 11 of them classified themselves as traceability experts. Of these, four used traceability in their workplace, and 7 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.

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

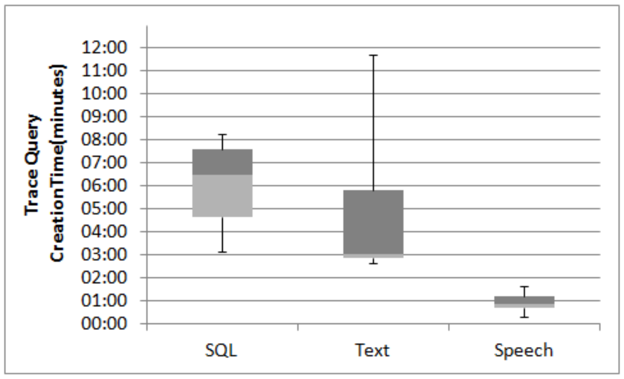


Fig. 5: Trace query creation time for each technique.

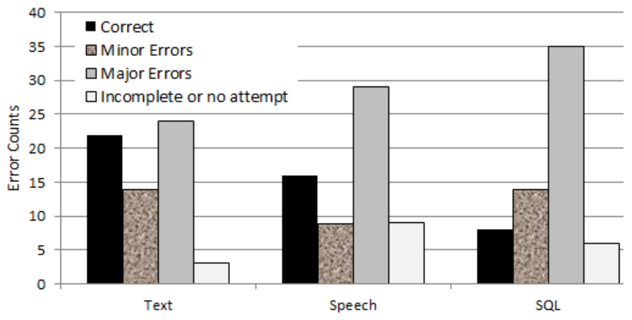


Fig. 6: Correctness of Speech, Text, and SQL queries

each of the three techniques was first computed and then the mean query creation time was calculated for each technique.

Results are displayed in Figure 5 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 minutes and 47 seconds, and SQL was the slowest at 6 minutes and 2 seconds. A t-test failed to reveal a statistically reliable difference between the two means 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-Speech and found a statistically significant difference 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 minutes or less, a few queries took much longer. In these cases the user had tried to recreate the SQL query in spoken form.

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 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 based loosely on a framework proposed by Jarke et al. [9] 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*. The categories are listed here from [9].

Correct: Statements are formulated correctly and, if executed, could return the correct information. For natural language queries this meant that the query was solvable as defined in Section III-B.

Minor Error: Query output would have been correct, but a minor error is introduced in the statement of the problem. Such errors can be fixed with minor changes.

Major Error: The query is not for the request at hand but for a different one.

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

We assessed each query manually as we did not want the effectiveness of the NL-Queries to be judged according to the current ability of TiQi. Each query was therefore encoded manually by two members of the team following a clearly defined set of prescribed rules. When disagreements occurred a third team member was consulted. For example, for the NL-approach, we encoded queries as major errors if they included entire paragraphs of text instead of succinct queries. For each of the three query techniques we calculated the percentage classified into each category. As reported in Figure 6, only eight SQL queries were categorized as completely correct compared to 16 for Speech and 22 for Text. Both SQL and Text had 14 minor errors, while Speech had only 9. Merging these two categories into a 'basically correct' one results in 36 for Text, 25 for Speech, and only 22 for SQL. As we report later in this paper, 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 Section V-A3.

3) User Perception: We designed a post-test 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 Figure 7 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 of the 21 participants assigned SQL a score of four of higher on the difficulty scale, in comparison to 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 3 preferred SQL. None of them registered a preference for Speech. In the

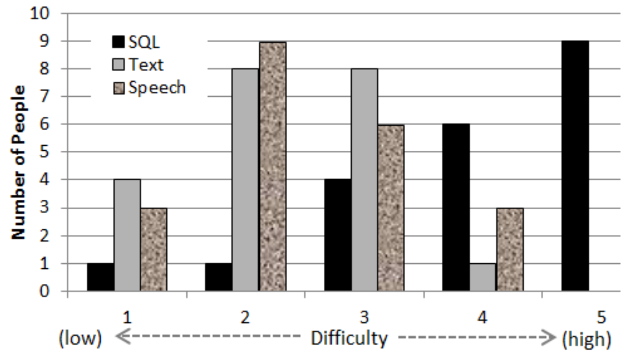


Fig. 7: User perceptions of Query Difficulty

case of the traceability experts, 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. 2 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 didn't 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.

B. *Natural Language to SQL Transformations*

The second experiment focused on the ability of TiQi to transform written NL queries to executable SQL that returned the targeted information. For these purposes we used the prototype TiQi tool we developed in C#. 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, 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 three 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.

1) Isolette Data set: The Isolette TIM 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 was primarily extracted from a documented case study about a safety-critical Isolette system [11]. 70 NL trace queries were developed. Queries were created by five different software engineers and included a variety of jargon, dates, times, negation, and included non-trivial queries which could only be addressed by traversing multiple artifacts and trace matrices.

2) Easy Clinic: The Easy-Clinic TIM includes 7 different artifacts, namely: HIPAA-goals, requirements, design, code, unit and acceptance test case and test log 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 goals were taken from the US HIPAA Technical Safeguards. All other artifacts were created by one of the researchers on this project for purposes of the project. As Easy-Clinic contains a mix of English and Italian terms, we translated Italian terms to English. 40 NL trace queries were developed.

C. *Results*

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 report results for the number of completely correct queries versus incorrect ones for each dataset in Table III. We also report results for the subset of queries for which all necessary TiQi functions have been implemented. More specifically, we exclude queries which required aggregation and identification of missing artifacts (e.g., not yet tested).

TABLE III: SQL queries generated from NL Text

All Queries	**Correct**	**Incorrect**	**% Correct**
Isolette	49	21	70.0%
Easy Clinic	25	15	62.5%

Supported Queries	**Correct**	**Incorrect**	**% Correct**
Isolette	49	7	87.5%
Easy Clinic	25	9	73.53%

Results for the Isolette dataset were very promising with 70% of the queries transformed correctly. Mistakes primarily involved the inclusion of extra tables and constraints. On the other hand, only 62.5% of the Easy-Clinic queries were correctly transformed. An initial analysis suggested a double edged problem which included more verbose queries and some ambiguous attribute names. Nevertheless both data sets represent realistic domains and will serve as baselines for future improvements.

VI. 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 many of which were created by members of our team. 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. It is clearly important to extend the sample size in the future so that we can construct a more robust transformation process.

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.

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 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 TIMs.

VII. Conclusion

The work described in this paper transforms natural language trace queries into SQL. The experiments that we conducted and the data we gathered show that the majority of users had a preference for written NL queries over either speech or SQL, and that in fact such queries were written more accurately and were perceived as being less difficult than SQL. These results support the notion that a NL interface can potentially be useful for helping project stakeholders utilize project trace data in a meaningful way. Our second experiment demonstrated the viability of transforming trace queries into executable SQL. Nevertheless there is still a large amount of work to be done in order for TiQi to consistently and accurately handle diverse and complex queries.

In future work we plan to collect trace queries from practitioners working in safety critical domains so that we can construct a more extensive trace query domain model, with a more complete set of disambiguators and transformation rules. Furthermore we plan to evaluate our approach against more complex trace queries and build supporting tools for engaging users interactively in the disambiguation process.

VIII. Acknowledgments

This work was partially funded under US National Science Foundation Grants CCF 1319680.

References

[1] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Trans. Software Eng.*, 39(5):725–741, 2013.

[2] I. Androutsopoulos and G. Ritchie. Database interfaces. In *Handbook of Natural Language Processing*, pages 209–240. Marcel Dekker Inc., 2000.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms (Second ed.).* MIT Press and McGrawHill, 2001.

[4] M. H. Göker, C. A. Thompson, S. Arajärvi, and K. Hua. Connecting people with questions to people with answers. *KI*, 21(4):23–26, 2007.

[5] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. Inter-modelling: From theory to practice. In D. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 376–391. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16145-2.

[6] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, E. A. Holbrook, S. Vadlamudi, and A. April. Requirements tracing on target (retro): improving software maintenance through traceability recovery. *ISSE*, 3(3):193–202, 2007.

[7] H. Jaakkola and B. Thalheim. Visual sql – high-quality er-based query treatment. In M. Jeusfeld and Ó. Pastor, editors, *Conceptual Modeling for Novel Application Domains*, volume 2814 of *Lecture Notes in Computer Science*, pages 129–139. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39597-3.

[8] M. Jarke, J. Krause, and Y. Vassiliou. Studies in the evaluation of a domain-independent natural language query system. In *Cooperative interfaces to information systems*, pages 101–130. Springer, 1986.

[9] M. Jarke, J. Krause, Y. Vassiliou, E. A. Stohr, J. A. Turner, and N. H. White. Evaluation and assessment of a domain-independent natural language query system. *IEEE Database Eng. Bull.*, 8(3):34–44, 1985.

[10] H.-J. Kim, H. F. Korth, and A. Silberschatz. Picasso: a graphical query language. *Software – Practice and Experience*, 18:169–203, March 1988.

[11] D. L. Lempia and S. P. Miller. Requirements engineering management handbook. *National Technical Information Service (NTIS)*, 2009.

[12] P. Mäder and J. Cleland-Huang. A visual language for modeling and executing traceability queries. *Software and System Modeling*, 12(3):537–553, 2013.

[13] P. Mäder, O. Gotel, and I. Philippow. Getting back to basics: Promoting the use of a traceability information model in practice. In *Traceability in Emerging Forms of Software Engineering, 2009. TEFSE '09. ICSE Workshop on*, pages 21 –25, may 2009.

[14] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang. Strategic traceability for safety-critical projects. *IEEE Software*, 30(3):58–66, 2013.

[15] J. I. Maletic and M. L. Collard. Tql: A query language to support traceability. In *TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 16–20, Washington, DC, USA, 2009. IEEE Computer Society.

[16] P. McFetridge and C. Groeneboer. Novel terms and cooperation in a natural language interface. In *Knowledge Based Computer Systems*, pages 331–340. Springer, 1990.

[17] H. H. Meng and K. C. Siu. Semiautomatic acquisition of semantic structures for understanding domain-specific natural language queries. *IEEE Trans. on Knowl. and Data Eng.*, 14(1):172–181, Jan. 2002.

[18] M. Minock. C-phrase: A system for building robust natural language interfaces to databases. *Data Knowl. Eng.*, 69(3):290–302, 2010.

[19] National Research Council of the National Academies, The National Academies Press, Washington DC. *Critical Code, Software Producibility for Defense*. 2010.

[20] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[21] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang. Mind the gap: Assessing the conformance of software traceability to relevant guidelines. In *Intn'l Conf. on Software Engineering (ICSE)*, 2014.

[22] S. P. Shwartz. Problems with domain-independent natural language database access systems. In *Proceedings of the 20th annual meeting on Association for Computational Linguistics*, ACL '82, pages 60–62, Stroudsburg, PA, USA, 1982. Association for Computational Linguistics.

[23] H. Störrle. VMQL: A visual language for ad-hoc model querying. *Journal of Visual Languages & Computing*, 22(1):3–29, 2011.

[24] Y. Vassiliou, M. Jarke, E. Stohr, J. Turner, and N. White. Natural language for database queries: A laboratory study. *MIS Quarterly*, 7(4):47–61, 1983.

[25] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9(4):529–565, 2010.

[26] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. An ontology-based approach for the recovery of traceability links. In *3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, Genoa, Italy, October 1st 2006.

[27] M. Zloof. Query by example. In *Proceedings of the NCC*. AFIPS.