TiQi: A Natural Language Interface for Querying Software Project Data

Jinfeng Lin, Yalin Liu, Jin Guo, Jane Cleland-Huang
University of Notre Dame
South Bend, IN, USA
Email: jlin6,JaneClelandHuang@nd.edu

William Goss, Wenchuang Liu, Sugandha Lohar,
Natawut Monaikul, Alex Rasin
School of Computing
DePaul University, Chicago, USA.
Email: arasin@cdm.depaul.edu

***Abstract*—Software projects produce large quantities of data such as feature requests, requirements, design artifacts, source code, tests, safety cases, release plans, and bug reports. If leveraged effectively, this data can be used to provide project intelligence that supports diverse software engineering activities such as release planning, impact analysis, and software analytics. However, project stakeholders often lack skills to formulate complex queries needed to retrieve, manipulate, and display the data in meaningful ways. To address these challenges we introduce TiQi, a natural language interface, which allows users to express software-related queries verbally or written in natural language. TiQi is a web-based tool. It visualizes available project data as a prompt to the user, accepts Natural Language (NL) queries, transforms those queries into SQL, and then executes the queries against a centralized or distributed database. Raw data is stored either directly in the database or retrieved dynamically at runtime from case tools and repositories such as Github and Jira. The transformed query is visualized back to the user as SQL and augmented UML, and raw data results are returned. Our tool demo can be found on YouTube at the following link:http://tinyurl.com/TIQIDemo.**

***Keywords*-Natural Language Interface, Project Data, Query**

I. Introduction

Software and Systems engineering projects accumulate large amounts of data in the form of requirements, design artifacts, source code, test cases, feature requests, bug reports, project plans, burn-down charts, and safety-related assets. These data artifacts can be connected using trace links - constructed either manually during the development process or after-the-fact with the help of information retrieval techniques [3], [4]. If leveraged effectively, project stakeholders can utilize this data to answer questions such as: "which elements of the design mitigate safety-related hazards?" or "list all test-cases written in the past week which test the functionality related to temperature controls". Traditionally, such queries have been executed using query languages such as SQL or XQuery. However, the queries can become quite complex [12], [11] and many project stakeholders find them difficult to compose.

NL database solutions have been available since the early 1970s and 80s [19], [20], [6], [2], [18]; however, it is widely accepted that NL query languages should be customized for specific domains [14]. To address this problem we have developed TiQi as a tool for querying software projects. It is supported by a traceability domain model and algorithms designed specifically to transform natural language project queries into executable SQL statements.

Supporting queries that integrate information from multiple artifact types requires interconnected data sources. Fortunately, data is becoming increasingly connected. In safety-critical projects traceability is prescribed across many artifacts by certifying bodies. Integrated design and development environments such as *Jazz* or the *Github-Jira* bridge capture associations between artifacts as a byproduct of the development process, and products such as *Mylyn*, *TaskTop* and *CollabNet* handle the challenges of integrating data across diverse toolsets. Furthermore, when trace links are not available, automated tracing techniques can be used to *generate* usable trace links upon demand [5], [1].

In this tool demo we present a Natural Language Interface called *TiQi*, which accepts both verbal and written natural language queries targeted at software project data. It then transforms these queries into executable Structured Query Language (SQL). The transformed query is visualized back to the user as simplified SQL and augmented UML, and raw data results are returned [17], [16].

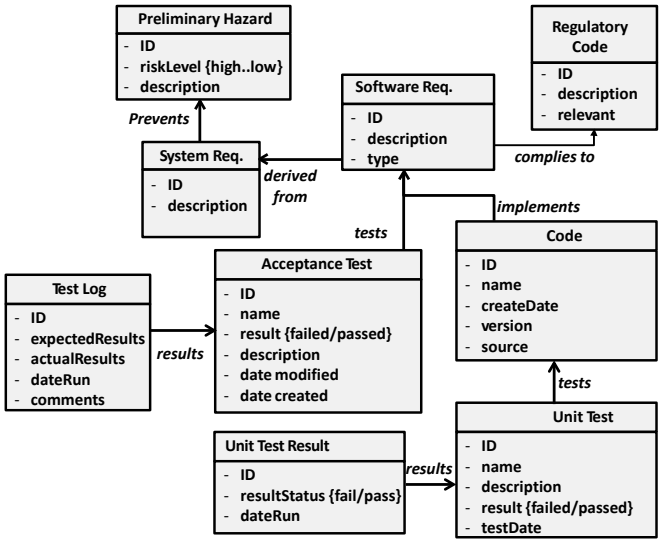


Fig. 1: A Traceability Information Model (TIM) showing available artifacts and attributes for querying against.

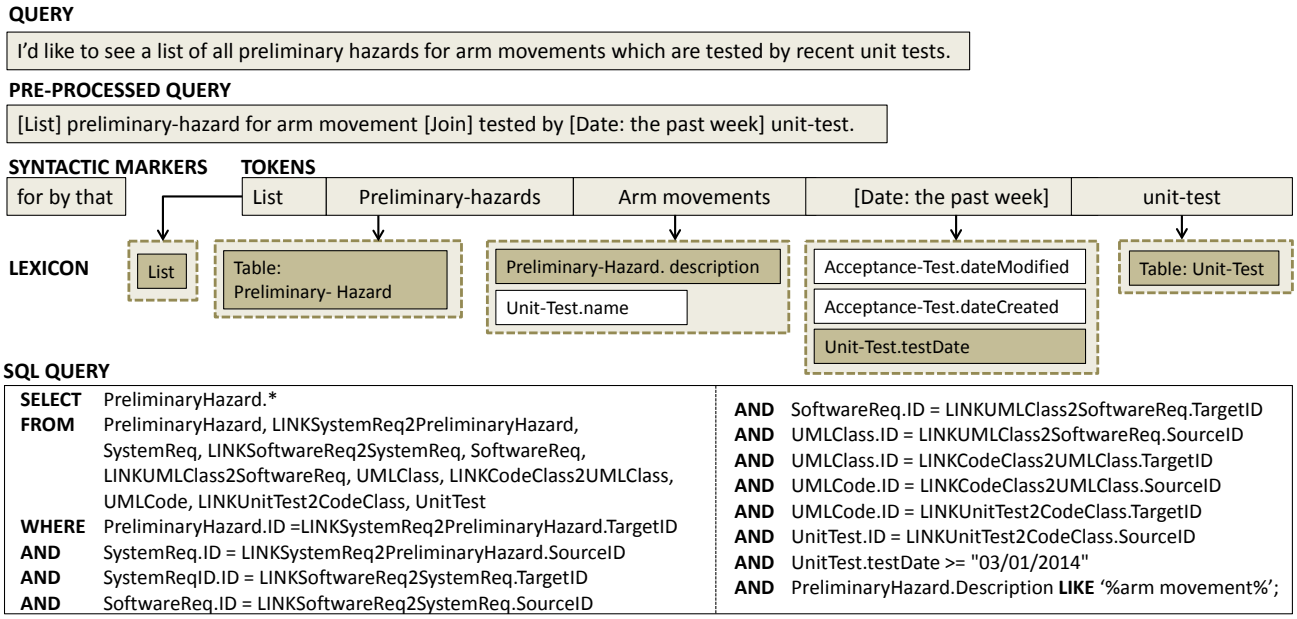


Fig. 2: An example of the Query Transformation Process in TiQi

II. TiQi Overview

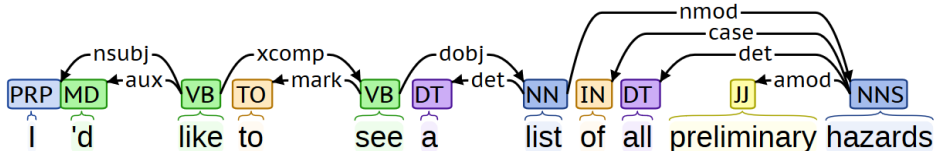
TiQi prompts the user for a NL query by displaying a Traceability Information Model (TIM), which as depicted in Figure 1, models artifact types, attributes, analytic functions, and semantically typed links. Raw data can either be stored in a centralized database or in native repositories such as Jira or Github. In the distributed scenario, we store the TIM in a central location and provide mappings to distributed nodes capable of accessing and retrieving the data specified in a specific query. In order to provide up-to-date answers over distributed heterogeneous software engineering data silos (e.g., IBM DOORS, Jira) we have developed a custom H2-based [15] database prototype. H2 is a lightweight, open-source Java SQL database with in-memory mode and full JDBC support. Our database engine therefore natively supports integration of custom analytic functions and can, through JDBC API, integrate data sources ranging from Jira to Excel spreadsheets. However, as the focus of this tool demo is on TiQi's ability to transform a natural language query, all examples in this paper are run against a central repository containing all artifact data.

III. Translation Process

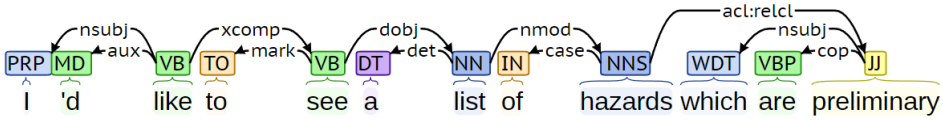
TiQi translates a wide range of NL project-related queries [10] into executable SQL. We illustrate the multi-step translation process in Figure 2. First, the Stanford Parser [7] is used to identify parts of speech and syntactical dependencies. A section of the parsed query is shown in Figure 3 [13]. A number of preprocessing tasks are then performed including identifying and replacing the prelude "I'd like to see a list of" with [SELECT]. In earlier versions of TiQi, this task was accomplished by collecting hundreds of typical query phrases and detecting them in the query; however, the approach was overly brittle and failed when unusual preludes were used. We have therefore replaced it with the NLP approach. During preprocessing, additional tasks such as replacing association terms such as "which are tested by" with the term [JOIN], and formatting dates and numbers are also performed.

In the next step, the query is transformed into a set of syntactic markers and tokens such as *preliminary hazards* and *unit-test*. The goal is then to correctly map each token onto an artifact name, attribute name, or an attribute value. TiQi does not limit users' vocabulary to the artifact and attribute names depicted in the TIM. It therefore utilizes *synonym-like* words and phrases to increase the accepted vocabulary. However, synonyms must be used judiciously in order to minimize erroneous mappings. For example, Wordnet defines several synonyms for the word *test* including *trial* and *examination*, however the term *examine* in the query "I'd like to examine a list of recently added requirements" should not be mapped to the term *tests* as this could create an incorrect mapping to the unit-test case artifact. Further, standard lists of synonyms fail to include synonym-like terms for Software Engineering such as: code:java or requirements:specs. We have therefore developed a domain-specific set of synonym-like words and phrases associated with common Software Engineering artifacts. This list, which we continue to iteratively evolve, was developed through inspecting and analyzing the terminology used in over 1,000 sample queries collected during a series of industry-targeted user studies [9].

As users can express a single NL query in diverse ways, we developed a set of grammatical patterns to guide detection of syntactic markers. This approach establishes some independence between the extraction process and the exact wording of the query. For example, given the two queries shown in Figures 3a and 3b respectively, we apply a rule to identify restrictions (e.g., adjectives and verbs) that describe a noun. In both queries we trivially locate the initial noun *hazard* and then apply one of our grammatical patterns to search through



(a) Dependency for the example query



(b) Dependency for a mutation of the example query

Fig. 3: A visualization of Dependencies identified by the Stanford Parser for a shortened form of the our example query.

'amod' and *'acl:relcl'* dependencies in the dependency graph to identify the complete syntactic marker *preliminary hazard*.

While TiQi is often able to map a token onto a specific table and/or attribute; there are other cases in which multiple mappings are possible. Figure 2 provides an example with the case of *Arm Movements*. This phrase does not match a table or attribute name, but does occur in at least one Unit-Test case name as well as a preliminary hazard description. TiQi currently utilizes a set of five disambiguators which are used to select tables, attributes, and data values. In this example, the rule of *compounding evidence* suggests that as the token *Preliminary-hazards* is already clearly mapped onto a table with the same name, the *arm movements* is more likely to map to the *preliminary hazard* table than to the *unit-test* table. Other disambiguators prioritize table names and attributes over raw data matches, and matches against constrained value lists (e.g. pass—fail) over larger textual descriptions such as source code files. The disambiguators therefore enable TiQi to identify the most likely interpretation of the query, but they do not guarantee a correct match.

IV. Displaying Results

Once TiQi has transformed a query into executable form, it executes the query and reports results back to the user as depicted in Figure 4. A series of user studies which explored various display representations [9] demonstrated the benefits of providing both visual and textual display of the query. TiQi therefore displays the subset of the TIM which is relevant to the query, and displays the artifacts, attributes, and filter conditions. TiQi also displays a simplified version of the generated SQL. The primary simplification is achieved through hiding the joins across the hidden trace matrices. Finally, retrieved data is depicted in tabular format. This multi-faceted display enables the user to determine whether the query was correctly interpreted and also to view its results [9].

V. TiQi Architecture

The current TiQi architecture has three major components as depicted in Figure 5:

Backend TiQiEngine: The backend Trace Engine is written in java and utilizes external libraries such as the Stanford parser. It takes a natural language query and attempts to output a well-formed SQL statement. If it fails to do so, it reports its inability to understand the query.

Web-Server: The webserver is written in Java using the Spring framework. It responds to client requests for artifact data from the central or distributed database. It also serves as an intermediary between the web-client and the TiQi engine. In one typical scenario, the web-server receives a NL query from the web-client, asks the TiQi engine to transform it, executes the SQL query against project artifacts, and finally displays results to the user. The web-server utilizes the GraphViz dots format to dynamically layout the TIM and to visualize the query. The generated layout is transformed into JSON and passed to the front-end client.

Web-Client: The web-client supports interaction with the user. After the user selects a project, the web-client forwards the request to the web-server, waits for the JSON representation of the TIM, and then renders the visual form of the TIM. When the user formulates an NL query and presses the submit button, the web-client forwards the request to the trace engine, waits for the results, and finally displays them in the browser.

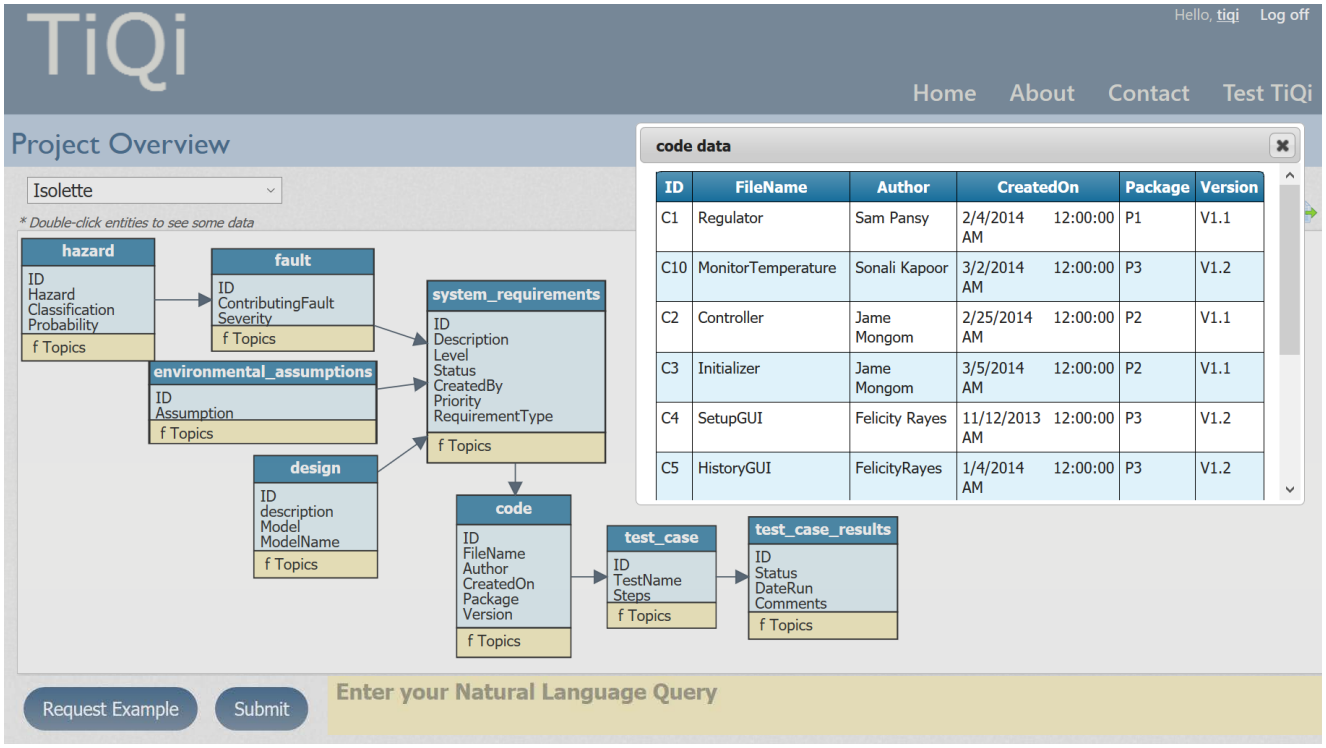
These scenarios are illustrated in Figure 4a. The user has opened one of the sample projects named *Isolette*. The project's TIM has been retrieved and displayed and the user has clicked on the *code* artifact to view the raw data. The user then formulates a query either by activating the microphone or by typing it into the query field at the bottom of the screen. The query is processed as previously explained in Section III and the result is displayed as shown in Figure 4b.

VI. TiQi's Performance

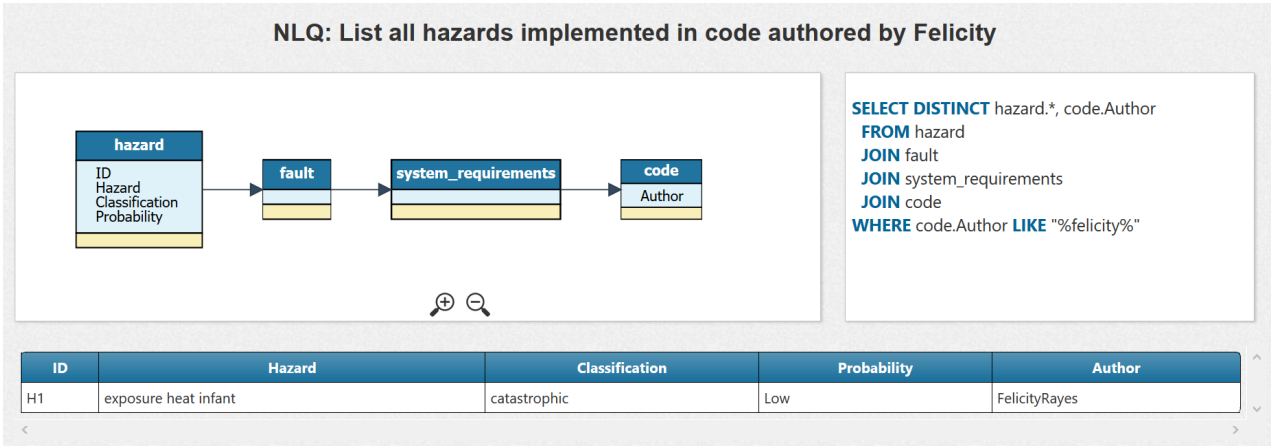
We report results from an earlier study conducted to evaluate TiQi's performance against two data sets [17]. While these data sets are relatively small, they demonstrate TiQi's capabilities. The *Isolette* data set included hazards, faults, environmental assumptions, system requirements, design, code,

TABLE I: SQL queries generated from NL Text

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



(a) Project artifacts serve as a query prompt



(b) Results are presented in three formats: relevant TIM, simplified SQL, and retrieved data

Fig. 4: Issuing a Query and Displaying results in TiQi's web-client

test cases, and test results connected through six unique trace paths and described using 26 unique attributes. The data was taken from a case study documenting the safety-critical Isolette system [8]. We collected queries from five different software engineers (unrelated to our project) and then augmented them by incorporating a variety of jargon, dates, times, and negations. This produced a total of 70 natural language queries. The *Easy-Clinic* data set included HIPAA-goals, requirements, design, code, unit and acceptance test case and a test log connected through six unique trace paths and described by 17 attributes. The functional requirements, source code and test cases were taken from the Easy-Clinic data (available from CoEST.org) with italian terms translated to English. HIPAA Technical Safeguards were added as goals, and all other artifacts were created by one of the researchers in order to have a richer dataset for querying against.

To evaluate TiQi we issued each of the queries and then reviewed both the generated SQL and the data output when the SQL was executed. Results were marked as correct or incorrect, and the numbers of completely correct queries versus incorrect ones are reported for each dataset as shown in Table I. Results from this study showed TiQi's promise. However, in a subsequent study, when the size of the second dataset was significantly increased by adding additional records and

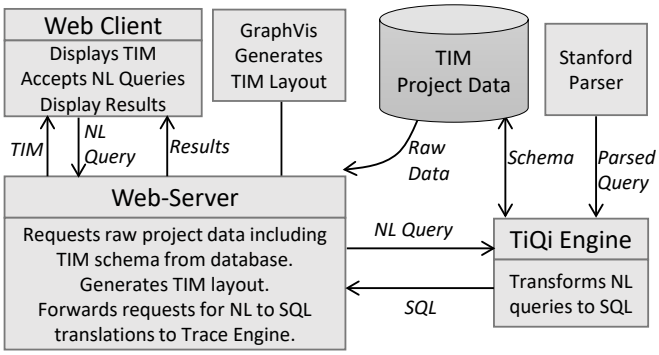


Fig. 5: TiQi Architecture with Centralized Repository

TABLE II: Two examples in which TiQi currently fails: (1) Faults containing the keyword 'temperature' were retrieved, while the ones containing words such as 'heat' and 'thermostat' are excluded from the answer set, (2) As no 'low' severity faults existed in the data at the time of the query, TiQi failed to add the condition 'fault'.'Severity' = 'low'.

Queries	**SQL**
List any fault related to temperature	SELECT 'fault'.'ID' FROM 'fault' WHERE 'fault'.'ContributingFault' Like "%temperature%"
List all low severity faults which have pending requirements	SELECT 'system_requirements'.'ID','fault'.'Severity' FROM 'system_requirements' JOIN 'fault' JOIN 'hazard' WHERE 'system_requirements'.'Status' LIKE "pending"

large quantities of unstructured text, accuracy dropped to 48%. We are therefore working to incrementally improve TiQi in several ways. For example, we have experimented with the use of machine learning at several key decision points such as differentiating between different types of questions. Results have shown high accuracy yes/no (0.914), negation (0.954), and aggregation (0.974) and therefore this classification feature will be integrated into the next TiQi release. A second major focus is to create a interactive dialog as a mean of solving ambiguities in queries. Finally, our longer term goal is to publicly deploy TiQi and to leverage it to interact with users in order to better understand the categories of questions users wish to pose. The current version of TiQi is based upon approximately 1000 queries collected through a series of surveys [10], [9]

VII. Ongoing Challenges

TiQi is designed to make project data more accessible; however, it is limited by the underlying domain ontology and its ability to interpret the intent of the NL query. Table II shows 2 typical errors during the SQL translation. Both of the errors indicate the fact that the accuracy of translation is associated with the content and the design of the target databases. While we are currently working to further improve TiQi's interpretive ability in order to improve accuracy, we are also exploring more interactive solutions based on question and answering. TiQi is expected to calibarte itself to fit the target database during the interactions with users.

VIII. Acknowledgments

The work described in this tool demo is partially funded by NSF grants CCF-0959924 and CCF-1265178.

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] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 55–69, 2014.

[4] J. Cleland-Huang, O. Gotel, and A. Zisman, editors. *Software and Systems Traceability*. Springer, 2012.

[5] 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.

[6] Y. Kambayashi. An overview of a natural language-assisted database user interface: Enli. In *IFIP Congress*, pages 1055–1060, 1986.

[7] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[8] D. L. Lempia and S. P. Miller. Requirements engineering management handbook. *National Technical Information Service (NTIS)*, 1, 2009.

[9] S. Lohar, J. Cleland-Huang, and A. Rasin. Evaluating the interpretation of natural language trace queries. In M. Daneva and O. Pastor, editors, *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*, volume 9619 of *Lecture Notes in Computer Science*, pages 85–101. Springer, 2016.

[10] S. Lohar, J. Cleland-Huang, A. Rasin, and P. Mäder. Live study: Collecting natural language trace queries. In *Research Method Track, Requirements Engineering: Foundation for Software Quality (REFSQ 2015), Essen, Germany, March 23, 2015.*, pages 207–210, 2015.

[11] P. Mäder and J. Cleland-Huang. A visual traceability modeling language. In *MoDELS (1)*, pages 226–240, 2010.

[12] J. I. Maletic and M. L. Collard. TQL: A query language to support traceability. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE@ICSE 2009. Vancouver, BC, Canada, 18 May, 2009*, pages 16–20, 2009.

[13] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

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

[15] T. Mueller. H2 database engine, 2006.

[16] P. Pruski, S. Lohar, R. Aquanette, G. Ott, S. Amornborvornwong, A. Rasin, and J. Cleland-Huang. TiQi: Towards natural language trace queries. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 123–132, Aug 2014.

[17] P. Pruski, S. Lohar, W. Goss, A. Rasin, and J. Cleland-Huang. Tiqi: answering unstructured natural language trace queries. *Requir. Eng.*, 20(3):215–232, 2015.

[18] B. Shneiderman and P. Maes. Direct manipulation vs. interface agents. *Interactions*, 4(6):42–61, 1997.

[19] C. Thompson, R. Mooney, and L. Tang. Learning to parse natural language database queries into logical form. *Proceedings of the ICML-97 Workshop on Automata Induction, Grammatical Inference, and Language Acquisision*, 1997.

[20] 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.