Automated Fine-Grained Requirements-to-Code Traceability Link Recovery

Juan Manuel Florez  
The University of Texas at Dallas  
jflorez@utdallas.edu

Abstract—Problem: Existing approaches for requirements-to-code traceability link recovery rely on text retrieval to trace requirements to coarse-grained code documents (e.g., methods, files, classes, etc.), while suffering from low accuracy problems. Hypotheses: The salient information in most requirements is expressed as functional constraints, which can be automatically identified and categorized. Moreover, people use recognizable discourse patterns when describing them and developers use well-defined patterns for implementing them. Contributions: Recasting the requirements-to-code traceability link problem as an accurate matching between functional constraints and their implementation.

Keywords—traceability, static analysis, discourse analysis, qualitative analysis

I. PROBLEM AND RESEARCH STATEMENT

Requirements traceability [1] is mandatory in safety-critical domains [2] and it is essential in supporting acceptance testing or regulatory reviews. The creation and maintenance of traceability links between requirements and code is recognized as a grand challenge in software traceability [3]. One solution to address the costs and problems related to producing and maintaining traceability links is automated traceability link recovery (TLR). TLR is most challenging when tracing between artifacts of different types. Particularly difficult is requirements-to-code traceability link recovery (RCTLR). Antoniol et al. [4] proposed formulating RCTLR as a text retrieval problem, leveraging the fact that requirements are written mostly in natural language and source code also contains natural language within comments and identifiers. Nearly two decades of research was devoted to developing and improving such automated RCTLR techniques [5], [6]. Unfortunately, these approaches suffer from problems that limit their applicability.

Ill-defined trace granularity leads to low RCTLR accuracy [7]. Given that the RCTLR problem is defined as document retrieval, there is a need to treat the source code as a document corpus. Most requirements-to-code traceability links are of the many-to-many type, for example, one requirement is implemented using several classes, and conversely, one class may be involved in the implementation of multiple requirements. In consequence, trace links are too coarse-grained and do not reference the artifacts of interest [7]. Rempel et al. [2] observed that method-level trace links are less accurate than class-level links and their cost is much higher. Not only this, but a mismatch between the terms used in source and target artifacts further challenges the accuracy of these approaches [8].

Trace link usage in real-world applications has been identified as a remaining major challenge in the field [3]. Recent work also highlighted that engineers do not trust traceability links due to their low accuracy, which greatly limits their usefulness [8].

II. RESEARCH GOAL AND HYPOTHESIS

We propose transforming RCTLR from a text retrieval problem (as it is most often formulated) into a heuristic-based approach, using static code analysis and text analysis. The goal is to achieve automated, highly accurate, and fine-grained RCTLR. We will use such fine-grained traceability links to improve software engineering tasks, such as bug localization.

Our main hypothesis is that most requirements contain functional constraints (FCs). We define FCs as constraints that apply to specific entities, their attributes, events, action, or users from the software domain. By contrast, non-functional requirements constrain elements such as runtime performance, system responsiveness, etc. We also believe that FCs can be categorized into a relatively small number of FC types and that developers use a limited number of identifiable patterns to implement each FC type, which we call functional constraint implementation patterns (FCIPs). For example, consider this excerpt from the requirements of the iTrust¹ system (a healthcare software):

“After three failed attempts with a userid in a given session, disallow attempts to log in via IP address...”

The FC in this case limits failed login attempts to three. The implementation of this constraint is:

```java
return authDAO.getLoginFailures(ipAddr) < 3;
```

We argue that using static code analysis techniques to find one or a few statements that implement the constraint (i.e. a fine-grained trace) would have a higher chance of success than using a text-retrieval-based approach to look for the method or class in which the implementation is contained. However, text-retrieval-based approaches cannot support this kind of granularity. Therefore, we plan to leverage the regularity of FCs and FCIPs to improve the precision of RCTLR by formulating a heuristic-driven approach, which we will describe in the next sections.

III. PROPOSED RESEARCH AND PRELIMINARY RESULTS

We define four specific research goals for fulfilling the vision of this dissertation, which we will describe in detail.

A. Identifying and Classifying FCs in Requirements

We conducted a preliminary investigation on the requirements of the iTrust system, which is an application in the

---

¹ http://agile.csc.ncsu.edu/iTrust/wiki/doku.php
healthcare domain developed as a student project at NCSU, and has been used extensively in traceability research [9]. The iTrust dataset is composed of 131 requirements that represent user scenarios described in the 40 use cases of the system.

We employed a qualitative content analysis methodology based on open coding [10] to extract the FC instances found in the 131 requirements and sort them into abstract categories (i.e., FC types). The open coding methodology was applied in a data-driven way, i.e., the FC types were inferred from the FC instances rather than predefined. Each type was identified with a unique name (a.k.a. code [10]) and was defined using a textual description of the category and specific categorization rules.

The process started with a discussion of the concept of FC, which resulted in the coding frame [10] to be employed. After the research team agreed that the concept was clear enough, we proceeded to examine all the use case scenarios identifying the FCs embedded therein. Each time a new FC was identified, the coder checked whether this new instance fit into any of the defined FC types. If this was the case, the instance was categorized under the corresponding type, otherwise, a new type was defined to accommodate the instance. As the process progressed, the proposed types were discussed with the team to ensure logical consistency, and the FC types were renamed and merged accordingly. The process resulted in the identification of 409 FC instances, which were categorized into 31 FC types. These 31 types were further grouped into 11 high-level types.

To illustrate what constitutes a FC type, we refer to the example FC in section I. The FC presented there belongs to the type attribute-less-than, which is a sub-type of attribute-value. The attribute-value type constrains an attribute in the software with respect to a value (in this case the constant 3). The constraint can be of type greater than, less than, equal to, etc.

An important observation is that approximately 95% of the identified FCs are described by only four high-level FC types. This distribution leads us to expect that the number of FC types will not increase drastically as we investigate more data. These preliminary results serve as evidence towards confirming our hypothesis: functional constraints in requirements can be categorized into a small set of conceptual types.

Our proposed work includes extending our taxonomy of FC types by leveraging more requirements data used in previous traceability studies [9]. These data will not only include textual requirements and use cases, but also change requests and bug reports [11], since we theorize that FCs can be found in all kinds of requirements documents. Once we have refined our taxonomy by including as many data sources as possible, we will examine the discourse patterns that are used to express each FC type and build a catalog. There are examples of this kind of research in the fields of requirements [12] and bug triage [13]. We will evaluate the quality of our taxonomy and catalog by employing multiple coders and calculating agreement metrics, as done in other research that uses open coding [13], [14]. This will result in the development of a catalog of discourse patterns, which we will later use to automatically detect FCs (section C.1).

Related work has focused on studying the regularity of language in requirements documents [12], [15], [16] and in identifying requirement patterns for reuse [17], however, none of these studies aim to develop a taxonomy to categorize constraints in functional requirements.

B. Discovering and Cataloging FCIPs in Source Code

Our preliminary results also stem from our study of the iTrust system, which consists of approximately 38K lines of code, distributed among 227 Java files and 141 JSP files. We selected 110 FC instances (of the 409 identified) and manually traced them to the source code. We leveraged the existing coarse-grained traces of the iTrust dataset, available from the original developers. The existing iTrust traces are from use case scenarios to code classes or JSPs.

Each time a new fine-grained trace was identified, it was classified into an FCIP. We focused on identifying minimal-size implementations, i.e., the smallest number of lines of code that implement the FCIP. The average implementation size (in terms of lines of code) for the 110 traced FC instances is 2.2 (the median implementation size is 2), and the maximum implementation size is 14, occurring for a single FC. We identified 27 FCIPs and we consider 10 of these to be frequent, as they were used to implement more than five FCs in our data.

The example in section I corresponds to the type labeled comparison-to-constant. Implementations corresponding to this FCIP use an operator such as $<$, $>$, or a method such as Java’s equals() to compare an attribute to a constant value. Note that this classification allows for substantial variability. For example, the result of the getLoginFailures() method could be assigned to a variable, or the literal 3 could be stored in a constant field, and the implementation would still correspond to this pattern.

The relationships between FC types and FCIPs are many-to-many. In other words, a given FC type may be implemented in more than one way. On average, each FC type corresponds to 2.5 FCIPs. An extreme example is the attribute-empty FC type (a subtype of attribute-value), which is implemented by eight different FCIPs. Conversely, a given FCIP may correspond to several FCs. In other words, developers sometimes implement different FCs in the same way. On average, each FCIP corresponds to 2.1 FCs.

The distribution of FC implementations among FCIPs is evidence in favor of our hypothesis: FC types are implemented by a narrow set of well-defined patterns.

Our proposed work for this goal is very similar to that detailed in the previous section, except in this case we will be looking for evidence of regularity in source code instead of natural language requirements. We will be able to leverage the same datasets and their existing traces. This will result in the creation of a catalog of FCIPs much like our catalog of FCs, which we will use as detailed in section C.2).

To the best of our knowledge, our work is the first aiming to develop a qualitative understanding of the space of requirements implementations.

C. Developing Automated Fine-Grained Traceability

Our work on the first two research goals will allow us to develop a qualitative understanding of FCs in both natural language documents and their implementations in the code. We will leverage this knowledge to develop novel approaches that allow us to automatically trace FCs from requirements to code.

1) Automatically Identifying Functional Constraints

Our proposed automated approach will first scan a requirement and identify FCs using the discourse patterns catalog (described in section A). It will then identify the constraint elements...
found in the pattern (e.g. attributes, events, values). Third, we will match these elements to identifiers used to name variables, fields, parameters, methods, types, or constants, and/or to literals in the code. Finally, the candidate identifiers will be ranked according to a similarity score. Note that we use text matching algorithms since these are substantially more accurate and easier to use than text retrieval techniques, especially on a subset of identifiers. We propose to investigate different strategies for matching the FC elements to code identifiers and utilize various similarity scores for ranking the identifiers. In each case, we will use standard preprocessing techniques (e.g. stemming, identifier splitting) [18].

2) Automatically Identifying FCIPs

Once FCs are identified in requirements, we propose applying static code analysis to identify the atomic implementation of FCs in the code. Numerous static analysis techniques have been developed and applied in practice for various software engineering tasks. However, such techniques have not been used to address RCTL. The main reason, as mentioned before, is that past research has mainly focused on text retrieval.

We will utilize data- and control-flow analysis, as well as other static analysis techniques (e.g. program slicing) to trace relevant code statements to the FCs. However, there are several challenges to address when developing principled, effective static analyses for tracing implementation patterns. First, new implementation patterns are identified incrementally. Since tracing different implementation patterns may require the combinations of various static analysis algorithms, it can be time-consuming to develop new analyses for these patterns. Second, different solutions may exist to trace an implementation pattern and it is difficult to decide the best solution, especially when the performance often depends on the target program.

To address these challenges, we propose to construct a modular and reusable static analysis infrastructure. The infrastructure will integrate various static analysis algorithms that are designed to be easily combined for our task, so an algorithm can be reused and combined with other algorithms to trace different FCIPs. Specifically, we will build on top of an existing static analysis infrastructure (e.g., WALA\(^2\) and SOOT\(^3\)), integrating program slicing, and various other control/data flow analyses. We will make algorithms available that have different levels of precision as analysis options. For example, we will enable various context sensitivity options that distinguish how precise the slicing algorithms are when handling method calls, including object sensitivity [19], and call-site sensitivity [20].

3) Automatically Tracing Functional Constraints to Code

We will combine the approaches of automatic detection of FCs and their implementations to achieve a complete RCTL solution. The proposed approach will work by automatically identifying FCs in requirements and then using the catalog of FCIPs to determine candidate patterns. Instances of these candidate patterns will be located in the code using static analysis, and finally, they will be ranked using heuristics and textual similarity scores. We plan to perform empirical evaluations with ground-truth benchmarks (i.e., programs with identified FC elements and corresponding links). Such data will be produced by the work described in sections A and B. We will assess the accuracy of our automated solution by checking if the ground truths are correctly identified by the static analysis and the natural language heuristics. We will evaluate the precision by computing the false positive rate. A false positive is an FCIP reported by the static analysis that is not considered as a traceable pattern in the ground-truth benchmarks. Recall will also be assessed for those FCIPs that involve multiple code elements.

Related work in automated TLR has formulated the problem as text retrieval [4]–[6]. More related to our research, existing work evaluates the use of structural analysis of code artifacts to improve automated trace recovery [21]–[23]. This research differs from ours because the code structure used is still restricted to coarse-grained relationships between methods or classes based on use relations. In contrast, we propose reasoning at statement level in the code and FC level in requirements.

D. Improving Bug Localization

We aim to improve applications of RCTL to potentially incentivize its use in real-world scenarios. Our proposed work in this area aims at investigating different strategies for using the recovered fine-grained traceability links to improve existing text-retrieval-based bug localization (TRBL) approaches. We focus on improving TRBL approaches that rely on bug reports (and optionally on other data sources). We will consider the bug descriptions as pseudo-requirements and identify the embedded FCs and trace them to the code. TRBL approaches typically work at class and file level, and sometimes at method level. Our RCTL approach, resulting from the work in Section C.3) will be able to operate at a range of granularities, it will thus be compatible with all types of approaches.

First, we will explore ways to use the traceability links to improve the ranking produced by the TRBL techniques. The recovered links may crosscut different methods, classes, or files. We may use our fine-grained traces to boost the ranking of program elements at any granularity, e.g. we will boost the methods that contain statements used to implement an FC identified in the bug report.

We will also use the recovered traceability links to point to the buggy code directly. Bug reports may contain both functional and technical constraints. Technical constraints describe code level restrictions or conditions, which are often described by developers rather than end users. We expect that such constraints will be similarly expressed/implemented as FCs, hence we plan to use our RCTL method to trace such constraints to the code. We expect that our FC taxonomy and implementation patterns catalog will cover most (if not all) of the constraints in the reports as well as their implementation. If not, we will expand the catalog as needed. Recent related work has explored leveraging requirements-to-code links to improve TRBL [24]. Our previous work has also shown promise in leveraging discourse analysis to improve TRBL [14].

IV. TIMELINE FOR COMPLETION

We expect to complete the proposed work within two years. During the first year, we will concentrate on acquiring the necessary data and performing the analysis required for the qualitative part of our research (sections IIIA and IIIB). We plan to use as many of the datasets used in previous traceability studies as possible [9] (contingent on the availability of both

\(^2\) http://wala.sourceforge.net/wiki/index.php/Main_page

\(^3\) https://github.com/sable/soot
Once the data have been acquired, we will manually build the catalogs of FC types, FC discourse patterns and FCIPs. It should be noted that work of this kind entails significant effort, as noted in previous studies using similar methods [13], [14]. Therefore, we expect at least a year will be needed to develop this knowledge base to an actionable state.

The second year will be spent developing the automated FC detector and the FCIP analysis framework. These will be evaluated with empirical studies using the knowledge base developed during the first year as ground truth. Towards the end of the second year, these two approaches will be combined to develop a full RCTLR solution. We will also explore its application in TRBL, using existing datasets as a starting point [11], [14].

V. ANTICIPATED CONTRIBUTIONS

We propose transforming requirements-to-code traceability link recovery from a text retrieval problem, as it is most often formulated, into a heuristic-based approach, (using static code analysis and text analysis) which accurately matches the functional constraints embedded in requirements to their source code implementations. Our research will also increase the body of knowledge of software engineering by discovering and building a catalog of FCs and FCIPs. This catalog will be publicly accessible online, and the research community will be encouraged to submit contributions. We will consolidate both existing and newly acquired datasets in the area of traceability link recovery and produce new ground truth data which explicitly identifies FCs and FCIPs. The newly created datasets will be available to the community. More importantly, we expect this paradigm shift to enable more precise automated traceability link recovery, which in turn will allow the development of new approaches that benefit from the recovered links in software engineering tasks (e.g. bug localization) and enable new avenues of research.

VI. ACKNOWLEDGMENTS

Juan Manuel Florez is supported in part by the following NSF grants: CCF-1526118 and CCF-1848608.

REFERENCES


