Retrieving Data Constraint Implementations Using Fine-Grained Code Patterns

Juan Manuel Florez
Jonathan Perry
Shiyi Wei
Andrian Marcus
{jflorez,jperry,swei,amarcus}@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Business rules are an important part of the requirements of software systems that are meant to support an organization. These rules describe the operations, definitions, and constraints that apply to the organization. Within the software system, business rules are often translated into constraints on the values that are required or allowed for data, called *data constraints*. Business rules are subject to frequent changes, which in turn require changes to the corresponding data constraints in the software. The ability to efficiently and precisely identify where data constraints are implemented in the source code is essential for performing such necessary changes.

In this paper, we introduce Lasso, the first technique that automatically retrieves the method and line of code where a given data constraint is enforced. Lasso is based on traceability link recovery approaches and leverages results from recent research that identified line-of-code level implementation patterns for data constraints. We implement three versions of Lasso that can retrieve data constraint implementations when they are implemented with any one of 13 frequently occurring patterns. We evaluate the three versions on a set of 299 data constraints from 15 real-world Java systems, and find that they improve method-level link recovery by 30%, 70%, and 163%, in terms of true positives within the first 10 results, compared to their text-retrieval-based baseline. More importantly, the Lasso variants correctly identify the line of code implementing the constraint inside the methods for 68% of the 299 constraints.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; *Design patterns*; Requirements analysis; **Software maintenance tools**.

KEYWORDS

business rule, data constraint, traceability link recovery, empirical study, fine-grained traceability, code pattern

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510167

ACM Reference Format:
Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. 2022. Retrieving Data Constraint Implementations Using Fine-Grained Code Patterns. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510167

1 INTRODUCTION

Business rules describe the operations, definitions, and constraints that apply to an organization [68]. When a software system is developed to support such an organization, these rules inform the creation of the software requirements. However, business rules are known to change unpredictably. For example, the Reserve Requirements for Depository Institutions (§204.2(d)(2), Regulation D) of The Federal Reserve [8] limits withdrawals or outgoing transfers from a savings or money market account to no more than six such transactions per statement period. This restriction was temporarily lifted in 2020 [7], making the number of transfers no longer limited. Performing this change on a system subject to this regulation requires knowing the source code elements that are responsible for implementing this rule.

Many business rules (and other type of requirements) are translated within the software system into *data constraints* [27, 67, 68]. Data constraints specify what values are allowed or required for the given data. In the example above, the relevant data are the number of monthly withdrawals and the number of monthly transfers from an account (savings or money market). The constraint states that the sum of the values of these two data elements should be less than or equal to six. The two data elements are defined in the code (we call these *data definition statements*) and the constraint is checked in some other part of the code (which we call *constraint enforcing statement*). Developers implementing the changes caused by the new rule will have to find the data constraint enforcing statements, and they could benefit from tool support, as is the case in any software change process [12, 13].

Recent research by Yang *et al.* [69] explored the implementation of data constraints in database-backed web applications. They discovered that developers struggle with maintaining consistent data constraints and with checking them across different components and versions of their web applications. This observation underscores the need for tool support when it comes to maintaining the implementations of data constraints in particular.

In this paper, we propose and evaluate a new approach, Lasso (**L**ocating d**A**ta con**S**traints in **S**ource c**O**de), that can automatically

identify the enforcing statements for a given data constraint in a software. Lasso is designed as an extensible framework (Section 3). The framework is built on top of a standard *traceability link recovery tool* (*i.e.,* for a textual input, it returns a list of relevant methods from the code), which is replaceable and customizable. The main novelty is that Lasso formalizes and uses structural information about data constraint implementations originating from previous work on the subject [27] (Section 2). With this information, Lasso is able to improve the method-level retrieval performance of the underlying traceability link recovery tool. More importantly, Lasso is able to pinpoint the lines of code where a constraint is implemented. The ability to trace data constraints to line-of-code level implementations sets Lasso apart among traceability link recovery approaches, which largely operate at coarser granularity levels (*i.e.,* file, class, method) [9].

Florez et al. [27] conducted an empirical study and identified 30 data constraint implementation patterns used by developers in Java code. The implementation patterns describe the structure of the implementations, and are presented in a pattern catalog. We convert the pattern descriptions into context-free grammar production rules. These productions rules are then used by Lasso to identify lines of code that exhibit the implementation patterns.

We instantiate Lasso with detectors for 13 of the most commonly occurring patterns from the pattern catalog, and refer to this version of Lasso as Lasso-13. With these components, we create three instances of Lasso, *i.e.,* Lasso-13Luc which uses a Lucene-based traceability tool [48] (BM25); Lasso-13VSM which uses the Vector Space Model [32]; and Lasso-13LSI which uses an LSI-based traceability tool [22]. We evaluate these three instances of Lasso-13 and compare the performance of each with their respective traceability tool as the baseline. Specifically, we compare their performance on retrieving methods implementing 299 constraints in 15 real-world Java systems. 163 of these constraints are from previous research [27], while 136 are new to this paper. We found that the Lasso-13Luc, Lasso-13VSM, and Lasso-13LSI outperform their baselines by 30%, 70%, and 163% (in terms of true positives retrieved in the first 10 results), respectively. In addition, we evaluate Lasso's accuracy in pinpointing the lines of code enforcing the constraints within the relevant methods. We found that Lasso-13 ranks the correct enforcing statements accurately for 79% of the 299 constraints (68% at rank 1 and 11% at rank 2-3).

The main contributions of the paper are:

- A novel approach and framework that, for a given data constraint, automatically finds the method and lines of code where it is implemented.
- A new annotated data set of 136 data constraints and their implementations, from 7 real-world Java systems, which complements data from existing research.
- The results of the evaluation of two Lasso instances and two baseline approaches.

The data, code, and results are in our replication package [28].

2 BACKGROUND

In this paper, we use the *constraint implementation pattern* (CIP) catalog defined by Florez *et al.* [27]. To make the paper self-contained, we summarize here the most important information we use.

Table 1: CIP Catalog [27] Excerpt.

CIP name: binary-comparison. **Description:** Two values are compared using an operator such as *equals, does not equal, greater than, etc.* **Statement type:** Expression. **Parts:** {*variable1*, *op* ∈ {>, ≥, <, ≤, =, ≠}, *variable2*} **Example:** *Instance:* `if(maxFreq > wave.getNyquist())` *Parts:* {`maxFreq`, >, `wave.getNyquist()`}.
CIP name: if-chain. **Description:** A chain of ifs is used like a switch on a field, checking against the possible values of the variable. **Statement type:** If statement. **Parts:** {*variable*} **Example:** *Instance:* `if(onset == EMERGENT) {...} else if(onset == IMPULSIVE) {...} else if ...` *Parts:* {`onset`}.

Florez *et al.* studied the textual formulation and line-of-code implementation on a set of 187 constraints from 8 Java systems. The constraints were categorized into four types:

Value comparison has 2 *operands*. The value of *X (variable)* is constrained by the value of *Y (variable)* with an equality or relational operator. Example: "*SWARM will allow the maximum frequency to be set to any positive value greater than the minimum frequency*" [64] contains two constraints: *max frequency* > 0" and "*max frequency* > *min frequency*".

Dual value comparison has 2 *operands*. *X (variable)*, and *Y (condition)* is one of the 2 mutually-exclusive values that implies the other (*e.g.,* true/false, enabled/disabled). Example: "*If configuration file is not available or readable…*" [64] contains two constraints: "*file is available*" and "*file is readable*".

Categorical value has 3+ *operands*. *X (variable)* is constrained to a finite set *S (2+ options)* of two or more values. Example: "*onMissingExtensionPoint: What to do if this target tries to extend a missing extension-point. (fail, warn, ignore)*" [4] contains one constraint: "*onMissingExtensionPoint* ∈ {*fail*, *warn*, *ignore*}".

Concrete value has 2 *operands*. The constraint directly declares *X (variable)* to be *C (value)*. Example: "*The default [switch] date is 1582-10-15.*" [39] contains a constraint: "*switchDate* is *1582-10-15*".

The implementation of a constraint consists of two parts: *an enforcing statement* and *data definition statements*. Each enforcing statement was categorized according to the type of code construct where it appears (*e.g.,* expression, `if` statement, `return` statement, *etc.*) and the number of operands it uses. The result was the definition of 30 CIPs, *i.e.,* code patterns with line-of-code granularity. In this paper, we only utilize information about 13 CIPs, which were identified as the most commonly used: *boolean property*, *binary comparison*, *constant argument*, *null check*, *assign constant*, *binary flag check*, *if chain*, *equals or chain*, *switch len char*, *self comparison*, *return constant*, *null zero check*, and *null empty check*. Table 1 shows the definition of 2 of the 13 patterns. The complete CIP Catalog can be found in the original publication [27].

For each CIP, we use the description and the number of required operands (which we convert into a grammar - Section 3) . In addition, for each constraint type, Florez *et al.* identified which CIPs are most commonly used for implementing constraints of that type.

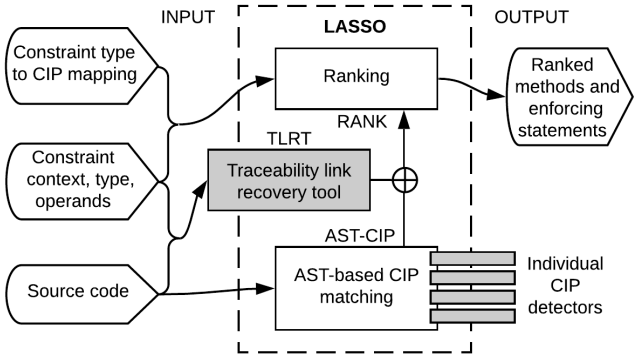

Figure 1: Overview of Lasso. Gray boxes are replaceable components. There can be any number of CIP detectors.

For example, constraints of *dual value comparison* type are most frequently implemented using the *boolean property* CIP, with the next most common CIP being *null check*. Lasso also leverages this information as part of its ranking model.

3 APPROACH DESCRIPTION

Our approach, Lasso (**L**ocating d**A**ta con**S**traints in **S**ource c**O**de), is designed as an extensible framework that automatically identifies implementations of given data constraints in the source code. An overview of the Lasso framework is presented in Figure 1.

The main novelty of Lasso is its ability to *identify constraint implementations at line-of-code granularity*. This is enabled by the AST-based CIP matching component (AST-CIP, see Section 3.2). AST-CIP is composed of several CIP detectors. Each one can detect instances of one CIP in the source code by traversing the abstract syntax trees (ASTs) of the target system. As shown in Table 1, Florez et al. [27] defined each CIP using natural language description and an example. Such definition is ambiguous and can not be used for specifying the pattern for each detector. Based on their definitions, we define a syntactic pattern of each CIP, using a context-free grammar (CFG). AST-CIP takes the source code as input, and is composed of a number of detectors, each capable of identifying instances of one CIP. The user can make a choice of how many CIP detectors to provide. Adding more can result in more constraints being correctly traced. AST-CIP produces a set of source code statements, *i.e.*, enforcing statement candidates (ESCs), that match the syntactic patterns of the CIPs used by the detectors (Section 3.2.2).

In addition, Lasso uses a traceability link recovery tool (TLRT), that accepts natural language (*i.e.,* the description of a data constraint) and source code as inputs. The TLRT retrieves relevant source code methods for the constraint. Lasso is agnostic to the internal retrieval model of the TLRT, which means the framework can use any TLRT, as long as it uses the appropriate input and produces output at method-level granularity. We implemented three instances of Lasso, using Lucene-, VSM-, and LSI-based TLRT (Section 4).

Finally, Lasso's ranking component (RANK) uses the ESCs identified by the AST-CIP component and the method-level results returned by the TLRT to produce a list of methods, ranked by their likelihood that they implement a given data constraint, and a list of code statements inside each method, ranked by their likelihood to enforce the constraint (Section 3.3).

> While SWARM will allow the maximum frequency to be set to any positive value greater than the minimum frequency, this value will adjust automatically if **it is greater than the Nyquist frequency of the wave** being manipulated.

Figure 2: Example of constraint input from the Swarm system. The entire text is the context. The constraint in bold, and the operands are highlighted in gray.

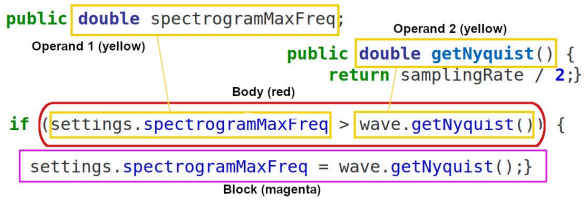

Figure 3: The operands, body, and block elements of the ESC that implements the constraint in Figure 2.

3.1 Lasso Inputs

For each target system, Lasso takes as input its *source code*, which is used by the TLRT and the AST-CIP components.

The *Constraint type to CIP mapping* (Section 2) contains information about the type of data constraints and frequency of existing CIPs implementing each constraint type. This information is employed by the user to describe the constraint and also by the ranking component to estimate the likelihood that a certain CIP implements a given constraint type. For each constraint, three elements are specified by the user and used as queries by Lasso.

1. The *constraint context* is the paragraph where the constraint is described in the existing documentation (*e.g.,* requirements, use cases, and manuals). This is used by TLRT.

2. The *constraint type* is set according to the definitions in the constraint type catalog. The CIP catalog we use defines four constraint types: *value comparison*, *dual value comparison*, *categorical value*, and *concrete value* (Section 2). This is used by RANK.

3. The *constraint operands* are the noun phrases referring to the data on which the constraint is defined. The number of operands varies for each type of constraint (Section 2): (1) two operands for *value comparison*; (2) two operands for *dual value comparison*; (3) three or more operands for *categorical value*; and (4) two operands for *concrete value*. This information is used by RANK.

Figure 2 shows an example of the constraint inputs that Lasso takes. The *constraint context* is all the text in the figure. The *constraint type* (derived from the text describing the constraint) is *value comparison*, as it matches the definition "a value is constrained by another value using a relational operator" (*greater than* in this case). The *operands* of this constraint are [maximum frequency, Nyquist frequency of the wave]. Note that reference resolution is not done automatically, but left to the user, who in this case should select the noun phrase "maximum frequency" instead of the pronoun "it".

3.2 AST-CIP component

The AST-CIP component uses the source code and the constraint detectors to identify enforcing statement candidates (ESCs).

3.2.1 ESC definition. Each ESC has the elements:

- A *body*, *e.g.*, the AST element that matches the CIP, which can be a field definition, a statement, or an expression.
- A *method* where it appears.
- A list of *operands*.
- An optional *block* of statements.
- A *CIP type*, corresponding to the CIP that it matches.

The number of *operands* of an ESC depends on its *CIP type*, as described in Section 2. Figure 3 shows the ESC that implements the constraint in Figure 2. The ESC's *CIP type* is *binary-comparison*; hence, it has has two *operands* (based on the CIP catalog we use). Lasso aggregates the *terms of an operand*, by collecting both the identifiers contained inside the ESC *body* ("wave.getNyquist" for operand 2, in the example), and the identifiers in the *operand*'s definition ("double getNyquist()" for operand 2, in the example). The definitions are obtained by symbol resolution, and can be done with any analysis framework that provides the functionality.

The *block* element applies only to the ESCs whose *body* appears in the condition of a `if`, `while`, or `do-while` statement. The *block* corresponds to all the text in the body of the statement, including both *then* and *else* blocks in the case of `if` statements. The ESC *block* does not apply to ESCs whose body appears between the parentheses of a `for` statement, because these conditions are more complex than those of the previously mentioned statements.

Note that some ESCs may appear outside of methods, *e.g.*, in field definitions. In this case, the corresponding definition is considered to be a *pseudo-method*.

3.2.2 CIP matching. As discussed, the ambiguity in the CIP definitions by Florez *et al.* [27] makes it hard to specify which instances should be matched by each CIP detector. To address this challenge, we express the CIPs using a context-free grammar (CFG), shown in Figure 4. We define garmmars for 13 most common CIPs (Section 2). The grammar of each CIP covers all instances of the pattern in Florez's data. By convention, non-terminals are in uppercase and terminals are in lowercase. The non-terminals in blue are the start symbols for matching the 13 CIPs. In this grammar, a terminal or non-terminal may be associated with an *operand specifier*, which is an annotation following a colon, *e.g.*, `op in BOOL_VAL:op` on line 1. Operand specifiers are used to identify the symbols that are operands in each CIP. The same operand specifier defines the same symbol in different places in a CIP's grammar. For example, **`var`**`:op` on lines 23 to 25 requires the same variable to appear in these conditional expressions.

Lasso can have any number of CIP detectors. To identify ESCs, Lasso first parses all files in the source code except for the test files to generate ASTs. Test files are ignored because these do not contain the implementation of business rules (in this case data constraints). Investigating the association between data constraint implementations and their test code is subject of future work. Lasso then visits every node in each AST. When an AST node is visited, AST-CIP attempts to match the grammar of each detector on said node. An individual CIP detector will return an ESC *if and only if the node is a valid production of the detector's corresponding grammar.* If the node is not a valid production, the detector returns nothing. Traversing all nodes in all ASTs results in the final list of ESCs.

```
1  BOOLEAN_PROPERTY → BOOL_VAL : op
2  BOOL_VAL → var_bool | method_call_bool | field_access_bool
3
4  BINARY_COMPARISON → VAL : op1 CMP VAL : op2
5                    | VAL : op1 RELOP VAL : op2
6  CMP → < | > | <= | >=
7  RELOP → == | !=
8  VAL → var | method_call | field_access
9
10 CONSTANT_ARGUMENT → var . m_name : op1 ( ARGS literal : op2 ARGS )
11 ARGS → expr ARGS | λ
12
13 NULL_CHECK → null RELOP VAL : op
14              | VAL : op RELOP null
15
16 ASSIGN_CONSTANT → VAR : op1 = literal : op2
17 VAR → var | field
18
19 BINARY_FLAG_CHECK → INT_VAL : op1 BITOP INT_VAL : op2 RELOP
       lit_int : op2
20 INT_VAL → var_int | method_call_int | field_access_int
21 BITOP → & | |
22
23 IF_CHAIN → if ( var : op == literal ) BODY ELSEIF
24 ELSE_IF → elseif ( var : op == literal ) BODY ELSE
25 ELSE → elseif ( var : op == literal ) BODY ELSE | λ
26
27 EQUALS_OR_CHAIN → var : op == literal CHAIN
28 OR_CHAIN → || var : op == literal CHAIN
29 CHAIN → || var : op == literal CHAIN | λ
30
31 SWITCH_LEN_CHAR → switch ( VAL : op . length() ) LEN_CASE
32 LEN_CASE → case literal_int : stmts LEN_CASE | λ
33
34 SELF_COMPARISON → var : op RELOP var : op
35
36 RETURN_COSTANT → return literal : op1
37
38 NULL_ZERO_CHECK → NULL_CHECK AND_OR var : op . length ( ) > 0
39 AND_OR → && | ||
40
41 NULL_EMPTY_CHECK → NULL_CHECK AND_OR var : op . equals ( " " )
```

Figure 4: Grammar for 13 CIPs.

To produce all elements of the ESC as result, each detector uses the matching AST node (ESC *body*), the method where the matching node appears (ESC *method*), the specific detector (ESC *CIP type*), and the list of operands (ESC *operands*). The number of operands differs by CIP, and is extracted by each detector based on the grammar.

In addition to the operand identifiers in the ESC *body*, Lasso also includes the text from their definitions (see Figure 3). The definition of each operand is resolved using symbol resolution, yielding one of the possible definitions: field definition, method definition, variable/parameter definition. In case the operand definition is a method definition, only the identifiers corresponding to the method name, parameter names, and parameter types are added to the operand. For parameters, only the parameter name and type are added to the operand. For the remaining types of definitions, all identifiers in the defining statement are added to the operand.

Consider the example in Figure 3; when the Lasso reaches the AST node corresponding to the `if` statement condition, it attempts to match the grammar of each detector. All detectors but the one for *binary comparison* will return nothing, as this node matches no other CIPs. The given node matches the CIP, as it is a binary expression using a relational operator. After finding that the node matches the grammar, the detector identifies the operands. In this case there are two operands, as per the CIP definition; they are `settings.spectrogramMaxFreq` and `wave.getNyquist()`. Symbol resolution is applied as appropriate for each operand: in the first case, the field is resolved, and in the second case, the method is resolved. The identifiers found in the definitions are then added to the terms of the corresponding operand.

3.3 Ranking component

RANK uses four types of information from the inputs (*i.e.,* user input and CIP catalog) and from the output of the other two components (*i.e.,* TLRT and AST-CIP) to rank all ESCs for a given constraint:

- The *constraint context*, the *constraint type*, and the *list of constraint operands*, from the user input.
- The ESC *CIP type*, the ESC *method*, the list of ESC *operands*, and the ESC *block*, from the AST-CIP output, for each ESC that matches any of the CIPs implemented in the detectors.
- Which CIPs are used most frequently to implement constraints of the given *constraint type*, from the constraint type to CIP mapping.
- The ranked list of methods produced by the TLRT.

We introduce abbreviated notations for the various information used in the ranking. For the constraint elements we use: C_c (*constraint context*), C_t (*constraint type*), and C_o (*list of constraint operands*). For the ESC elements we use: E_t (ESC *CIP type*), E_m (ESC *method*), E_o (list of ESC *operands*), and E_b (ESC *block*).

Lasso first applies standard text retrieval preprocessing techniques to all textual fields of both constraint inputs (C_c, C_o) and ESCs (E_o, E_b). Specifically, we apply identifier splitting (based on *camelCaseFormat* and *underscore_format*, stemming using the Porter algorithm [54], and stop word removal (the list of stop words is available in our replication package [28]). Lasso then uses C_c or C_o (as appropriate) as input to TLRT and obtains a ranked list of methods, which we denote as U_m.

Given the input constraint, for each ESC identified by the AST-CIP, Lasso uses **four heuristics** for computing a relevance score:

1. The common terms between the *constraint operands* (C_o) and the ESC *operands* (E_o). The intuition is that if the operands from the constraint description match the ESC operands, then the ESC is likely to implement the input constraint, *e.g.,* the term "Nyquist" appears in one constraint operand and one ESC operand in Fig. 3.

2. The common terms between the *constraint operands* (C_o) and the ESC block (E_b). In some cases the ESC *body* uses terms different from those in the constraint (*e.g.,* `i <= j`), so the previous heuristic will not find common terms. However, operations in the ESC block may indeed use these terms if the logic is related to the constraint. Matching *constraint operand* terms with terms in the ESC *block* increases the likelihood that the ESC enforces the given constraint.

3. The frequency of the ESC *CIP type* E_t, with respect to the constraint type C_t, extracted from the CIP catalog. Previous research by Florez et al. [27] (see Section 2) identified which constraint types are implemented by which CIPs. For example, they found that 59% of constraints of dual value comparison type are implemented using the boolean-property CIP, while 17% are implemented with the null-check CIP. Hence, if the ESC *CIP type* matches the former, then it is more likely it implements the given constraint.

4. The TLRT rank of the ESC *method*, E_m. If the ESC *method* is ranked high by the TLRT, then it is likely that the ESC implements the given constraint and less likely if the rank is low.

For these heuristics, we define *five measures* (two for the first one, and one each for the others) that take values between 0 and 1.

1. *COE* (*Constraint-ESC operands*). Lasso pairs each operand o_i in C_o to the operand o_j in E_o with which it has the largest number of terms in common. The pairing is strictly one-to-one. Then the *COE* measure is calculated according to Equation 1.

$$COE = \frac{\sum_{i,j} s(o_{ci}, o_{ej})}{|C_o|} \tag{1}$$

Where s returns the percentage of terms in the constraint operand o_{ci} that are also in the ESC operand o_{ej}. Unpaired elements (*e.g.,* if the constraint has more operands than the ESC) have a value of zero for s. For the example in Figure 3, the second constraint operand o_{c2} "Nyquist frequency of the wave" gets paired with the second ESC operand o_{e2} `wave.getNyquist()`. The first constraint operand o_{c1} "max frequency" gets paired with the remaining ESC operand o_{e1}, though they have no terms in common. The value of the measure is then $COE = \frac{s(o_{c1},o_{e1})+s(o_{c2},o_{e2})}{|C_o|} = \frac{0+0.4}{2} = 0.2$. The value of $s(o_{c2}, o_{e2})$ is 0.4 because the second constraint operand has two out of five terms in common with the second ESC operand: Nyquist and wave.

2. *ECO* (*ESC-constraint operands*) performs the same pairing as *COE*. This time, the *ECO* measure captures the percentage of terms from each ESC operand o_j that in common with the constraint operand o_i. For the example in Figure 3, $ECO = \frac{s(o_{e1},o_{c1})+s(o_{e2},o_{c2})}{|E_o|} = \frac{0+0.5}{2} = 0.25$. The value of $s(o_{e2}, o_{c2})$ is 0.5 because o_{e2} has four terms (wave, get, Nyquist, double), and two of them match the constraint operand: Nyquist and wave. The reason we perform the matching both ways (*COE* and *ECO*) is that even though an ESC may contain most or all terms in the constraint operands, also including a lot of unrelated terms suggests that it might be dealing with different data.

3. *COB* (*Constraint-operand block*) is the percentage of terms in all elements of C_o that appear in E_b. We do not perform the matching in the opposite direction (analogous to *COE* and *ECO*) because the ESC blocks can be long, and hence contain a lot of different terms, causing the values to be too small to make a difference in the score. This score is based on the intuition that if the names of operands are used in the body of the conditional statement, then the ESC is more likely to be relevant compared to the case in which the operands only appear in the condition.

4. *ECIP* (*Expected CIP*). Based on the information from the CIP catalog, *ECIP* is 1.0 if E_t is the most frequently used pattern implementing C_t, 0.5 if it is the second most frequent, and 0.0 otherwise. Specifically, *value comparison*: [*binary comparison*]; *dual value comparison*: [*boolean property*, *null check*]; *concrete value*: [*constant argument*, *assign constant*]; *categorical value*: [*if chain*]. Notice that

in the cases of *value comparison* and *categorical value* there is only one "most common CIP". In these cases we considered that the second most common CIP did not appear in enough instances to make it common enough, so only one CIP gets the full score (1.0) and all others get zero.

5. *CM* (*Context method*). *CM* is $1/\sqrt{r}$ where r is the rank of the E_m in U_m, or 0 if TLRT did not return that method.

The final *relevance score* for an ESC is computed according to Equation 2. Namely, the relevance score (rs) for an ESC (e) is equal to the sum of the weighted values of the five measures for that ESC (*i.e.,* *COE*, *ECO*, *COB*, *ECIP*, and *CM*).

$$rs(e) = \sum_{i=1}^{5} m_i(e) * w(m_i) \quad (2)$$

Where each $m_i(e)$ is the value of a measure and each $w(m_i)$ is its corresponding weight.

The ESCs are sorted in descending order of their relevance score; those with a score of zero are omitted from the result list. Results that only have a non-zero value for *CM* are also omitted, as our tool prioritizes results found via CIP matching. At most one ESC is returned for each line of code in each file, with only the ESC with the highest score returned for each line.

The ESC rankings are converted to method level by ranking each distinct E_m (produced by the TLRT) in the same order as they appear in the ranked ESC list, without repetition. For example, if ESCs in ranks 1, 2, and 100 are in method A, and ESCs in ranks 3 and 4 are in method B, method A will be ranked first, and method B will be ranked second. The list of ESCs is preserved alongside each method, and they appear in the same order as in the original list of ESCs. Continuing the above example, method A will have an ESC result list with 3 ESCs, and method B will have 2 ESCs in its list.

4 LASSO INSTANTIATION

We implemented 13 CIP detectors, one for each of the frequently occurring CIPs (see Section 2). We used the JavaParser library [38] to implement these detectors. For this, we employed the parsing capabilities of the library to parse the source code files, and used AST visitors to implement each detector as specified by the corresponding grammar. We also relied on the library's symbol resolution to resolve operand definitions.

We used these detectors to formulate three Lasso instances:

- Lasso-13Luc, which uses as TLRT Lucene 8.6.3 [48] with its default similarity metric, implementing the BM25 model [57].
- Lasso-13VSM, which uses as TLRT Lucene 8.6.3, with its classic similarity metric, implementing the Vector Space Model (VSM) [32], more specifically, TF-IDF.
- Lasso-13LSI, which uses as TLRT Latent Semantic Indexing (LSI) [22].

The three instances also differ in what text is provided as input to TLRT: in the case of Lasso-13Luc and Lasso-13VSM, the constraint *context* is used, while for Lasso-13LSI it is the concatenated terms of all the constraint *operands*. These inputs were selected because they achieved the highest performance for each TLRT according to a preliminary test. The results of such test can be found in our replication package [28]. The three TLRT components use the same text processing, as described in Section 3.3 and return results at the method level. For LSI, we used a dimension parameter of 300. This parameter yielded the best performance on our data according to our preliminary tests (results found in our replication package [28]). These two text retrieval-based traceability link recovery techniques have been commonly used as baselines in prior studies on traceability link recovery [6, 11, 42] and bug localization [16, 50, 52].

The Lasso instances and the two TLRT work on Java code.

Table 2: Software systems in the validation data set (VDS).

System	Domain	KLoC	MTH [a]
mybatis-3.5.5	Persistence Framework	60	3,639
shardingsphere-5.0.0-rc1	DB Sharding middleware	110	9,599
skywalking-8.0.1	App. Perf. Manager	127	10,825
jabref-5.0	Citation Manager	130	12,796
jpos-2.1.4	Finance library	175	8,291
log4j-2.13.3	Logging Framework	191	15,952
checkstyle-8.35	Source Code Style Checker	276	6,379

[a]: Number of methods

5 EVALUATION

The goal of our evaluation is to assess how effective Lasso is in locating the data constraint implementations at both method and line-of-code level, as it works at both granularities.

Our evaluation answers two research questions:

- **RQ1:** What is the performance of Lasso-13Luc, Lasso-13LSI, and Lasso-13VSM on method-level data constraint traceability link recovery?
- **RQ2:** How accurately can Lasso-13Luc, Lasso-13LSI, and Lasso-13VSM retrieve the lines of code that implement a constraint?

5.1 Experimental Setup

Subjects of the study. We perform an intrinsic evaluation by comparing Lasso-13Luc and Lasso-13VSM with the Lucene-based TLRT they are built upon. Likewise, we compare Lasso-13LSI with the LSI-based TLRT.

Datasets. We need ground-truth datasets where the traces from the data constraints to their implementations (lines of code) are known. We use two datasets in the evaluation. The first is the data published by Florez et al. [27], which we refer to as the Calibration Data Set (CDS), since we use it for calibrating the weights of the five measures used by the Lasso instances for ranking (see Section 3.3). We describe the calibration below. CDS consists of 163 traced constraints in 8 real-world Java systems [27]. That study had a dataset of 187 constraints. We selected only the 163 constraints implemented with one of the 13 CIPs detected by Lasso-13.

We collected an additional dataset, called Validation Data Set (VDS), consisting of 136 constraints from 7 new systems different from the ones in CDS (Table 2). The textual artifacts for all systems

are their corresponding user manuals. We used the following protocol to construct VDS, which is similar to the one used by Florez et al. [27]. First, one author examined the textual artifacts of the 7 systems and extracted 30 data constraints from each. Next, we recruited four tracers to identify the implementations of these constraints. All tracers are Computer Science graduate students, each with at least 3 years of programming experience in Java.

The tracers were instructed to find a single implementation per constraint, specifically the one described in the textual artifact where the constraint originated. The reasons for this are aligned with those argued by Florez et al. [27]. Namely, (1) it is very difficult to define a stopping criterion that would consistently result in all implementations of a constraint being reliably found, and (2) term mismatch and implementation complexities make it necessary to use the constraint context to have a reasonable certainty that the trace is correct, which means only the instance of the constraint defined in the text can be reliably traced.

Each constraint was assigned to two tracers.The tracers produced identical traces for 116 constraints (55% of 210). The disagreements were caused either from misunderstandings of the code semantics (as the code does not always use the same terms as the constraint's textual description), or from tracers selecting a statement that does not refer to the specific implementation of the assigned constraint, but rather one in related functionality. This relatively low agreement rate is to be expected due to the complex nature of the task, which requires the tracers be familiar with the target system's code. To ensure the quality of VDS, two authors determined the final trace through discussion based on the tracers' answers. Finally, to determine which constraints were suitable for our evaluation, one author labeled each trace with the CIP that corresponded to it. Only those constraints implemented with one of the 13 CIPs for which we implemented detectors were added to the VDS data set, resulting in 136 constraints in total.

Metrics. Because both CDS and VDS contain traceability links at line-of-code granularity, the lines of code for each trace correspond to the *ground truth* for each constraint. Both Lasso instances and the baselines produce ranked lists of methods as their outputs. To measure their effectiveness, if a method in the result list contains the ground truth lines, we consider the constraint as retrieved at that rank. We call this the *method rank* of the constraint. Recall that the Lasso instances also produce a ranked ESC list for each method. We define the *ESC rank* as the position of the ESC containing the ground truth lines in the ESC corresponding ground truth method.

As we discussed above, each constraint in CDS and VDS has only one ground truth. This means that commonly used information retrieval metrics are not very meaningful here. Specifically, MAP [55] will always have the same value as MRR (1 divided by method rank, or 0 if the constraint was not retrieved). Precision will always be either 0 if the ground truth was not retrieved, or 1 divided by the amount of results if it was. Recall will either be 1 if the ground truth was retrieved, and 0 if not. In light of this, we report MRR, average recall, and the average method rank.

These metrics are provided for completeness, as we focus our analysis on the %HITS@N metrics, defined as the percentage of constraints with a *method rank* between 1 and N, *i.e.,* constraints where the ground truth was retrieved within the first N methods of the output ranked list. This metric is easy to interpret, as it means that a potential user has to examine N results (in this case methods) to find the constraint implementation. The importance of this metric (sometimes under different names) has been argued in the fields of bug localization [66, 70], query reformulation [15, 16, 26, 55], and duplicate bug report detection [17, 58].

Input generation. As discussed in Section 3.1, Lasso requires the type, operands, and context of each constraint as input. To construct the input, one author extracted the required input fields for each constraint. Specifically, the constraint context is a paragraph where the constraint is found in the textual artifacts. The constraint type was assigned as one of the four constraint types in Sec. 2. The operand list is composed of the noun phrases that describe each operand, using only terms found in the constraint context, except in the case of numbers. If numbers were spelled out, they were turned into digits, *e.g.,* "one" became "1". Additionally, symbols were also spelled out, for example "∞" becomes "infinity". Sec. 3.1 shows an example of specific input for a constraint. The inputs for all 299 constraints can be found in our replication package [28].

Baseline calibration. To find the optimal input (*i.e.,* constraint context or constraint operands) for the TLRT of each Lasso-13 instance and the best parameters for LSI, we evaluated the baselines on the combined CDS and VDS data sets using each input with each technique. The best performing combinations are explained in Section 4, and the full results are in our replication package [28].

Calibration of Lasso ranking weights. We empirically calibrated the weights for the five ranking measures (Sec. 3.3), using CDS. We designed an algorithm to find the combination of weights that result in the best results for Lasso-13Luc in CDS. We used Lasso-13Luc for calibrations, as opposed to Lasso-13LSI or Lasso-13VSM, because Lucene with its default similarity performed better than the other two approaches. That is, we optimized the weights based on the strongest baseline.

The calibration algorithm runs 5 rounds of testing (one for each weight). Each metric begins as a free metric, and at the end of each round, one metric will become fixed with a weight. On each round, for each free metric, the algorithm generates scenarios corresponding to all combinations of weights for all free metrics (one of $\{0.0, 0.1, 0.2, \ldots, 0.9, 1.0\}$) and the fixed weights. The value of the free weight that leads to the highest value of %HITS@20 becomes fixed. The optimal configuration for Lasso-13Luc, for CDS, is $COE = 0.7$, $ECO = 0.2$, $ECIP = 0.2$, $COB = 0.2$, $CM = 1.0$. As mentioned above, we use the same configuration for Lasso-13LSI and Lasso-13VSM, as we want to assess how robust these weights are to changes in the data sets and TLRT.

5.2 RQ1 Results

Table 3a shows the results obtained by Lasso-13Luc, Lasso-13LSI, Lasso-13VSM, the Lucene-based TLRT, the VSM-based TLRT, and the LSI-based TLRT, on the calibration data set (CDS), with 163 constraints. Table 3b shows the results obtained by the six approaches on the validation data set (VDS), with 136 constraints, while Table 3c shows the results on the combined data sets, with 299 constraints.

5.2.1 Weight calibration validation. To ensure that the configuration was not overfitted to the test data, we evaluated the Lasso instances on the validation data set with the configuration obtained from the algorithm in Sec. 5.1. Comparing the results from Table 3a

and Table 3b, we observe that all three Lasso-13 instances achieve improvements on both data sets. Lasso-13Luc improves %HITS@10 from 39.3% to 57.7% (47%) over the Lucene baseline on CDS, and from 66.2% to 77.9% (18%) on VDS. Similarly, Lasso-13VSM improves %HITS@10 from 22.7% to 49.7% over the VSM baseline on CDS, and from 44.1% to 72.8% on VDS. Finally, Lasso-13LSI improves %HITS@10 over the LSI baseline from 12.9% to 39.9% (209%) on CDS, and from 27.9% to 66.2% (137%) on VDS. This indicates that the weight calibration is robust and transfers to other data sets.

From here on, we perform all analyses on the results on the combined data set (*i.e.,* CDS + VDS).

5.2.2 Lasso-13Luc vs. Lucene-based TLRT. Table 3c shows the results obtained by Lasso-13Luc and the Lucene-based baseline on the combined data sets.

As discussed before, %HITS@N indicates the percentage of constraints for which the relevant method is retrieved in top N In theory, in such cases, the users need to check at most N methods to find the relevant one. Prior research on traceability link recovery argued that retrieving the ground truth on the top position is perfect performance, returning it in the top 5 is excellent, and in top 10 very good [15, 16]. Note that the number of methods in the target systems is 9.6k on average (Table 2). We focus the analysis of the results on %HITS@10 (*i.e.,* indicating very good performance).

We observe that Lasso-13Luc obtains 25.4% %HITS@1, which means that for one in four constraints, Lasso-13Luc retrieves the relevant method in the first place. At %HITS@10, Lasso-13Luc improves the baseline approach by 30% (66.9% vs 51.5%). In other words, for two third of the constraints Lasso-13Luc retrieves the relevant method in top 10, compared to half the constraints for the baseline. Lasso-13Luc also improves the %HITS@1 and %HITS@5 results over the baseline by 44% and 42%, respectively.

Note how Lasso-13Luc improves the average method rank from 148.9 to 24.6, a reduction of one order of magnitude.

The lower recall for both Lasso-13 instances vs. their baselines is to be expected, and also both return the same results despite having different TLRT (*i.e., Avg. Recall* column in Table 3 is the same for both approaches). This is because (as explained in Sec. 3.3), only the ESCs that have a value for the first 4 metrics are retrieved, meaning that which ESCs are returned does not depend on the results of TLRT. Instead, the TLRT results are used as part of the ranking of these ESCs, and as such, both Lasso-13 instances achieve different values of %HITS@N, average method rank, and MRR, as expected.

5.2.3 Lasso-13LSI and Lasso-13VSM vs. baseline TLRT. Table 3c shows the results obtained by Lasso-13LSI, the LSI-based baseline, Lasso-13VSM, and the VSM baseline on the combined data sets. We observe that Lasso-13VSM improves the %HITS@10 results of the VSM baseline by 86% (60.2% vs. 32.4%), while Lasso-13LSI has an improvement of 163% (51.8% vs. 19.7%) over its baseline. We note that the VSM baseline performs worse than the Lucene-based one by 37% in terms of %HITS@10, while the LSI baseline performs 62% worse. The improvement in terms of average rank for Lasso-13VSM over its baseline is similar to that achieved by Lasso-13Luc over its baseline, namely, one order of magnitude. With that in mind, the size of the improvement obtained by Lasso-13LSI (compared to Lasso-13Luc or Lasso-13VSM) indicates that Lasso is especially well suited to improve a poorly performing baseline TLRT. Notably, the reduction in average rank is even more dramatic for Lasso-13LSI, going from 2,286.2 to 41.4, or two orders of magnitude.

5.2.4 Analysis of the results. We perform a deeper analysis of the Lasso-13Luc results, given it performs better than the other two Lasso variations. We examined a random sample of 30 constraints: 10 where the ground truth was not retrieved, 10 where it was retrieved with method rank 11-20, and 10 where the method rank was 21+.

The most common reason for not retrieving or low-ranking of the ground truth was a term mismatch between the operands in the constraint text and the operands of the ESC, which is a known problem in traceability link recovery. Specifically, we identified the following causes of term mismatches:

(1) uses of abbreviations (ArgoUML constraint "[Minimise Class icons in diagrams] is enabled by default", which appears in the source code as "mini");
(2) compound identifiers (Ant constraint "If the value of [clonevm] is true", clonevm vs. isCloneVm); and
(3) misspellings (ArgoUML constraint "Use guillemots («») for stereotypes (clear by default)" *guillemots* in the text vs the correct *guillemets* in the code).

The standard text retrieval techniques that Lasso-13 uses to process the text in operands cannot successfully overcome these mismatches. There are, however, techniques that have been specifically designed to tackle these situations (*e.g.,* abbreviation expansion [36, 41], identifier splitting [25, 30], spell checkers [5, 58]), which could be easily integrated into our approach.

The next most common cause for low rankings is the TLRT high scores for unrelated methods. This happens mostly because of terms in the constraint context that are very common on the system or that happen in combinations that the BM25 scoring considers to be very relevant. For example, for the Ant constraint "default for cache still is false" the ground truth ESC has maximum value for *COE* and *ECO*, but a value of only 0.03 for *CM*. This happens for two reasons: (1) the ground truth ESC is in a field definition, which only has two matching terms with the constraint context (cache and false); (2) BM25 ranks longer methods with the word combination "resource collection" (from the context) near the top of its list.

Implementation decisions can also cause this problem: for the Log4J constraint "If [locationInfo is] true", there are 6 classes in the system with "locationInfo" properties, plus a class actually named "LocationInfo". The usages of any of these symbols are ranked highly not only by the TLRT, but also by Lasso-13 specific scoring, since they also have operand terms in common. For a MyBatis constraint, the first 8 ranked methods contain the same error message: "Error: Cannot rollback. No managed session is started.", which matches terms form the context and causes them to be ranked highly. This appears to be a commonly used pattern in this system that hinders our approach, but can be addressed by removing error messages from the ESC block. One way to address these situations is to use the code around the ESC to better understand of the semantics of the ESC. Such exploration is subject of future work.

Finally, an expected type of constraint implementation that is difficult for Lasso-13 to retrieve is the case where the enforcing statement is used to check multiple constraints. These were described by

Table 3: Retrieval performance of Lasso-13 variations and their corresponding baselines.

Data Set	Technique	%H@1	%H@5	%H@10	Avg. Method Rank	Avg. Recall	MRR
CDS	Lasso-13Luc	14.1% (23)	45.4% (74)	57.7% (94)	35.4	81.6%	27.5%
	Lucene	11.0% (18)	27.6% (45)	39.3% (64)	218.3	98.2%	19.8%
	Lasso-13VSM	11.7% (19)	39.9% (65)	49.7% (81)	39.4	81.6%	22.9%
	VSM	2.5% (4)	13.5% (22)	22.7% (37)	270.5	98.2%	8.3%
	Lasso-13LSI	11.0% (18)	27.6% (45)	39.9% (65)	58.6	81.6%	19.6%
	LSI	4.9% (8)	9.2% (15)	12.9% (21)	3309.6	98.8%	7.3%
(a) Results on CDS (163 constraints)							
VDS	Lasso-13Luc	39.0% (53)	66.9% (91)	77.9% (106)	12.9	89.7%	51.5%
	Lucene	25.7% (35)	52.2% (71)	66.2% (90)	66.0	98.5%	37.5%
	Lasso-13VSM	39.7% (54)	64.0% (87)	72.8% (99)	13.5	89.7%	50.5%
	VSM	18.4% (25)	38.2% (52)	44.1% (60)	90.1	98.5%	27.6%
	Lasso-13LSI	29.4% (40)	52.2% (71)	66.2% (90)	22.7	89.7%	40.9%
	LSI	12.5% (17)	22.1% (30)	27.9% (38)	1065.7	99.3%	18.0%
(b) Results on VDS (136 constraints)							
CDS + VDS	Lasso-13Luc	25.4% (76)	55.2% (165)	66.9% (200)	24.6	85.3%	38.4%
	Lucene	17.7% (53)	38.8% (116)	51.5% (154)	148.9	98.3%	27.8%
	Lasso-13VSM	24.4% (73)	50.8% (152)	60.2% (180)	27.0	85.3%	35.4%
	VSM	9.7% (29)	24.7% (74)	32.4% (97)	188.3	98.3%	17.1%
	Lasso-13LSI	19.4% (58)	38.8% (116)	51.8% (155)	41.4	85.3%	29.3%
	LSI	8.4% (25)	15.1% (45)	19.7% (59)	2286.2	99.0%	12.2%
(c) Results on CDS + VDS (299 constraints)							

Florez et al. [27], and serve as justification for the inclusion of "data definition statements" in the definition of the CIPs. For example, for the SkyWalking constraint "[instance_name] [m]ax length is 50", its corresponding enforcing statement is `if (value != null && value.length() > lengthDefine.value())`. This statement is used to check many properties in the system, namely those that use the "Length" annotation (`lengthDefine` is of type `Length`), defined in the code of the system. Locating this enforcing statement would require identifying usages of the "Length" annotation (relevant in this case "`@Length(50) public volatile static String INSTANCE_NAME = "";`"), and add the concrete value of the annotation to the corresponding operand, in this case `lengthDefine.value()`. While this process would most likely improve the performance of Lasso-13, it is outside of the scope of this paper.

In 9 of the 30 analyzed constraints, a *true positive*, different from the ground truth was retrieved within the first 10 results. This was expected, as Florez et al. [27] documented that in some cases, one constraint is enforced in several places in the code (*e.g.,* when it is involved in multiple features). Since the ground truth data only annotates a single enforcing statement per constraint (even if there are more), we do not count the extra enforcing statements as true positives. Expanding the data sets to annotate all enforcing statements for each constraints is subject of future work.

> Lasso-13Luc outperforms the Lucene-based baseline by 30% (66.9% vs 51.5%); Lasso-13VSM outperforms the VSM-based baseline by 70% (60.2% vs. 35.4%); Lasso-13LSI outperforms the LSI-based baseline by 163% (51.8% vs. 19.7%); all in terms of %HITS@10.

5.3 RQ2 Results

To evaluate how accurately Lasso-13 can point to the correct enforcing statement, we further examine the ESC lists for the evaluation results. We perform all analyses on the combined CDS + VDS data. The ESC ranks for the three Lasso variations are the same, as the TLRT does not change which ESCs the AST-CIP component matches, only the method ranking (this is why we get the same recall for all variations).

Table 4 shows the distribution of the ESC ranks for all 299 constraints. Lasso-13 places the correct enforcing statement in ESC rank 1 for 202 of 299 constraints (68%) of cases, and in 2-3 for 33 (11%). Lasso-13 does not return the correct enforcing statement for 13 constraints (4%) ("None" in table), and does not return the ground truth method for 44 (15%) ("N/A" in table).

We focus on the subset of constraints for which the Lasso-13 instances return method-level results in top 10 (*i.e.,* 200 constraints for Lasso-13Luc, 180 for Lasso-13VSM, and 155 for Lasso-13LSI), shown in Table 4.

Table 4: Distribution of the ESC ranks for constraints with methods retrieved in top 10 by each baseline and also for all constraints.

ESC Rank	Top 10 LSI	Top 10 VSM	Top 10 Luc.	All
1	130 (83.9%)	148 (82.2%)	162 (81.0%)	202 (67.6%)
2 – 3	16 (10.3%)	23 (12.8%)	28 (14.0%)	33 (11.0%)
≥ 4	4 (2.6%)	4 (2.2%)	5 (2.5%)	7 (2.3%)
None	5 (3.2%)	5 (2.8%)	5 (2.5%)	13 (4.3%)
N/A	·	·	·	44 (14.7%)
Total	155	180	200	299

For the 200 constraints, Lasso-13Luc ranks the ground truth enforcing statement as 1 in 162 cases (81%), as 2-3 in 28 (14%), as 4+ in 5 (2.5%), and it does not point to the ground truth in 5 (2.5%). For 155 constraints, Lasso-13LSI ranks the ground truth first in 130 cases (84% of 155), 2-3 in 16 (10%), 4+ in 4 (3%), and does not point to it in 5 (3%). In other words, for the methods returned by Lasso-13Luc in top-10, the approach pinpoints precisely 95% of the ESC within the returned method (81% in the top position and 14% on position 2 or 3). For Lasso-13LSI, this figure is 94% (84% in 1, and 10% in 2).

We manually examined the ESC results for the ground truth of 27 constraints: 10 where the ground truth ESC had a rank of 2-3, 7 where the ESC rank was 4+ (all such cases), and 10 where Lasso-13 did not return the ground truth ESC.

Inconsistent identifier naming in the source code was the largest cause of low-ranked or missed ESCs. For example, the JabRef constraint "If a file is imported" has the terms "file" and "imported" as part of its operands, however, its enforcing statement uses the term "loaded" instead, and "file" appears inside this ESC's block, giving it a score of 0 for *COE*, though it achieves a maximum score for *COB*. However, other ESCs in the method contain these terms because they are related in functionality, but do not enforce the same constraint, for example an ESC containing the identifier `isFileExport` achieves a higher *COE* score and is thus ranked higher. Similarly, the enforcing statement of the JabRef constraint "If there are [parsing] problems" uses the term "warning", which causes it to not be retrieved, while the string passed to the exception reporting the problem does contain the term, which is returned as an ESC.

In three cases, the ground truth enforcing statement could not be found because it checks the opposite condition to the one specified in the constraint, *e.g.,* the JabRef constraint "there are more than two persons in the author list" is implemented as `if (authors.length < 3)`. One way to address these cases is to look for the negation of the constraints as well, but that is subject of future work.

In three cases, the ground truth enforcing statement was returned in position 2, and the ESC ranked first has the same score as the ground truth ESC. This happens because both ESCs have the same number of terms in their body and the same matching terms with the constraint operands. For example, in the JabRef constraint "an export option is also specified", the two ESCs are `f (cli . isFileExport())` and the ground truth is `if (cli . isExportMatches())`. Both have the same number of terms and match only "export".

For three constraints (different from those mentioned in the previous section) we found the top ranked ESC to be a true positive. The Ant constraint "If the manifest is omitted" is checked in two places in the ground truth method. This is because the manifest is loaded on-demand, and so the object can be in a state where the manifest was provided by the user but has not been loaded. The method checks once to see if the manifest has been loaded, and attempts to load it if not, and then again to confirm that the loading returned anything (because it can return null if the manifest was omitted). Since our ground truth data only has annotated one enforcing statement per constraint, Lasso-13 finds additional "true positives" for the constraints that are enforced in multiple places in the code. Florez et al. [27] found that 71 (44%) of the constraints in the CDS are enforced in multiple places. Creating a data set where all enforcing statements are annotated is subject of future work.

> Lasso-13 ranks the correct enforcing statement accurately for 79% of the 299 constraints (68% at rank 1 and 11% at rank 2-3). For the methods returned by Lasso-13Luc in top-10, it accurately pinpoints 95% of the ESC within the returned method (81% in at rank 1 and 14% at rank 2-3).

6 THREATS TO VALIDITY

Our evaluation is implicit and relies on data sets from previous research and new data we produced for this paper. These data sets are built following the same protocol and only contain a single ground truth enforcing statement per constraint, even if there may be more in some cases. The results may be different on data with more complete ground truth annotations.

Our evaluation assumes a "perfect" user, as Lasso-13 is given the correct inputs for each constraint, specifically the operand list and the constraint type, which were derived from each constraint by an author, who is familiar with both the catalog of data constraint types and data constraints, in general. In a real-world situation the user may not be able to provide the correct input in each case. Evaluating how robust Lasso is with respect to incomplete or incorrect input is subject to future work. To mitigate this issue, the input for the Lasso-13 instances and the baselines are the same.

Our evaluation is performed only on constraints that are implemented with the 13 CIPs that Lasso-13 has detectors for. This is by design, as our goal was to evaluate the performance in detecting known patterns (*i.e.,* corresponding to the detectors). Tackling the presence of unknown patterns is part of our future work.

The correctness of the traces in our data sets is also a threat. Even though the level of agreement between tracers is arguably low (55%), we argue this is actually high, due to the very unlikely nature of two tracers agreeing on a line-of-code trace by chance, as our agreement criterion was strict (*i.e.,* the tracers had to report the exact same lines). To mitigate this threat, each trace was set only after a discussion between two authors.

The weight calibration algorithm we used to set the parameters for Lasso-13 might not have generated the optimal combination of weights. However, we show that a relatively simple process with a small data set provides values that are robust against overfitting, hence we believe they will translate to other data sets.

As we any such empirical study, external validity depends on the size of our data. Given the novelty of this work, only two annotated data sets exists for supporting this work (*i.e.,* the one from [27] and the one we created). To mitigate this threat, we annotated data from systems not used in the previous study.

7 RELATED WORK

Lasso is related to automated requirements-to-code traceability link recovery techniques, albeit it focuses on a specific subset of the requirements (*i.e.,* data constraints). Research in this field spans decades and has produced a multitude of techniques that can trace natural language artifacts to source code artifacts. Our work differs from the existing approaches in two main ways: (1) the natural language artifacts are data constraints, a source of information not yet leveraged by any other approaches, and (2) Lasso retrieves both method-level and line-of-code-level code artifact, which has only been achieved with limited success in previous studies.

Improvements in the performance of these techniques have originated from both the use of different retrieval models (VSM [1, 34, 35, 53], probabilistic [1, 2, 11, 19], topic models [34, 43, 47, 53], machine/deep learning [31, 51], AI techniques [6, 63]), and the use of alternate sources of information (code authorship [23, 56], non-functional keywords [45, 46], dynamic analysis [24, 29, 52]). More related to the present work, code structure has been exploited [24, 40, 49, 59, 60], but only in the form of method/class relationships, unlike the line-of-code patterns that enable our approach.

The fine-grained code patterns represent an additional source of information that is orthogonal to those previously studied (presented above), and similar performance improvements as the ones reported here could be expected to be achieved by integrating any of these approaches with Lasso, as the TLRT component.

The semantic grep tool [10] can be used to define and retrieve patterns like the CIPs that enable Lasso. While we chose to use a parsing and analysis library for Java (JavaParser [38]) to implement the detectors for the AST-CIP component of Lasso-13, any tool that can return a list of ESCs could be used instead.

The work by Blasco et al. [6] uses LSI and genetic algorithms to retrieve traces at lower granularity than methods for a natural language requirement, in a specific commercial video game. This approach is fundamentally different from Lasso-13, as it is meant to trace requirements that are larger than data constraints, and that are assumed to be implemented in multiple code locations, each. In contrast our technique assumes that each data constraint will be implemented in a discrete location inside a method, as observed by previous work [27]. Additionally, their technique randomly selects seed lines for the genetic algorithm based, and the results are non-deterministic. Our approach both uses specific line-of-code patterns to build the ESCs and is deterministic.

A related line of research uses static analysis to reverse engineer business rules, usually from a legacy system where only the binary code is available [14, 20, 21, 33, 37, 61, 62, 65]. These approaches differ from Lasso in that they do not use the text of the business rule as input, instead rely on the developer selecting relevant input/output variables. From this set of variables they perform forwards or backward slicing, and find the branches in the slice, since conditional branches are the locations where business rules are checked.

Finally, recent research by Yang *et al.* [69] explored the implementation of data constraints in database-backed web applications. They discovered that developers struggle with maintaining consistent data constraints and with checking them across different components and versions of their web applications. This work is one of the main motivation behind our research.

8 CONCLUSIONS

Lasso is a novel traceability link recovery technique, designed as framework, which uses fine-grained code patterns to enable the retrieval of links with line-of-code granularity. Three concrete versions of this framework, Lasso-13Luc, Lasso-13VSM, and Lasso-13LSI were shown to achieve a %HITS@10 of 66.9%, 60.2%, and 51.8%, respectively, while outperforming their corresponding baselines by 30%, 70%, and 163%, respectively. Additionally, all three Lasso-13 variations can return the correct line of code implementation within the first 3 ranks in the corresponding method for 79% of constraints.

These findings show that empirical knowledge of the space of constraint implementations and common-sense heuristics can enable effective retrieval at line-of-code granularity, while improving the performance of method-level approaches.

Lasso is extensible with new types of constraint implementation pattern matchers and, with enough such extensions, Lasso could be used to trace any type of data constraints implemented with any current or future implementation patterns. Since Lasso operates at line-of-code level, we envision tools that will automatically identify and change the enforcing statement when the underlying rules change.

Researchers identified trace accuracy and trace granularity as two remaining grand challenges in traceability [3, 18, 44]. Future requirements-to-code traceability link recovery approaches can bootstrap on Lasso, which will find the lines of code implementing the constraints embedded in the requirements, and use these "seeds" to improve the accuracy of tracing the larger requirements. This will not only improve the accuracy of coarse-grained traceability, but will also provide fine-grained links that can be directly used when they are needed.

ACKNOWLEDGMENTS

This research was supported in part by grants for the US National Science Foundation: CCF-1955837, CCF-1910976.

REFERENCES

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering* 28, 10 (Oct. 2002), 970–983. https://doi.org/10.1109/TSE.2002.1041053

[2] Giuliano Antoniol, Gerardo Canfora, Andrea De Lucia, and Ettore Merlo. 1999. Recovering Code to Documentation Links in OO Systems. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*. 136–144. https://doi.org/10.1109/WCRE.1999.806954

[3] Giuliano Antoniol, Jane Cleland-Huang, Jane Huffman Hayes, and Michael Vierhauser. 2017. Grand Challenges of Traceability: The Next Ten Years. *arXiv:1710.03129 [cs]* (Oct. 2017). arXiv:1710.03129 [cs]

[4] Apache Ant. 2021. Targets. https://archive.apache.org/dist/ant/manual/apache-ant-1.10.6-manual.zip.

[5] Nicolas Bettenburg, Bram Adams, Ahmed E. Hassan, and Michel Smidt. 2011. A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC)*. 185–188. https://doi.org/10.1109/ICPC.2011.36

[6] Daniel Blasco, Carlos Cetina, and Óscar Pastor. 2020. A Fine-Grained Requirement Traceability Evolutionary Algorithm: Kromaia, a Commercial Video Game Case Study. *Information and Software Technology* 119 (March 2020), 106235. https://doi.org/10.1016/j.infsof.2019.106235

[7] Board of Governors of the Federal Reserve System. 2020. Federal Reserve Board Announces Interim Final Rule to Delete the Six-per-Month Limit on Convenient Transfers from the "Savings Deposit" Definition in Regulation D. https://www.federalreserve.gov/newsevents/pressreleases/bcreg20200424a.htm.

[8] Board of Governors of the Federal Reserve System. 2020. Regulation D Reserve Requirements. https://www.federalreserve.gov/boarddocs/supmanual/cch/int_depos.pdf.

[9] Markus Borg, Per Runeson, and Anders Ardö. 2014. Recovering from a Decade: A Systematic Mapping of Information Retrieval Approaches to Software Traceability. *Empirical Software Engineering* 19, 6 (2014), 1565–1616. https://doi.org/10.1007/s10664-013-9255-y

[10] R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey. 2002. Semantic Grep: Regular Expressions + Relational Abstraction. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*. 267–276. https://doi.org/10.1109/WCRE.2002.1173084

[11] Gerardo Canfora and Luigi Cerulo. 2006. Fine Grained Indexing of Software Repositories to Support Impact Analysis. In *Proceedings of the 3rd Mining Software Repositories Conference (MSR) (MSR '06)*. Association for Computing Machinery, New York, NY, USA, 105–111. https://doi.org/10.1145/1137983.1138009

[12] Karel Cemus, Tomas Cerny, and Michael J. Donahoo. 2015. Evaluation of Approaches to Business Rules Maintenance in Enterprise Information Systems. In *Proceedings of the 2015 International Conference on Research in Adaptive and Convergent Systems (RACS) (RACS)*. Association for Computing Machinery, New York, NY, USA, 324–329. https://doi.org/10.1145/2811411.2811476

[13] Thomas Cerny and Michael J. Donahoo. 2011. How to Reduce Costs of Business Logic Maintenance. In *Proceedings of the 6th IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, Vol. 1. 77–82. https://doi.org/10.1109/CSAE.2011.5953174

[14] Oscar Chaparro, Jairo Aponte, Fernando Ortega, and Andrian Marcus. 2012. Towards the Automatic Extraction of Structural Business Rules from Legacy Databases. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*. 479–488. https://doi.org/10.1109/WCRE.2012.57

[15] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2017. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME)*. 376–387. https://doi.org/10.1109/ICSME.2017.100

[16] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2019. Using Bug Descriptions to Reformulate Queries during Text-Retrieval-Based Bug Localization. *Empirical Software Engineering* 24, 5 (Oct. 2019), 2947–3007. https://doi.org/10.1007/s10664-018-9672-z

[17] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. 2019. Reformulating Queries for Duplicate Bug Report Detection. In *Proceedings of the 26th International Conferene on Software Analysis, Evolution and Reengineering (SANER)*. 218–229. https://doi.org/10.1109/SANER.2019.8667985

[18] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software Traceability: Trends and Future Directions. In *Proceedings of the Future of Software Engineering (FOSE) of the 36th International Conference on Software Engineering (ICSE) (FOSE 2014)*. ACM, New York, NY, USA, 55–69. https://doi.org/10.1145/2593882.2593891

[19] Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. 2005. Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability. In *Proceedings of the 13th IEEE International Requirements Engineering Conference (RE)*. 135–144. https://doi.org/10.1109/RE.2005.78

[20] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. 2012. A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. In *Rules on the Web: Research and Applications (Lecture Notes in Computer Science)*, Antonis Bikakis and Adrian Giurca (Eds.). Springer, Berlin, Heidelberg, 17–31.

[21] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. 2013. Extracting Business Rules from COBOL: A Model-Based Framework. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. 409–416. https://doi.org/10.1109/WCRE.2013.6671316

[22] Scott Deerwester, Susan Dumais, G. W. Furnas, Thomas Landauer, and R. Harshman. 1990. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science* 41 (1990), 391–407.

[23] Diana Diaz, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Silvia Takahashi, and Andrea De Lucia. 2013. Using Code Ownership to Improve IR-based Traceability Link Recovery. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*. 123–132. https://doi.org/10.1109/ICPC.2013.6613840

[24] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*. 53–62. https://doi.org/10.1109/ICPC.2008.39

[25] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2009. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Proceedings of the 6th Mining Software Repositories Conference (MSR)*. 71–80. https://doi.org/10.1109/MSR.2009.5069482

[26] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. 2021. Combining Query Reduction and Expansion for Text-Retrieval-Based Bug Localization. In *Proceedings of the 28th International Conferene on Software Analysis, Evolution and Reengineering (SANER)*. 166–176. https://doi.org/10.1109/SANER50967.2021.00024

[27] Juan Manuel Florez, Laura Moreno, Zenong Zhang, Shiyi Wei, and Andrian Marcus. 2021. An Empirical Study of Data Constraint Implementations in Java. *arXiv:2107.04720 [cs]* (July 2021). arXiv:2107.04720 [cs]

[28] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. 2022. Retrieving Data Constraint Implementations Using Fine-Grained Code Patterns (Replication Package). https://doi.org/10.5281/zenodo.5915650.

[29] Malcom Gethers, Huzefa Kagdi, Bogdan Dit, and Denys Poshyvanyk. 2011. An Adaptive Approach to Impact Analysis from Change Requests to Source Code. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 540–543. https://doi.org/10.1109/ASE.2011.6100120

[30] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. TIDIER: An Identifier Splitting Approach Using Speech Recognition Techniques. *Journal of Software: Evolution and Process* 25, 6 (2013), 575–599. https://doi.org/10.1002/smr.539

[31] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. 3–14. https://doi.org/10.1109/ICSE.2017.9

[32] Donna Harman. 1993. Overview of the First TREC Conference. In *Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '93)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/160688.160692

[33] Tomomi Hatano, Takashi Ishio, Joji Okada, Yuji Sakata, and Katsuro Inoue. 2016. Dependency-Based Extraction of Conditional Statements for Understanding Business Rules. *IEICE Transactions on Information and Systems* E99.D, 4 (2016), 1117–1126. https://doi.org/10.1587/transinf.2015EDP7202

[34] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Transactions on Software Engineering* 32, 1 (Jan. 2006), 4–19. https://doi.org/10.1109/TSE.2006.3

[35] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, E. Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. 2007. REquirements TRacing On Target (RETRO): Improving Software Maintenance through Traceability Recovery. *Innovations in Systems and Software Engineering* 3, 3 (Sept. 2007), 193–202. https://doi.org/10.1007/s11334-007-0024-1

[36] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. 2008. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. In *Proceedings of the 5th Mining Software Repositories Conference (MSR) (MSR '08)*. Association for Computing Machinery, New York, NY, USA, 79–88. https://doi.org/10.1145/1370750.1370771

[37] Hai Huang, Wei-Tek Tsai, Sourav Bhattacharya, Xiaoping Chen, Yamin Wang, and Jianhua Sun. 1996. Business Rule Extraction from Legacy Code. In *Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC)*. 162–167. https://doi.org/10.1109/CMPSAC.1996.544158

[38] JavaParser. 2021. JavaParser. https://javaparser.org/.

[39] Joda-Time. 2021. GregorianJulian (GJ) Calendar System. https://www.joda.org/joda-time/cal_gj.html.

[40] Hongyu Kuang, Jia Nie, Hao Hu, Patrick Rempel, Jian Lü, Alexander Egyed, and Patrick Mäder. 2017. Analyzing Closeness of Code Dependencies for Improving IR-based Traceability Recovery. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 68–78. https://doi.org/10.1109/SANER.2017.7884610

[41] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Extracting Meaning from Abbreviated Identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 213–222. https://doi.org/10.1109/SCAM.2007.17

[42] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 324–335. https://doi.org/10.1109/ICSE43902.2021.00040

[43] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2007. Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods. *ACM Trans. Softw. Eng. Methodol.* 16, 4 (Sept. 2007). https://doi.org/10.1145/1276933.1276934

[44] Patrick Mäder, Paul L. Jones, Yi Zhang, and Jane Cleland-Huang. 2013. Strategic Traceability for Safety-Critical Projects. *IEEE Software* 30, 3 (May 2013), 58–66. https://doi.org/10.1109/MS.2013.60

[45] Anas Mahmoud. 2015. An Information Theoretic Approach for Extracting and Tracing Non-Functional Requirements. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference (RE)*. 36–45. https://doi.org/10.1109/RE.2015.7320406

[46] Anas Mahmoud and Grant Williams. 2016. Detecting, Classifying, and Tracing Non-Functional Software Requirements. *Requirements Engineering* 21, 3 (Sept. 2016), 357–381. https://doi.org/10.1007/s00766-016-0252-8

[47] Andrian Marcus, Jonathan I. Maletic, and Andrey Sergeyev. 2005. Recovery of Traceability Links Between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering* 15, 05 (Oct. 2005), 811–836. https://doi.org/10.1142/S0218194005002543

[48] Michael McCandless, Erik Hatcher, and Otis Gospodnetić. 2010. *Lucene in Action* (2nd ed ed.). Manning, Greenwich.

[49] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. 2009. Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery. In *Proceedings of the 5th ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. IEEE Computer Society, Washington, DC, USA, 41–48. https://doi.org/10.1109/TEFSE.2009.5069582

[50] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. 2020. On the Relationship between Bug Reports and Queries for Text Retrieval-Based Bug Localization. *Empirical Software Engineering* 25, 5 (Sept. 2020), 3086–3127. https://doi.org/10.1007/s10664-020-09823-w

[51] Mehdi Mirakhorli and Jane Cleland-Huang. 2016. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Transactions on Software Engineering* 42, 3 (March 2016), 205–220. https://doi.org/10.1109/TSE.2015.2479217

[52] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*. 151–160. https://doi.org/10.1109/ICSME.2014.37

[53] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2010. On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC)*. 68–71. https://doi.org/10.1109/ICPC.2010.20

[54] Martin F. Porter. 1980. An Algorithm for Suffix Stripping. *Program: electronic library and information systems* 14, 3 (1980), 130–137. https://doi.org/10.1108/eb046814

[55] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-based Bug Localization with Context-aware Query Reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 621–632. https://doi.org/10.1145/3236024.3236065

[56] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE) (ICSE '18)*. ACM, New York, NY, USA, 834–845. https://doi.org/10.1145/3180155.3180207

[57] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 Extension to Multiple Weighted Fields. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM '04)*. Association for Computing Machinery, New York, NY, USA, 42–49. https://doi.org/10.1145/1031171.1031181

[58] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the 29th IEEE/ACM International Conference on Software Engineering (ICSE)*. 499–510. https://doi.org/10.1109/ICSE.2007.32

[59] Giuseppe Scanniello and Andrian Marcus. 2011. Clustering Support for Static Concept Location in Source Code. In *19th IEEE International Conference on Program Comprehension (ICPC'11)*. 1–10. https://doi.org/10.1109/icpc.2011.13

[60] Giuseppe Scanniello, Andrian Marcus, and Daniele Pascale. 2015. Link Analysis Algorithms for Static Concept Location: An Empirical Assessment. *Empirical Software Engineering* 20, 6 (Dec. 2015), 1666–1720. https://doi.org/10.1007/s10664-014-9327-7

[61] Harry M. Sneed. 2001. Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment. In *Proceedings of the 9th IEEE Workshop on Program Comprehension (IWPC)*. 167–175. https://doi.org/10.1109/WPC.2001.921728

[62] Harry M. Sneed and Katalin Erdös. 1996. Extracting Business Rules from Source Code. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC)*. Berlin, Germany, 240–247. https://doi.org/10.1109/WPC.1996.501138

[63] Hakim Sultanov, Jane Huffman Hayes, and Wei-Keat Kong. 2011. Application of Swarm Techniques to Requirements Tracing. *Requirements Engineering* 16, 3 (Sept. 2011), 209–226. https://doi.org/10.1007/s00766-011-0121-4

[64] Swarm. 2021. Seismic Wave Analysis and Real-Time Monitor: User Manual and Reference Guide. Version 2.8.10. https://github.com/usgs/swarm/blob/97f8b2f26830c764b816ca0a74270d5c0db35d06/docs/swarm_v2.pdf.

[65] Xinyu Wang, Jianling Sun, Xiaohu Yang, Zhijun He, and Srini Maddineni. 2004. Business Rules Extraction from Large Legacy Systems. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR)*. 249–258. https://doi.org/10.1109/CSMR.2004.1281426

[66] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 262–273.

[67] Karl E. Wiegers and Joy Beatty. 2013. *Software Requirements* (third ed.). Microsoft Press, Redmond, WA.

[68] Graham C. Witt. 2012. *Writing Effective Business Rules : A Practical Method*. Morgan Kaufmann, Waltham, MA.

[69] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing Data Constraints in Database-Backed Web Applications. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1098–1109. https://doi.org/10.1145/3377811.3380375

[70] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE)*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210