DCLINK: Bridging Data Constraint Changes and Implementations in FinTech Systems

Wensheng Tang*[†], Chengpeng Wang*[†], Peisen Yao[‡], Rongxin Wu[§], Xianjin Fu[¶], Gang Fan[¶], and Charles Zhang*
*The Hong Kong University of Science and Technology, Hong Kong, China
{wtangae, cwangch, charlesz}@cse.ust.hk
[‡]Zhejiang University, China pyaoaa@zju.edu.cn
[§]Xiamen University, China wurongxin@xmu.edu.cn
[¶]Ant Group, China {fuxianjin.fxj, fangang}@antgroup.com

***Abstract*—A FinTech system is a cluster of FinTech applications that intensively interact with databases containing a large quantity of user data. To ensure data consistency, it is a common practice to specify data constraints to validate data at runtime. However, data constraints often evolve according to changes in business requirements. Meanwhile, the developers can hardly keep up with the latest requirements during the development cycle. Such an information barrier increases the communication burden and prevents FinTech applications from being updated in time, impeding the development cycle significantly.**

In this paper, we present a comprehensive empirical study on data constraints in FinTech systems, investigating how they evolve and affect the development process. Our results show that developers find it hard to update their code timely because no mapping from data constraint changes to code is provided. Inspired by the findings from code updates respecting data constraint changes, we propose DCLINK, a traceability link analysis for linking each data constraint change to target methods demanding the code update in the FinTech application. We extensively evaluate DCLINK upon real-world change cases in Ant Group. The results show that DCLINK can effectively and efficiently localize the target methods.

***Index Terms*—static analysis, impact analysis, data constraint, FinTech application**

I. INTRODUCTION

FinTech systems, e.g., e-payment services Paypal and Stripe, serve on a daily basis for ordinary people. These systems have also been increasingly prevalent in the software industry. Typically, a FinTech system comprises multiple microservices and cloud databases. The core business logic relies on frequent interactions that involve reading and updating database records. Since these records contain a lot of sensitive data, e.g., user balance, ensuring data consistency is one of the most critical concerns in developing and maintaining FinTech systems. For example, Amazon store processed 288 billion transactions using 5,326 database instances of Amazon Aurora, stored 1,849 terabytes of data, and transferred 749 terabytes of data on Prime Day 2022 [1]. Any erroneous data value can affect the system's robustness and even introduce enormous economic loss [2].

[†]The first two authors contributed equally to this work and are sorted in alphabetical order.

To ensure data consistency, FinTech system managers specify target properties of data values in the database tables as *data constraints*, which constrain the data value in the databases at runtime. Specifically, to create these constraints, the managers summarize business requirements, list target properties, and write data constraints in a domain-specific language. When a data constraint is violated, error alarms notify the managers and developers, guiding them to take immediate reactions.

Due to constant changes in business requirements, the managers have to update the corresponding data constraints, and meanwhile, the developers are expected to refine their implementations to fit the changed requirements. Delayed updates would impede the development cycle and even postpone the testing and deployment to the brink. Unfortunately, there are no systematic studies on understanding the difficulties and challenges in bridging data constraints and implementations and no mature tools to assist developers. Thus, in this work, we conducted a large-scale study of 5,906 data constraints of a FinTech system and their relevant developers in Ant Group, a global FinTech company. The study aims to explore the following three research questions.

- **RQ1:** Is keeping up with data constraint changes for code updates difficult?
- **RQ2:** How does a data constraint change in an evolving FinTech system?
- **RQ3:** How does the application code change when a data constraint changes?

Through developer interviews and analysis of different versions of data constraints and application code, we have uncovered several significant implications. First, we find that most (73.9%) of the interviewees admitted that they had encountered difficulties when making corresponding implementation updates, of which the majority (76.5%) thought mapping data constraint changes to code is the key obstacle. Second, data constraints are often modified by adding or deleting a clause, which takes up 52.02% of the total change cases. Third, there is a particular class of field variables in the application, namely *anchored field variables*, which indicates the correspondence between the data constraint change and code updates effectively.

Based on the findings, we realize that it is essential and feasible to bridge data constraint changes and implementations. While there have been several studies in analyzing data constraints [3], [4] in other domains, their approaches are not applicable to our problem. Several studies focus on building the links between the data constraints and the bug reports [4]. The other takes natural language descriptions of data constraints as the inputs for the mapping [3]. Adapting these approaches to the data constraints described in a domain-specific language (DSL) requires substantial manual efforts in writing descriptions. Test-to-code traceability studies are also relevant to our work. They leverage naming similarities, like naming conventions [5] or string distance [6], to build linkage to the implementations [7]. However, none of them consider the impacts of the data constraint changes on building the links to the code implementation, thus causing missing leakage.

Inspired by the empirical findings, we propose DCLINK, a traceability link analysis that automatically infers the code implementations demanding modifications according to data constraint changes. Our technique targets inferring the method level implementation, since methods are commonly used in unit testing and helpful for debugging [8]–[10]. To overcome the limitation of existing studies, DCLINK is aware of the change patterns of data constraints and localizes the target methods following the change patterns accordingly. First, we determine the program field variables related to the data constraint change as the anchored field variables. Second, we collect the impacted methods whose statements use the anchored field variables. Third, we aggregate all the methods and prioritize them based on a heuristic policy, returning an ordered list of methods to developers for code updates.

We evaluate DCLINK using 75 cases of data constraint changes among five crucial and most actively-developed FinTech applications in Ant Group. By comparing the methods in the returned list with the ground-truth target method labeled by developers, we determine if each target method is included in the Top K functions for each case. The experimental results show that the *Hit@10* and *Hit@20* values of DCLINK are 76.00% and 93.33%, respectively. Besides, DCLINK finished analyzing each case efficiently, taking only 30.35 seconds on average, which demonstrates its practical use in real industrial scenarios. Overall, our work makes three major contributions:

- We conduct the first large-scale study of data constraints in FinTech systems, which demonstrates the necessity of bridging data constraint changes and implementations.
- We introduce the traceability link analysis DCLINK for localizing the target methods that demand modifications based on data constraint changes, providing suggestions for the developers in FinTech application maintenance.
- We evaluate DCLINK on real-world FinTech applications and demonstrate its effectiveness and efficiency in development assistance. DCLINK has been deployed in the production line of Ant Group.

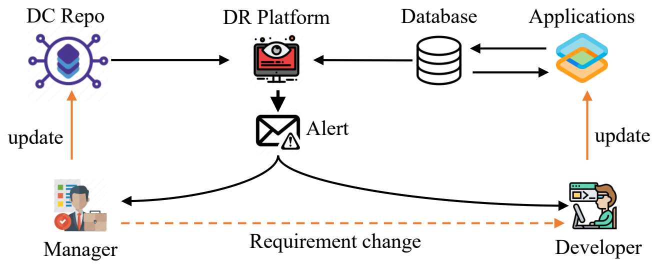


Fig. 1: The development cycle of a FinTech system

II. BACKGROUND

In this section, we first present the background of FinTech systems, and then introduce data constraints and their changes.

A. FinTech System Development

To enforce data consistency of FinTech systems, companies typically build a data reconciliation (DR) platform to examine the data properties dynamically. Fig. 1 shows the development cycle of a FinTech system in Ant Group.

- A manager specifies data consistency rules as *data constraints* in a DSL on the DR platform.
- The DR platform evaluates the data constraints on the database tables and then checks whether the records satisfy the desired properties.
- When a data constraint is violated, an alert will be sent to both managers and developers for further diagnosis.

In FinTech systems, service logic is frequently changed upon new business requirements, where both managers and developers need to update the data constraints and application code, respectively. Typically, to enforce data consistency in the persistent storage layer, the data constraints are updated ahead of the application code update, which should also address the changes to data consistency rules.

B. Data Constraints in FinTech Systems

There have been an increasing number of studies on data constraints [4], [11], [12]. For example, access control lists serve as a special data constraint describing network traffic rules [13], while they are not bounded to certain applications. Yang et al. [4] and Florez et al. [11] comprehensively studied data constraints in database-backed applications. To the best of our knowledge, the data constraints in FinTech systems have the most complex semantics studied so far, where

- A statement is an assertion statement, an assignment statement, or an *if*-statement.
- An assertion contains a *Boolean* expression of one or multiple clauses connected with logic connectives.
- A variable is either a column variable or a temporary variable, which loads the attribute value of a record from a table or stores the intermediate results, respectively.
- An operator can be a comparison operator used in a clause or an arithmetic operator. We also regard the string predicates, such as *contains* and *equals*, as generalized operators.

```
    assert(user.id != nil);
    if (contains(user.type, "MEMBER"))
       discount = user.discount;
R   else
       discount = 1.0;
    pay = discount * user.cost;
  - assert(user.balance - user.new_balance == pay);
    assert(user.id != nil);
    if (contains(user.type, "MEMBER"))
       discount = user.discount;
    else
       discount = 1.0;
R'  pay = discount * user.cost;
  + if (!equals(trans.type, "PROMOTION"))
  +    assert(user.balance - user.new_balance == pay);
  + else
  +    assert(user.balance - user.new_balance == pay * discount);
```

Fig. 2: The old and new data constraints checking the balances

In addition to the form differences, due to the frequent requirement changes in FinTech systems, data constraints must be updated accordingly and timely. Under such criteria, to build links between the data constraint changes and code demanding modification, existing similarity-based traceability approaches [5], [7] are limited to robustly finding the linkage, since the propagation of changes is discarded.

Example 1. The data constraint R in Fig. 2 contains two assertions. Both assertions have the same clause in Line 1, examining the nullability of the user id and the consistency of the account balances and the payment, respectively. The temporary variable *discount* stores the discounts for different types of users. Notably, R utilizes the string predicate *contains* to determine if the column variable *user.type* takes the literal *'MEMBER'* as its substring. Also, R leverages arithmetic operators $(-)$ to express the expected relation of account balances and payment. The new data constraint R' specifies promotional data consistency properties in addition to the original one by adding an *if*-statement and a new assertion. At this point, the developers must update their code to reflect the corresponding data constraint changes.

III. Empirical Study Methodology

Although making code updates keep up with data constraint changes is a critical task, there are no mature tools to assist this due to the lack of systematic studies in understanding the challenges and characteristics of such a task. To bridge this gap, in this section, we present a systematic empirical study based on the dataset collected from real-world applications.

A. Subject Collection

To answer the research questions (RQ1-RQ3), we collect the FinTech applications developed by a technical unit in Ant Group. From an internal dashboard, we choose the Top 5 most frequently-updated applications and their developers as our study subjects[1]. These applications have been developed by an average of 134 developers so far, and their sizes range from 147 kLoC to 269 kLoC. On average, each application is updated with 4 commits per day. In total, there are 5,905 data constraints specified for the five applications, and each of them is updated an average of 2.28 times. Overall, our study subjects exhibit the frequently-evolving feature, serving as typical subjects for studying data constraint changes and code updates.

[1]Due to confidential agreement, we are permitted to access the source code of no more than five applications.

B. Analysis of Study Subjects

We propose three analyses with the collected subjects, which are the user experience analysis, the change analysis, and the impact analysis. In what follows, we present the details of the analyses.

User Experience Analysis. To answer RQ1, we first design the questionnaire to understand whether developers have difficulties in updating code subject to data constraint changes, and if yes, what is the root cause making such a process difficult. We designed the questionnaire with the questions in Fig. 3a. To ensure that the questionnaire is handed out to the most relevant core developers of the five most actively-developed applications, we utilize Git histories to select the developers that have changed over one thousand lines of code over the last two months. In total, we hand out questionnaires to 23 relevant developers satisfying these criteria. Through conducting a thorough user experience analysis, we can gain a profound understanding of the problem challenges.

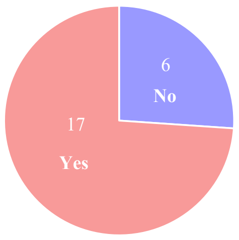
Change Analysis. To answer RQ2, we compare the data constraints before and after each modification. To describe the change of data constraints, we borrow the concepts of *edit action* and *edit script* in [14]. Specifically, an edit action can be a node addition, deletion, and replacement upon the abstract syntax tree (AST) of the data constraint, while an edit script is essentially a sequence of the edit actions. Technically, we leverage the AST differencing algorithm [14] to generate the edit script for each pair of old and new data constraints. Lastly, we classify the edit actions based on the AST node types as follows, which further forms several patterns of edit actions.

- *Statement*: We determine whether the edit action is an insertion or deletion of a statement. We do not consider the replacement of a statement because it can be represented by the composition of an insertion and a deletion.
- *Expression*: To have a fine-grained classification of the node type, we examine the parents and children of the node representing an expression. For example, if the newly-added expression is the operand of a logical connective, the edit action is actually the clause insertion for the data constraint.
- *Variable or operator*: When the node represents a variable or an operator, the edit action can be the replacement of a variable or an operator, respectively. We do not consider the insertion and deletion of variables and operators, as they are subsumed by the changes in expressions.

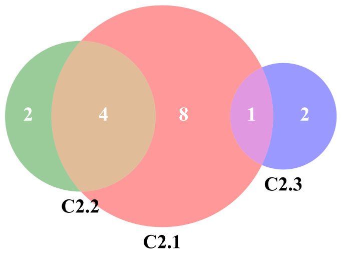
Using the change analysis, we process the change history of each data constraint and obtain the edit scripts of continuous

No.	Questions	
Q1	Have you met difficulties in updating code subject to data constraint changes?	Yes No ☐ ☐
Q2	(Multi) If yes, what makes this process hard?	
	C2.1 Map data constraint changes to code	☐
	C2.2 Confirm with managers with requirements	☐
	C2.3 Other not listed difficulties	☐

(a) User experience questionnaire



(b) Results of Q1



(c) Results of Q2

Fig. 3: The results of the user experience analysis

versions. By summarizing the patterns of edit actions, we can derive the typical ways in which data constraints evolve.

Impact Analysis. To answer RQ3, we utilize intermediate results of the user experience analysis and the change analysis to study the impact of data constraint changes on implementations. Our impact analysis consists of three parts as follows:

- First, we identify the code commits driven by data constraint changes and the application code diff. We then ask the developers of the commits to pick out all the related implementation changes from the diff information.
- Second, we perform the change analysis upon the corresponding data constraint pair for each commit and extract the edit script to understand the data constraint change.
- Third, we manually inspect its edit script and the modified implementations and summarize how data constraint changes affect the implementation updates.

C. Summary

The aforementioned empirical study setups enable us to investigate the necessity of bridging data constraint changes with implementations, and understand how data constraints evolve and affect application code updates in the wild. Particularly, our study involves various kinds of subjects, such as the feedback of questionnaires, git history, data constraints, etc. The overall analysis of our study aggregates the multi-domain knowledge to demystify the data constraint changes and their impacts on the evolving FinTech applications.

IV. Empirical Observations

A. RQ1: Is keeping up with data constraint changes for code updates difficult?

To answer RQ1, we perform a user experience analysis to understand developers' issues with the code update. Specifically, we collect the developers' feedback on the questionnaire. Fig. 3b and 3c show the results of the user experience analysis. According to the feedback, most developers (17/23=73.9%) encountered difficulties when keeping up with data constraint changes for code updates. Among the developers with difficulties, most developers (13/17=76.5%) thought mapping data constraint logic to their code changes was the main obstacle to keeping up with the changes. There are also some (6/17=35.3%) of developers who admit that they have to frequently communicate with at least one manager when a data constraint has been changed, so that they can make the required changes in the proper methods. Three developers argue that there are other difficulties in understanding complex data constraints. "The data constraints refer to multiple database tables, making it difficult to understand the business logic", said one of the three developers. According to the above comments, we find that the mapping data constraint changes to the application code, which demands an update, is the key obstacle. To understand the underlying reasons deeply, we also conducted interviews with two developers who have mapping difficulties. Both said the reasons are that there is no direct mapping from data constraints to application code, so it is laborious to examine the verbose list of methods that are possible to be updated. Finally, we can safely draw a conclusion to RQ1.

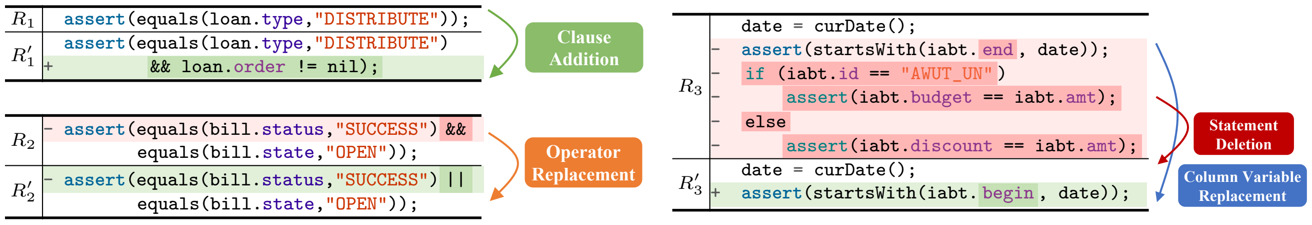
Finding 1: Developers have difficulty in keeping up with data constraint changes mainly due to the lack of linkage between data constraints and related application code.

B. RQ2: How does a data constraint change in an evolving FinTech system?

To answer RQ2, we conduct the change analysis for each modification of data constraints, and summarize six kinds of edit actions occurring in the edit scripts. Fig. 4 demonstrates three pairs of original and modified data constraints as examples, while Fig. 5 shows the proportions of the six kinds of edit actions. In what follows, we demonstrate each kind of edit action with the examples shown in Fig. 4.

Clause Addition/Deletion. The most common kind of edit action is clause addition, which takes up 33.79% of the total changes. The rules R_1 and R_1' in Fig. 4 show an example of the clause addition. Connected with logical conjunctions, clauses in the assertion pose a stronger restriction on the database table *loan*. Similarly, managers can delete a clause from an assertion, and the clause deletion takes up 18.23% of the investigated edit actions.

Statement Addition/Deletion. The addition and deletion of statements also account for relatively large proportions (15.90% and 9.53%) of data constraint changes. For example, R_3' in Fig. 4 contains a new if-statement, posing an additional



```
R1    assert(equals(loan.type,"DISTRIBUTE"));
R'1   assert(equals(loan.type,"DISTRIBUTE")
    +        && loan.order != nil);

R2  - assert(equals(bill.status,"SUCCESS") &&
             equals(bill.state,"OPEN"));
R'2 - assert(equals(bill.status,"SUCCESS") ||
             equals(bill.state,"OPEN"));

      date = curDate();
R3  - assert(startsWith(iabt.end, date));
    - if (iabt.id == "AWUT_UN")
    -      assert(iabt.budget == iabt.amt);
    - else
    -      assert(iabt.discount == iabt.amt);
R'3   date = curDate();
    + assert(startsWith(iabt.begin, date));
```

Fig. 4: The examples of edit actions

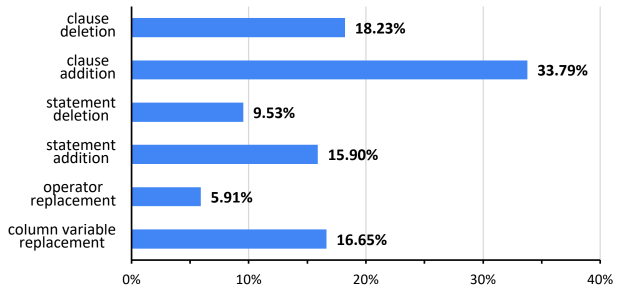


Fig. 5: The proportions of different edit actions

constraint upon the table *iabt*. In most cases, the insertion and removal of statements can introduce and eliminate assertions, respectively, which have the same effect as the insertion and removal of clauses.

Column Variable Replacement. We find that 16.65% of changes involve the column variable replacement. For example, the new data constraint R'_3 in Fig. 4 examines the values in the column *begin* rather than *end* of the table *iabt*. Such kind of changes are often caused by the refined design of the system, The new data constraint describes a totally different property from the original one. Besides, the database schema refactoring can also introduce column variable replacement, as the managers have to update the column variables according to the latest schema.

Operator Replacement. We also notice that several modifications are quite minor, involving the replacement of operators. In Fig. 4, as an example, the logical conjunction used in R_2 is replaced with a logical disjunction in R'_2. Similar cases include the replacement of comparison operators and string predicates.

We also quantify the number of edit actions in each edit script. For most data constraints, the changes are fairly minor, only involving one or two edit actions, which take up 47.67% and 21.89%, respectively. For example, the first two pairs in Fig. 4 only contain a single edit action, while the third pair has two edit actions. In real-world scenarios, managers are more likely to adjust a data constraint with a minor revision, as business requirements often evolve smoothly.

Finding 2: The data constraint changes can involve the addition or deletion of a clause or statement, and the replacement of a column variable or an operator.

C. RQ3: How does the application code change when a data constraint changes?

To answer RQ3, we contacted the developers to manually collect 116 commits representing the code changes subject to date constraint updates. Then, we do an impact analysis to investigate how data constraint changes affect implementation updates. Now we demonstrate two critical findings with the examples in Fig. 6, which correspond to the data constraint changes in Fig. 4.

First, we identify several characteristics of the correlations between data constraints and application code.

- A column variable in a data constraint corresponds to a field variable of a class. The application code manipulates the records in the database tables via field variables.
- A particular class of column variables is a good indicator of the correspondence between the data constraint change and the implementation. When column variables appear in a specific scope of the data constraint, the corresponding field variables are also used in the target methods.

For clarity, we name such column variables and field variables as the *anchored column variables* and the *anchored field variables*. When bridging data constraint changes and target methods, the crux is to identify anchored column and field variables. To resolve the issue, we summarize our observations on their features in Table I. In short, the occurrence of anchored column variables depends on the edit action type.

- When adding a new clause or statement, the anchored column variables are located in the original Boolean expression or pre-existing statements of the old data constraint, as the developers need to modify the methods using the corresponding field variables.
- For the deletion of a clause or a statement, the column variables in the deleted construct are regarded as anchored ones. The developers should update specific methods using the corresponding field variables.
- For the other two edit actions, the anchored column variables are exactly the changed one or the operands of the changed operator, respectively.

Example 2. For the clause addition shown in Fig. 4, the anchored column variable is the column *type* of the table *loan*, inducing the anchored field variable *type* of the class *Loan* in the application code, which is used in the target method *buildLoan*. Similarly, we can obtain the anchored column variables and field variables for other types of edit actions, which are shown in the last five rows of Table I.

```
public class Loan {
    @Getter @Setter private String type;
    @Getter @Setter private String order;
}
public class LoanDO {
    public void buildLoan() {
        String type = loan.getType();
+       String order = loan.getOrder();
-       if (type == "DISTRIBUTE") {
+               && order != null) {
            LoanDO.insert(type); // I/O
            ...
        }
    }
    public void confirmLoan() {
        String type = getType();
        String order = getOrder();
        ...
        ConfirmDAO.update(type, order);
    }
}
```

(a)

```
public class Bill {
    @Getter @Setter private Status status;
    @Getter @Setter private State state;
}
public class ConfirmDO {
    @Getter @Setter private Status s1;
    @Getter @Setter private State s2;
    private updateData() {
        this.setS1((Bill) t.getStatus());
        this.setS2((Bill) t.getState());
    }
    public void confirmDO() {
        updateData();
-       if (s1 == "SUCCESS" && s2 != "OPEN") {
+       if (s1 == "SUCCESS" || s2 != "OPEN") {
            ConfirmDAO.insertStatus(s1);//I/O
            ConfirmDAO.insertState(s2); //I/O
            ...
        }
    }
}
```

(b)

```
public class IABT {
    private DateTime begin;
    private DateTime end;
    private String id;
    private Money budget;
    private Money discount;
    private Money amount;
    ...

    public void buildAmount() {
+       if (begin == DateTime.now()) {
-       if (end == DateTime.now()) {
-           if (id == "AWUT_UN") {
-               amount = budget;
-           } else {
-               amount = discount;
-           }
            amountDO.insert(amount);//I/O
        }
    }
}
```

(c)

Fig. 6: The examples of application code changes

TABLE I: The features and examples of anchored column and field variables

Edit Action Type	Scope of Anchored Column Variables	Anchored Column Variables		Anchored Field Variables	
		Table	Column	Class	Field
Clause addition	Original Boolean expression	loan	type	Loan	type
Statement addition	Pre-existing statements	–	–	–	–
Clause deletion	Deleted clauses	–	–	–	–
Statement deletion	Deleted statements	iabt	id, amt, budget, discount	IABT	id, amount, budget, discount
Column variable replacement	Column variables before the change	iabt	end	IABT	end
Operator replacement	Operands of the operator	bill	status, state	Bill	status, state

Finding 3.1: Anchored column variables and field variables indicate the correspondence between the data constraint change and the updated implementation.

To better understand the features of the target methods, we further investigate the usage of anchored field variables in the application code. Specifically, we earn the three typical phenomena of their usage as follows, namely the *value propagation*, the *use aggregation*, and the *IO relevance*, which commonly exist in all the investigated cases.

- **Value propagation:** An anchored field variable is often retrieved by its getter method and modified by its setter method, which can yield the propagation of its value to local variables or the field variables of other classes.
- **Use aggregation:** The target methods tend to utilize the values of all the anchored filed variables simultaneously. The developers can enforce the property specified by a data constraint by checking all the anchored field variables.
- **Database I/O relevance:** The target methods contain the database I/O operations. The developers follow the common practice of checking anchored field variables near the database I/O operations.

Example 3. Fig. 6 demonstrates the above three phenomena.

- In Fig. 6b, the values of the anchored field variables *status* and *state* in the class *Bill* are propagated to the field variables in the class *ConfirmDO* due to the invocation of *updateData*.
- As shown in Fig. 6b and Fig. 6c, all the anchored field variables are utilized in the target methods.
- Lastly, the target methods in all the three examples in Fig. 6 contain the database I/O operations, inserting the records to database tables under specific conditions.

The way of using anchored field variables actually originates from the common programming practice and design. As the database-backed application, the FinTech application can form sophisticated def-use chains, propagating the values of anchored field variables to multiple methods. Also, the developers tend to constrain the relationship of anchored field variables within a method that uses all of them simultaneously, by checking their relationship before updating database tables. Hence, the above three phenomena have quite intuitive explanations, guiding us to automatically localize the target methods according to data constraint changes.

Finding 3.2: The values of anchored field variables are propagated to target methods and utilized simultaneously along with the database I/O operations.

D. Implications

Based on the aforementioned discoveries, it is essential to establish a mechanism that can identify the relevant methods in the application code based on alterations in data constraints.

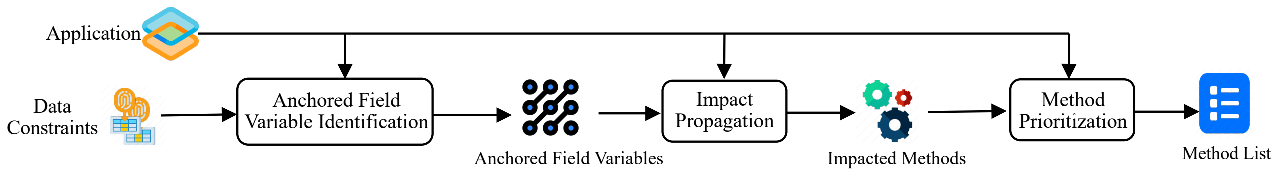


Fig. 7: The workflow of DCLINK

Essentially, it is a problem instance of traceability link analysis [3], [7], [15]–[17] of which an effective solution can assist the developers in updating their implementation on time. Test-to-code traceability techniques [5], [7] are the most relevant to our problem. However, they utilize string similarities or dynamic analysis to recover the linkage, which will neglect the change impacts of source artifacts, i.e., data constraints. Fortunately, our findings can provide key insights into bridging data constraint changes to implementation refinement. Specifically, the target methods often use a particular class of field variables, i.e., anchored field variables, with specific patterns, which inspires us to pinpoint the target methods with anchored field variables. In what follows, we propose a static analysis to bridge the data constraint changes and implementations.

V. TRACEABILITY LINK ANALYSIS

In this section, we present DCLINK, an approach to locating the target methods that may require modifications in response to data constraint changes. According to our empirical study, we observe that the methods using all the anchored field variables are more likely to be the target methods (Finding 3.2). Based on this key idea, we design the workflow of DCLINK, which is shown in Fig. 7. It takes the data constraints of two versions and the old-version application code as input and returns a list of methods. At a high level, DCLINK consists of three stages:

- First, it identifies the anchored field variables in the application code according to the data constraint change.
- Second, it conducts the impact propagation to identify the impacted methods using an anchored field variable.
- Finally, it aggregates all the impacted methods together and prioritizes them with a ranking policy.

In what follows, we present the technical detail of each stage. Throughout this section, we use the data constraints R_1 and R_1' in Fig. 4 as an example to explain how to localize the target function *buildLoan* shown in Fig. 6a.

A. Anchored Field Variable Identification

Identifying anchored field variables requires addressing two issues. First, we need to identify the anchored column variables in the data constraints, which can be derived from the edit actions according to Table I. Second, we must establish the relationship between anchored column variables and field variables, which can be obtained from the application's configuration files. After resolving the two issues, we can identify the anchored field variables, which can be further used to determine the impacted methods.

Algorithm 1: Identifying anchored field variables

```
  Data: Data constraints $a_1$ and $a_2$, configuration file $c$
  Result: $V$: A set of anchored field variables
1 $es \leftarrow$ EditScript$(a_1, a_2)$;
2 $\mathcal{M}_V \leftarrow$ ColToFieldVar$(c)$;
3 $V \leftarrow \varnothing$;  $A \leftarrow \varnothing$;
4 forall $ea \in es$ do
5     $A \leftarrow A \cup$ AnchoredColVar$(a_1, a_2, ea)$ ;
6 forall $(tb, col) \in A$ do
7     $V \leftarrow V \cup \mathcal{M}_V[(tb, col)]$;
8 return $V$;
```

Alg. 1 shows the details of identifying anchored field variables. Initially, the function *EditScript* obtains the edit script es via the change analysis (Line 1). The function *ColToFieldVar* derives the column-field variable mapping $\mathcal{M}_V$ from the configuration file (Line 5). We then process each edit action and extract the anchored column variables based on Table I. Finally, we leverage the mapping $\mathcal{M}_V$ to convert the anchored column variables to the anchored field variables (Lines 6–7).

Example 4. In Fig. 4, the data constraint R_1 is modified by adding a new clause. Based on Table I, we extract the anchored column variable *loan.type* from the clause in the original assertion. We derive the mapping of column variables and field variables from the configuration files from database integration frameworks, e.g., MyBatis [18]. Finally, we identify the anchored field variable *type* in *Loan*.

B. Impact Propagation

To identify the impacted methods of each variable, we can process each variable and extract the methods that utilize its value. Leveraging the datalog-based program analysis [19], [20], we evaluate the def-use relations to collect the methods using the anchored field variables.

Our analysis is formulated as Datalog rules [21] in Fig. 8. The analysis takes three kinds of relations, which indicate the definition of a variable, the def-use relation, and the method containing a statement, respectively. Notably, all the above relations are available in many Datalog-based program analyzers, such as CODEQL [22]. Based on these three relations, we define the analysis rules as follows:

- Given an anchored field variable v, identify all its definitions according to the relation *DefVar*.
- Compute the transitive closure of the relation *DUEdge* to identify all the uses of the value of v.
- Lift the uses to the methods based on *Method* and collect the methods in the relation *UseMethod*.

Input relations	
$\mathit{DefVar}(v, s)$:	A statement s defines the value of a variable v
$\mathit{DUEdge}(s_1, s_2)$:	The value defined by s_1 is used in s_2
$\mathit{Method}(s, m)$:	The statement s appears in method m
Output relations	
$\mathit{UseMethod}(v, m)$:	The method m uses the variable v
Analysis rules	
$\mathit{DUPath}(s_1, s_2)$	:- $\mathit{DUEdge}(s_1, s_2)$
$\mathit{DUPath}(s_1, s_3)$	:- $\mathit{DUPath}(s_1, s_2), \mathit{DUEdge}(s_2, s_3)$.
$\mathit{UseVar}(v, s_2)$	:- $\mathit{DefVar}(v, s_1), \mathit{DUPath}(s_1, s_2)$
$\mathit{UseMethod}(v, m)$	:- $\mathit{UseVar}(v, s), \mathit{Method}(s, m)$

Fig. 8: The rules of the impact propagation

Example 5. As shown in Fig. 6, the anchored field variable *type* is used by its getter method, which is not displayed explicitly. Also, the methods *buildLoan* and *confirmLoan* both invoke the getter method to use the value of *type*. Thus, our analysis rules can finally discover the three impacted methods.

C. Method Prioritization

After the impact propagation, we obtain a list of impacted methods for each anchored field variable. However, a data constraint change may induce multiple anchored field variables, resulting in a lengthy list of impacted methods. Thus, we need to find an effective policy of aggregating the impacted methods to pinpoint the target methods.

It is unveiled in Section IV-C that all anchored variables are often referred to by the same method, and the target method often contains database I/O operations. Inspired by those phenomena, we propose an algorithm for the method prioritization, which narrows down and sorts the methods in a method list. Given a set of anchored field variables V, we conduct the method prioritization after the impact propagation:

- Given the relation $\mathit{UseMethod}(v, m)$ maintaining the impacted methods of each anchored field variable, we compute their common impacted methods as the potential target methods, storing them in the set M' as follows:

$$M' = \bigcap_{v \in V} \{m \mid \mathit{UseMethod}(v, m)\}$$

- For the methods in M', we prioritize them based on the number of the database I/O operations in the method. Specifically, we define the priority function p as follows:

$$p(m) = |\{s \mid \mathit{Method}(s, m) \wedge \mathit{DBIOStmt}(s)\}|$$

  Here, the predicate *DBIOStmt* indicates whether the statement s induces a database I/O operation. Finally, we sort the methods in M' according to their priority function values, yielding the method list M as the result.

Example 6. According to Examples 4 and 5, there is only one anchored field variable which induces three impacted methods. As shown in Fig. 6a, only the method *buildLoan* of the class *LoanDO* contains a database I/O operation. Thus, it is ranked at the top of the list due to its highest priority. In fact, it is exactly the target method modified by the developers.

TABLE II: *Hit@K* and *MAP* values of DCLINK, DCLINK-C, and various conventional techniques

	Hit@1	**Hit@5**	**Hit@10**	**Hit@20**	**MAP**
DCLINK	0.040	0.360	0.760	0.933	0.421
DCLINK-C	0	0	0	0.013	0.033
NC	0	0	0.013	0.013	0.037
LCS-B	0	0	0.013	0.040	0.022
LCS-U	0	0	0	0.013	0.024
Levenshtein	0	0	0	0	0.024

D. Summary

Our traceability problem differs from existing targets and poses unique challenges. First, existing work [3], [15]–[17], [23] do not consider the propagation of changes to source artifacts, which might result in missing linkage to target artifacts. To this end, we utilize the anchored field variables to perform static analysis to calculate the impact of changes. Second, due to the propagation of change, the target method does not necessarily contain similar names data constraint variables, making existing similarity-based methods [3], [7] ineffective. To this end, we follow the observations, use aggregation and database I/O relevance from our studies, and design prioritization solution to build links from data constraint changes to code.

VI. IMPLEMENTATION AND EVALUATION

We implement DCLINK on top of a Datalog-based program analyzer in Ant Group. To identify the data constraint changes, we parse the two versions of data constraints and implement a tree diff algorithm [14] to extract the edit script. To obtain the mapping from table columns to program variables, we implement a parser to analyze the configuration files of FinTech applications, such as the Mapper XML files in the MyBatis framework [18]. Lastly, we leverage the def-use analysis provided by platform S to achieve impact propagation in the traceability link analysis.

We evaluate the effectiveness and efficiency of DCLINK by investigating the following research questions:

- **RQ4**: How effectively does DCLINK locate the target methods according to data constraint changes?
- **RQ5**: How much are the time and memory overhead in each round of the traceability link analysis?

Data Availability. The subjects of our study, including data constraints and FinTech applications, cannot be shared because of confidentiality agreements in Ant Group.

A. Experimental Setup

We conduct the experiments using 75 data constraint changes, which are not investigated for RQ3 in the empirical study. Specifically, 89.33% (67/75) of data constraint changes involve clause additions and deletions, which is consistent with our second finding of the empirical study. Similar to the method of studying RQ3, we utilize the git history to determine the developers committing the updated code, and ask them to label the target methods for each data constraint

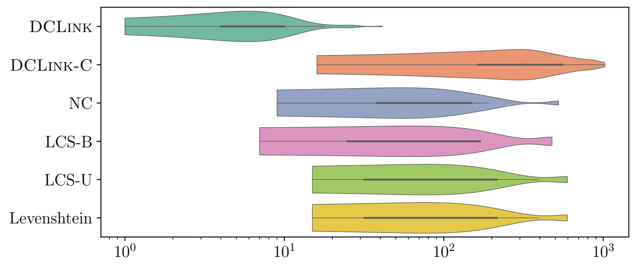


Fig. 9: Size distribution of method lists of each approach[3]

change, which serves as the ground truth for our evaluation. When evaluating DCLINK, we feed each data constraint pair and the old-version application code to it.

Baselines. To show the superiority of our traceability link recovery, we compare DCLINK with five conventional baseline approaches. First, we do an ablation study by comparing DCLINK to a baseline approach DCLINK-C that is set not to have method prioritization, similar to existing impact analysis-based approaches [24], [25]. Second, we adopt all possible baselines from the state-of-the-art test-to-code traceability recovery [7] set with the same thresholds t, namely *similarity-based approaches*, including the naming convention (NC) [5][2], two longest common sequence variants (LCS-B and LCS-U, $t = 0.8$), and Levenshtein distance ($t = 0.35$) [6]. Other execution-based (e.g., LCBA [5]) and textual approaches [26] are discarded since data constraints are not test cases that invoke application code directly and cannot be applied on the method level [7].

Metrics. We use the following commonly-adopted metrics [3], [8], [27] to measure the quality of the method lists computed by DCLINK:

- *Hit@K*: The percentage of the target methods that can be discovered by inspecting the top K of the returned list. An effective traceability link analysis should enable developers to find the target methods by examining as few methods as possible. The higher metric values indicate better quality of the solution.
- *MAP (Mean Average Precision)* [27]: The mean value of the average precision of predicting the target methods. When the ascending ranks of the target methods are h_1^i, $h_2^i, \cdots, h_{k_i}^i$ in the i-th traceability analysis, we have

$$MAP = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{k_i}\sum_{j=1}^{k_i}\frac{j}{h_j^i}\right)$$

Here N is the total number of analysis rounds. The higher the *MAP* value, the more precise the analysis is.

All experiments are conducted on a Macbook Pro with an eight-core M1 processor and 16 GB of physical memory.

[2]Similar to the prior study [7] which removes the "test" prefix for test methods, we have also removed the prefix of getters and setters.

[3]Failed linkages are not considered in the plot.

B. *Effectiveness of* DCLINK

To quantify the effectiveness of the traceability link analysis, we measure the size of the returned method list and compute the *MAP* and *Hit@K* values, where $K \in \{1, 5, 10, 20\}$. The small list size, the high *MAP* value, and the high *Hit@K* values mean effective recovery of data constraint changes with implementations.

Fig. 9 shows the distribution of different list sizes, ranging from 1 to 44. As shown in the violin plot, most of the returned list contains more than four and fewer than 20 methods. The average size is 8.66, and the median is 7, indicating that the developers only need to examine fewer than nine methods when inspecting the results of DCLINK for code update. To determine whether DCLINK's result contains significantly fewer methods, we conduct Mann-Whitney U Test with the null hypothesis that DCLINK does not significantly return fewer methods than DCLINK-C and other similarity approaches. Our results rejected this hypothesis with a p value less than 0.05, demonstrating the necessity of incorporating method prioritization originated from our study findings. Moreover, the similarity-based approaches often fail to identify the correct method demanding code updates. Specifically, our experiment shows NC, LCS-B, LCS-U, and Levenshtein failed to include the target methods in 56%, 79%, 68%, and 25% of linkage results. In these cases, developers need to pay extra effort to search the target methods manually from all other methods. To summarize, DCLINK significantly reduces the manual efforts in searching to-update methods upon data constraint updates.

Table II shows the *Hit@K* values of DCLINK and other baselines with different values of K. Specifically, the hit rate of DCLINK when inspecting the Top 1 and Top 5 methods are 0.040 and 0.360, respectively. Besides, 76% of the target methods are located successfully by examining the Top 10 methods. When inspecting the Top 20 methods, the developers can identify the target methods in 93.3% of the total cases. Notably, all the target methods are predicted successfully by DCLINK and DCLINK-C, as all the returned lists contain the target methods in the experimental subjects. However, DCLINK-C does not have a high hit rate within the Top 10 attempts and low *MAP* because the propagation might return a lot of candidates, which demands further prioritization. Similarity-based approaches have neither a high hit rate nor precision (≤ 0.04). Moreover, DCLINK's *MAP* value reaches 0.421, the highest among all approaches, showing a satisfactory recovery precision. This suggests that similarity-based approaches can be imprecise due to their unawareness of change propagations, while DCLINK bridges the gap via a change propagation analysis and thus is much more precise.

Answer to RQ4: DCLINK features the highest *Hit@K* and *MAP* among all of the approaches, which indicates its effectiveness in assisting developers in finding methods for code updates.

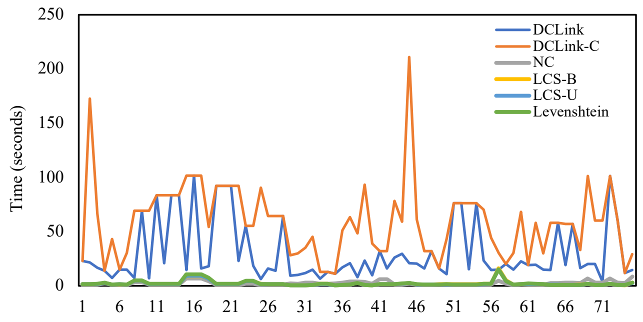


Fig. 10: Time cost of DCLINK and baselines

C. *Efficiency of* DCLINK

In our experiment, DCLINK and DCLINK-C consume 53.97 MB and 56.62 MB of peak memory to analyze all projects, respectively, while similarity-based approaches consume no more than 10 MB. Fig. 10 also shows the detailed statistics of time consumption. Overall, the time consumption is within 4 minutes. Specifically, a single run of the traceability link analysis takes 30.35 seconds on average. The major time cost comes from evaluating the queries performed by the Datalog engine. On average, the query execution consumes 30.16 seconds, while the change analysis only consumes less than 0.2 seconds. Although other similarity-based approaches generally take less than 9 seconds because they only make string comparisons, their effectiveness can hardly meet developers' needs. In general, DCLINK is efficient in terms of time and memory consumption, which is efficient enough to support developers in localizing and updating the target methods quickly. If integrated into the CI/CD pipeline of data constraint updates, DCLINK can accelerate the development workflow by providing supplementary tips to the developers within the development cycle. Thus, the overall efficiency of DCLINK is promising, showing its practical value in assisting FinTech system development.

Answer to RQ5: DCLINK consumes 30.35 seconds and 53.97 MB of peak memory on average, efficiently bridging data constraint changes with implementations.

D. *Discussions*

Threats to Validity. There are two potential threats to the validity of our work. The first is the subject selection bias. We only evaluate DCLINK upon the data constraint changes not investigated for RQ3 in the empirical study. Due to limited permissions, we could not access the FinTech applications of other technical units in Ant Group. However, the cases used for the study and the evaluation are selected randomly, making our experimental data reflect the performance of DCLINK for general cases to some extent. The second is the way of evaluating the effectiveness. In our evaluation, we conduct the control group study to show the superiority of DCLINK to the conventional practice. However, a more reasonable way to evaluate its usefulness should be to get feedback from developers in the long term, which remains our future work.

Limitations and Future Work. Although the evaluation demonstrates the advantages of DCLINK, it still faces two limitations. First, the input relations used in the impact propagation are generated exhaustively, while there might be a large proportion of the methods not contributing to the final result. Thus, the redundant computation occurs in the incremental impact analysis. Second, we observe that DCLINK can spend much time executing queries. When computing the impacted methods for different anchored field variables, the queries are processed independently. However, the impacted methods of the variables possibly overlap, introducing redundant computation in the query execution.

In the future, we can further explore the following directions for improvement. For example, it is meaningful to design an incremental mechanism for generating relational representations for an evolving application. Besides, we would improve the efficiency of query execution with a synergistic design, i.e., utilizing common intermediate results of the execution to avoid redundant computation.

VII. RELATED WORK

A. *Data Constraints*

Formulated as an important domain-specific language, data constraints have attracted increasing research interest in recent years. Yang et al. [4] and Florez et al. [11] comprehensively studied data constraints in database-backed applications, of which the findings guide the detection of inconsistent data constraints and the implementation of application programs [3], respectively. Besides, CFINDER [28] and AUTORECONCILER [12] target synthesizing data constraints from application code and runtime data, respectively, effectively alleviating the manual effort in writing data constraints. Similarly, CONSTROPT also synthesizes data constraints, which further guide the application refactoring to improve the performance [29]. In another recent work, EQDAC determines the data constraint equivalence efficiently to avoid the redundant runtime verification [30]. Our work concentrates on a different usage scenario from the ones in existing studies, where the application code demands modification according to data constraint changes. Essentially, DCLINK provides a systematic mechanism of bridging evolving system specifications expressed by data constraints and system refinement, which can be generalized and applied to other similar software systems. For example, the development of networking systems and protocols has demonstrated a similar pattern [31], [32], where the updates of data constraints and code changes are highly co-related. In the future, we intend to extend our work to study and bridge the data constraint changes and implementation updates in networking systems.

B. *Traceability Link Recovery*

There have been a broad number of literature working on traceability link recovery, establishing the linkage between different modules of the programs or multiple artifacts of the systems [3], [7], [15]–[17], [23], [33]–[39]. For instance, several studies summarize and link system requirements with the corresponding implementation by applying various techniques, including machine learning [15], retrieval models [40], and dynamic analysis [41]. The most relevant work to DCLINK is the test-to-code traceability study [7], [33]. A test case depicts the system specification, which bears similarities to data constraints in our work. However, previous studies do not consider the change impact of the test cases, only analyzing the relationship between specific test cases and program methods. Besides, DCLINK utilizes the configuration file to obtain the relationship between variables and attributes in data constraints instead of simply matching with names [7] and dynamic profiling [33]. Therefore, DCLINK supports a more robust traceability link recovery to bridge the data constraint and application changes.

C. *Datalog-based Program Analysis*

Datalog is a logic programming language widely used in declarative program analysis [19], [42]–[49]. In this work, we feed the change patterns of data constraints and derive basic relations from the application code of FinTech systems. Our effort shows the feasibility of bridging the gap between software artifacts of other forms, such as data constraints, to software application code. Datalog has also been applied in incremental program analysis [20], [50]–[52]. We do not claim to improve the performance of incremental analysis, but target a different problem, that is, how to abstract and propagate the change impact of data constraints in FinTech applications. We believe it would be promising to leverage existing techniques of incremental Datalog evaluation [53] to accelerate the analysis of application code.

VIII. CONCLUSION

In this work, we conduct a comprehensive study of data constraint changes in FinTech systems, demonstrating the necessity of bridging data constraint changes and implementations. We then propose the traceability link analysis DCLINK for inferring the target methods according to data constraint changes, effectively assisting the developers in updating their implementations. Our work makes the first step towards providing implementation support for the developers of evolving FinTech systems.

ACKNOWLEDGMENT

We thank the anonymous reviewers for valuable feedback on this paper, which helped improve its presentation. Rongxin Wu is the corresponding author and supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and NSFC 62272400. Other authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei. We also appreciate Dr. Xiaoheng Xie for insightful discussions.

REFERENCES

[1] Amazon. Amazon Prime Day 2022 – AWS for the Win! https://aws.amazon.com/blogs/aws/amazon-prime-day-2022-aws-for-the-win/, 2022. [Online; accessed 30-Aug-2022].

[2] The New York Times. Knight Capital Says Trading Glitch Cost It $440 Million. https://archive.nytimes.com/dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/, 2022. [Online; accessed 30-Aug-2022].

[3] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. Retrieving data constraint implementations using fine-grained code patterns. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1893–1905. ACM, 2022.

[4] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1098–1109. ACM, 2020.

[5] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 209–218. IEEE, 2009.

[6] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[7] Robert White, Jens Krinke, and Raymond Tan. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 861–872, 2020.

[8] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214, 2014.

[9] Shay Artzi, Sunghun Kim, and Michael D Ernst. Recrashj: a tool for capturing and reproducing program crashes in deployed applications. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–296, 2009.

[10] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE, 2012.

[11] Juan Manuel Florez, Laura Moreno, Zenong Zhang, Shiyi Wei, and Andrian Marcus. An empirical study of data constraint implementations in java. *Empir. Softw. Eng.*, 27(5):119, 2022.

[12] Tianxiao Wang, Chen Zhi, Xiaoqun Zhou, Jinjie Wu, Jianwei Yin, and Shuiguang Deng. Data constraint mining for automatic reconciliation scripts generation. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1119–1130. ACM, 2023.

[13] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324. ACM, 2014.

[15] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 97–108, 2019.

[16] Hui Gao, Hongyu Kuang, Xiaoxing Ma, Hao Hu, Jian Lü, Patrick Mäder, and Alexander Egyed. Propagating frugal user feedback through closeness of code dependencies to improve ir-based traceability recovery. *Empir. Softw. Eng.*, 27(2):41, 2022.

[17] Juan Manuel Florez. Automated fine-grained requirements-to-code traceability link recovery. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 222–225. IEEE, 2019.

[18] MyBatis. MyBatis Documentation. https://mybatis.org/mybatis-3/, 2022. [Online; accessed 30-Aug-2022].

[19] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–248, 2014.

[20] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1–15, 2021.

[21] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[22] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[23] Kevin Moran, David N Palacio, Carlos Bernal-Cárdenas, Daniel McCrystal, Denys Poshyvanyk, Chris Shenefiel, and Jeff Johnson. Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 873–885, 2020.

[24] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. *ACM SIGSOFT Software Engineering Notes*, 28(5):128–137, 2003.

[25] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*, pages 112–122, 2007.

[26] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[27] P.J. Layzell and P. Loucopoulos. A rule-based approach to the construction and evolution of business information systems. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 258–264, 1988.

[28] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. Protecting data integrity of web applications with database constraints inferred from application code. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 632–645. ACM, 2023.

[29] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging application data constraints to optimize database-backed web applications. *Proc. VLDB Endow.*, 16(6):1208–1221, 2023.

[30] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang. Verifying data constraint equivalence in fintech systems. *CoRR*, abs/2301.11011, 2023.

[31] Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Meeting slos in cross-platform nfv. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, pages 509–523, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 493–509, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, 28(10):970–983, 2002.

[34] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 68–71. IEEE, 2010.

[35] Muhammad Abbas. Variability aware requirements reuse analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 190–193. IEEE, 2020.

[36] Serin Jeong, Heetae Cho, and Seonah Lee. Agile requirement traceability matrix. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 187–188, 2018.

[37] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. Retrieving data constraint implementations using fine-grained code patterns. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1893–1905, 2022.

[38] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. Traceability transformed: Generating more accurate links with pre-trained bert models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 324–335. IEEE, 2021.

[39] Michael Rath, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. Traceability in the wild: automatically augmenting incomplete trace links. In *Proceedings of the 40th International Conference on Software Engineering*, pages 834–845, 2018.

[40] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery: A ten-year retrospective. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, page 1. ACM, 2020.

[41] Malcom Gethers, Huzefa H. Kagdi, Bogdan Dit, and Denys Poshyvanyk. An adaptive approach to impact analysis from change requests to source code. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 540–543. IEEE Computer Society, 2011.

[42] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. Data-driven synthesis of provably sound side channel analyses. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 810–822. IEEE, 2021.

[43] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2006.

[44] Magnus Madsen and Ondřej Lhoták. Safe and sound program analysis with flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 38–48, 2018.

[45] Ramy Shahin, Marsha Chechik, and Rick Salay. Lifting datalog-based analyses to software product lines. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 39–49, 2019.

[46] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[47] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.

[48] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2005.

[49] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. *ACM SIGPLAN Notices*, 36(5):24–34, 2001.

[50] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[51] Chungha Sung, Shuvendu K Lahiri, Constantin Enea, and Chao Wang. Datalog-based scalable semantic diffing of concurrent programs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 656–666, 2018.

[52] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. Towards elastic incrementalization for datalog. In *23rd International Symposium on Principles and Practice of Declarative Programming*, pages 1–16, 2021.

[53] Leonid Ryzhyk and Mihai Budiu. Differential datalog. *Datalog*, 2:4–5, 2019.