Building SOFL-to-Java Traceability Links using Multi-dimensional Similarity Measures

JIANDONG LI[1] SHAOYING LIU[1]
RUNHE HUANG[2]

Abstract: To achieve an automatic formal specification-based program fault detection, the open problem of how to automatically link the components in the formal specification to the corresponding ones in the implemented code must be addressed. To reduce the manpower and time cost, some automated techniques have already been developed but their effectiveness is limited mainly due to the over dependency of textual similarity. In this paper, we present an automatic method for constructing the traceability links between SOFL formal specifications and Java program code. Unlike the existing work, our method not only considers the semantic similarity, but also structural, functional, and relational similarities as the measurement dimensions. It also operates at multiple levels of a specification, such as data flows, processes, and modules, to establish fine-grained link relationships between artifacts. Further, we conduct a comprehensive empirical evaluation of the proposed method using selected two modules of a critical ATM system's SOFL formal specification and its Java implementation with the size of 951 code of lines and 36 traceability links. The result shows that we can establish SOFL-to-Java links with the precision of 97.2% which is close to highest accuracy of existing naming convention technique in the situation of consistent identifier and the precision of 88.8% illustrating high performance in precision and generality in the situation of inconsistent identifier.

Keywords: Software Traceability Links, Formal Specification, SOFL, Program Verification

Introduction

Our research in this paper focuses on the establishment of fine-grained traceability links that connect "high-level" SOFL formal specification artifact [1] to "low-level" code artifact written in Java programing language (SOFL-to-Java). This work stems from a subproblem existing both in the specification-based program inspection [2] and in software construction monitoring process in Human-Machine Pair Programing (HMPP) [3]. When current traceability links automatic techniques were adopted in the specific SOFL-to-Java trace link recovery, two major shortcomings were found:

1) Limited Measure Dimensions of Artifact Similarity. Existing traceability link techniques tend to mainly use a single semantic measurement dimension to model relationships between artefacts. Here we refer semantic narrowly to name similarity or textual similarity. The adoption of this relatively single measurement dimension is mainly due to artifacts' logical abstract gap and the amount and types of information extracted from different artifacts. The gaps are mainly caused by the fact that various artifacts are represented or described in different languages, such as requirements written in the SOFL formal language [1] and the code written in Java programing language. This is problematic in practical application and generality as developers usually use inconsistent name or identifiers to implement the requirements. When it comes to trace link recovery between a formal specification and code, we propose some new similarity measure dimensions that includes structural dimension, functional dimension, relational dimension that distinguish our work from previous work. The explanation of these new measure dimensions is given detailly in the following approach section.

2) Limited Link Granularity. Some previous work concentrates on the establishment of coarse-grained artifact level traceability link (e.g., user stories and class files) while some other existing work focuses on the building of fine-grained connections between components of artefacts (e.g., test-to-code traceability links on class level and method level). However, the method level granularity is not adequate and more granular traceability links are needed to generate a checklist for our specification-based program inspection or fault detection purpose. Our work on connecting SOFL formal specifications to Java program code focuses on the module and class level where the module or class component of the SOFL specification is linked to a Java class, the process level where the process component of the SOFL specification is linked to a Java method or class method, the data flow level where the data flow component of the SOFL specification is linked to a Java class filed or variable or constant. The expression of "multi-level" in the paper title means that we simultaneously address the module and class level, the process level, and the data flow level, which differs from existing work.

The limitation discussed above arises from the technical limitation of existing traceability techniques and the difference between artefacts involved in the problems to be solved. In this paper, we present an automatic SOFL-to-Java traceability links technique, aiming to overcome the weakness or limitations of existing approaches when adopted to support automatic specification-based program inspection. The proposed method combines a wide range of similarity measurement dimensions and extracts corresponding attribute set to produce a single score to predict trace links. Our approach is comprehensively evaluated using selected two modules of SOFL formal specification of a critical ATM system case [14] with the size of 951 code of lines,

1 Hiroshima University, Higashihiroshima, Hiroshima 739-8527, Japan
2 Hosei University, Koganei, Tokyo 184-8584, Japan

12 Java classes and 36 traceability links at multilevel. The result of the experiment shows that the proposed method with a 97.2% precision is close to the naming conventions method [4] with a 100% precision in the case of using same identifiers. Further, our method could also solve traceability links problem with the 88.8% accuracy in the situations of using inconsistent identifiers in whole programs. The main contributions of this paper are:

- A more specific framework for specification-based software construction monitoring realization in human-machine pair programming.
- An approach to SOFL-to-Java traceability link for completed program that combines multiple similarity measurement dimensions and achieves multilevel trace link.
- A comprehensive evaluation of proposed approach to SOFL-to-Java traceability link for completed programs at multilevel.

Specification-based Software Construction Monitoring and Traceability Links Problem

We describe a more specific framework for software construction monitoring (SCM) to be realized in human-machine pair programming by properly adding the specification to original framework and address the SOFL-to-Java traceability links problem.

1.1 Specification-based SCM

Programming is the major activity to provide working software advocated in the agile development paradigm but lacks effective techniques to address the challenges in ensuring software quality, productivity, and maintainability. Liu proposed a human-machine pair programming (HMPP) methodology to support efficient and reliable programming [3]. HMPP is an extension and refinement of pair programming by replacing one of the programmers with an intelligent machine which can either identify potential software defects and violation of standards in the program or predict useful program segments for enhancing the robustness and the completeness of the program. It can overcome the disadvantages of high cost and programmers' cooperation issue in conventional pair programming.

HMPP includes two key techniques: software construction predicting (SCP) for program segments generation automatically and software construction monitoring (SCM) for timely fault detection and error-free completion which is related to our traceability links concern. The technique SCM here refers to the automatic and dynamic check of whether the current software version satisfies the required properties (e.g., requirements in the specification, termination of loop body). The original SCM realization framework is shown in the internal rectangle part of Fig. 1. Specific property to be checked is determined or formed on the basis of both the pre-prepared property knowledge stored in the knowledge repository and the information of current software version through syntactical analysis.

In order to detect requirements-related semantic faults which refers to the ones that are inconsistent with the requirements or the developers' programming intention in software construction

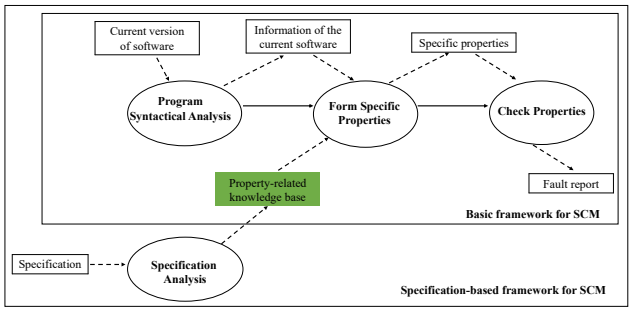

Figure 1. Specification-based framework for software construction monitoring.

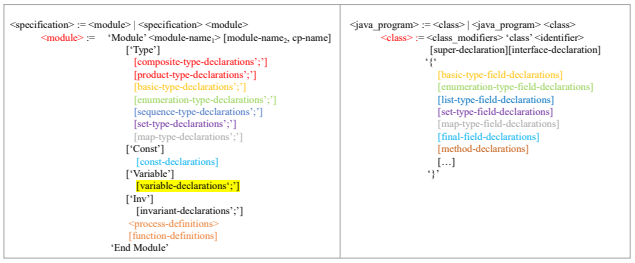

Figure 2. Components' corresponding relationships between SOFL and Java program.

monitoring, we introduce specification data and the specification analysis activity into original software construction monitoring realization framework as shown in Fig. 1. We call it specification-based software construction monitoring. By taking advantage of specification, this realization framework can benefit specification-based programing in terms of ensuring its quality by utilizing specification-based inspection technique and specification-based testing [5]. It should be pointed out that the specification-based framework for specification-based programming is a more specific one as original SCM realization framework is proposed for general programming and they have an inclusive relationship.

1.2 Traceability Links for Completed Program Problem

There is a challenge involved in the realization of specification-based SCM, which is to identify what subset of the specification has been implemented, what properties in the specification are implementing and what subset of specification is unimplemented. Fault detection could be done once an appropriate subset of specification has been chosen or determined.

Since the identification of the appropriate subset of the specification that has been implemented by the program under construction is a very difficult task, we actually first explore a solution that maps the specification to the completed program in this paper and then make a solution to the mapping from the integral specification to the partial program as our future work. As for the specification, we choose SOFL (Structured-Object-based-formal Language) [1] which generates accurate and unambiguous requirements description like other formal languages such as Z and VDM [6]. The chosen of SOFL is mainly based on its successful application to modeling software systems in the collaboration with industry and on our familiarity with it as it was developed by the second author [7]. Further, the Java programming language is

selected to implement specifications for two reasons. One is that it is still popular for decades since its birth and the other one is the availability of the relatively rich functional and non-functional bug dataset, such as NFBugs, MUBench, DEFECT4J, and iBUGS [8]. These bug datasets will be used as a basis to extract fault pattern in our future work of common programming fault detection which is not requirement specific.

Before defining the SOFL-to-Java traceability link for both completed programs, a brief introduction to SOFL is given. SOFL offers a formal and rigorous language to describe requirements in an unambiguous manner. The SOFL language is mainly comprised of the following components: class, module, data flow, data store, process, and function. More information about the structure of SOFL is detailed in [1]. Fig. 2 shows the usual components' corresponding relationships in the transformation from SOFL specification to Java program implementation. Data flows which could be defined with a basic type (e.g., int type in SOFL) or compound type (e.g., set type in SOFL) or user-defined type (e.g., composite type in SOFL) are understood as data item that need to be taken in and out by the required functions. Data flows except composite and product type usually are implemented as the fields of Java class. Data stores corresponding to the variable part in Fig. 2 are variables that hold data in rest for use by processes (equivalent to operations or functions in general term) and are usually implemented as fields of a Java class. Processes express or describe a specific function or operation via pre-condition and post-condition based on predicate logic and are usually transformed into the methods in Java class. The module containing data flow and processes could be interpreted as the functional or behavioral abstract of the object and usually implemented as a Java class.

SOFL-to-Java Traceability Links for Completed Program Definition. Our goal is to make a solution for capturing meaningful information regarding the transformative relationships between components of SOFL specifications and the ones of Java program and for establishing their trace links. More specifically, given a SOFL formal specification S such that $S = \{S\_component_1, ..., S\_component_n\}$ and a completed Java program P implementing specification S such that $P = \{P\_component_1, ..., P\_component_m\}$, we aim to discover whether a trace link L exists between all possible pairs of components in S and P such that $L = \{(s, p) \mid s \in S, p \in P, s \leftrightarrow p\}$ where each pair of components s and p are said to be the transformative links.

Approach

We present the proposed multilevel SOFL-to-Java trace links establishment approach here from the main aspects of measurement dimensions, its workflow, various components' attributes design, used similarity measurement techniques, respectively. To ensure the comprehensibility of our discussion, a money-box example is given in Fig. 3. The money-box has three required functions: save money, check money and purchase toy with a fixed price of 1000 Japanese yen.

```
module Money_Box
        const
         toy_price = 1500
        var
         money_box: int
        process Save_Money(amount: int)
                ext wr money_box
                pre amount > 0.0
                post money_box = ~money_box + amount
        end_process;
        process Check_Money() total: int
                ext rd money_box
                post total = money_box
        end_process;
        process Purchase_Toy() expense: int | warning: string
                ext wr money_box
                post ~money_box >= toy_price and expense =
toy_price and money_box = ~money_box – toy_price
                or
                ~money_box < toy_price and warning = "the shortage
of the money in the money_box, failed transaction"
        end_process;
end_module;
```

```
public class Money_Box {
        public static final int toy_price = 1500;
        private int money_box = 0;
        void save_money(int amount) {
                if (amount >0) {
                        money_box += amount;
                } }
        int check_money() {
                return money_box; }
        int purchase_toy() {
                int expense;
                if (money_box >= toy_price) {
                        money_box -= toy_price;
                        expense = toy_price;
                        return expense;

                } else {
                        System.out.println("the shortage of the money in the
money_box, failed transaction");
                        expense = 0;
                        return expense;
                } }
        public static void main(String args[]) {
                Money_Box ljd = new Money_Box();
                System.out.println(ljd.check_money());
                ljd.save_money(2000);
                int cost = ljd.purchase_toy();
                System.out.println("cost is " + cost);
                System.out.println(ljd.money_box);
        }
}
```

Figure 3. A money-box example of the transformation from SOFL specification to Java implementation.

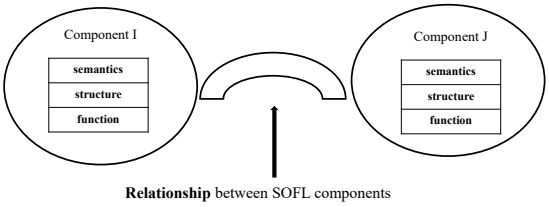

Figure 4. Attributes' dimensions of SOFL components.

1.3 Measurement Dimensions

Formal specifications like SOFL are different from other artefacts like the informal requirement documents written in natural language. We observe that each component of a SOFL specification has its own attributes including semantics, structure and function, and correlation attributes during its interaction with other components as shown in Fig. 4. Although there are some variants, these four different kinds of attributes of the SOFL specification will always be remained in their implementation in a programming language. This is the principle that supports the proposed traceability links method.

Semantic Dimension. By semantics we mean that the identifiers assigned in components of SOFL may be semantically similar or be kept in their corresponding Java implementation by developer. For instance, the process name "Save_Money" in SOFL is still used in the corresponding "save_money" method of Java class "Money_Box" as shown in Fig. 3.

Structural Dimension. Here we refer structure mainly to type and the type of component in different level of SOFL specifications. The structure of data flow component is directly its defined type. The structure of a process or function in SOFL is represented by a list of its input data flows and output data flows. The structure of a module or class in SOFL is the type list of the data flows in its const part, types part and var part. For instance, the structure of "money_box" data flow is "int" type, the input parameter structure of process "Save_Money" is "int" type, the field structure of "Money_Box" module is "const, int" as shown in Fig. 3. The structural attributes are still kept in its Java implementation.

Functional Dimension. Here we mean the function of different which refers to the purpose of it could be revealed by the dynamic

testing based on the test data generated by it and the function of component in different level of SOFL specification are different. For instance, the function of the "money_box" data flow is to store data of int type, the function of "Save_Money" process is to deposit money and "Money_Box" module has three functions: "Save_Money", "Check_Money" and "Purchase_Toy" as shown in Fig. 3. These attributes of functional dimension are actually not extracted and used in our following experiment as SOFL-to-Java traceability linked could be effectively established without it and the extraction procedure requires test data date generation and dynamic testing that leads to low time efficiency.

Relational Dimension. Here we refer relation to the interactions between different components in SOFL. Different data flows work together to be the input data flows of a process. One data flow type may be used as one input of a process or interact with the body of a process or be used as the return type of a process. One data flow or data store is constituted element of a process. A module usually includes several processes. For instance, the constant "toy_price" is used in the body of process "Purchase_Toy", the data store "money_box" is used as read mode in the body of "Check_Money" process and as write mode in two other processes, the "Money_Box" module has two data flows and three processes as shown in Fig. 3.

1.4 Workflow

Our approach reckons which component of Java program implements which component of SOFL specification. The establishment process of these trace links is illustrated in Fig. 5. The process consists of four major steps: attributes extraction for specification and program components respectively, similarity measurement and ranking. Each step that stands for an operation is represented by a diamond and each data item is represented by a box. An arrow from a box to an operation means that the data item of the box is an input to the operation, and an arrow from an operation to a box means that the data item of the box is an output of the operation. An arrow from one operation to another shows a control flow.

In general, our approach starts by extracting the attributes about each component in SOFL specification and Java program. It then creates candidate links between SOFL specification components and Java program components. It then calculates the similarity scores through some similarity measurement techniques based on the extracted attributes and assigns similarity scores to the candidate links. These similarity scores are the basis to rank the candidates and identify which of them is the true traceability links. The identified traceability links are used for program verification and fault detection purpose.

It should be pointed that specification need to be preprocessed like functional scenario derivation and specification purification. Both of them are for the purpose of extracting more accurate attributes for each component. Specification purification here refers to separating the composite type and product type data flow from the module in which they are declared. According to our specification to program transformation experience, we find that composite and product type data flow are usually implemented as class in Java programming language and not implemented as field of the class which implements the corresponding module. Functional scenario is mainly used to solve one-to-many traceability links situation introduced in the later 3.6 section. Besides, we introduce component type grouping trace link trick for reducing computation cost and improving trace link accuracy which is shown in our experimental and evaluation section. Component type grouping trace link means that we only calculate the similarity score for SOFL specification components and Java program components pairs which has consistent type. For example, the component pair which has constant data flow of SOFL specification and the method of Java class is invalid. Before similarity score calculation, type substitution should be done in advance based on some prior SOFL-to-Java data type transformation knowledge and class dependence graph. The class type data will be replaced by the type list of its constitutive fields.

1.5 Attributes Extraction of Multilevel Components

We design and extract different attributes for different components of the SOFL specification from the perspectives of semantics, structure, function and interactive relationships as described in section 3.1. The reason is that one kind of component has some unique attributes that can distinguish itself from other kinds of components. The 33 designed and extracted attributes for these components of SOFL specification is presented in Table 1. We marked the name, explanation of each attribute and type of corresponding attribute value. The attributes combination for constant type data flow, basic type data flow, set type data flow, map type data flow, sequence type data flow, enumeration type data flow, composite type data flow, product type data flow, process, function and module are shown in Table 2.

The unique attribute for data flow of constant type is its value attribute. The unique and different attributes for data flow of map type are its domain type and its range type. The attribute that distinguishes the data flow of enumeration type from others is its enumeration value. The distinguishable attribute that is extracted for data flow of set and sequence type is the data type of its element. If a data flow is also a data store, then the unique attributes for it also include the information of processes that read or write the data store. A process or function has unique attributes related to its own input parameters (data flow) and return data flow type. Besides, its interaction with the data flows that are declared in constant part, type part and var part of the module in specification and the call relations between itself and other functions or process could also be extracted. The data flows of composite type and product type have own attributes like associated fields' type and amount while the module has additional attributes about associated and constitutive processes or functions.

The above description is about the attribute extraction for the multilevel components of SOFL specification. The designed and extracted attributes for different components of Java program is actually the same as the ones extracted from the SOFL specification as shown in Table 2.

Table 1. EXTRACTED ATTRIBUTES FOR SOFL COMPONENTS.

Attribute ID	*Attribute Name and Explanation*	*Attribute Value Type*
1	“identifier”: what is its identifier?	String
2	“type”: Will it be transformed into class or field or method in Java programs?	String
3	“data_type”: what is its type?	String
4	“constant_value”: what is its constant value?	String, int or double
5	“interacted_method_body_identifier”: What are the identifiers of the methods using this component in their method bodies?	String
6	“interacted_method_body_amount”: How many methods using this component in their method bodies?	int
7	“interacted_method_amount_outside_class”: How many other classes’ methods using this component?	int
8	“interacted_method_amount_in_class”: How many local class’s methods using this component?	int
9	“interacted_class_type”: What are the types of interacted classes?	String
10	“interacted_class_identifier”: What are the identifiers of interacted class?	String
11	“fields_type”: What are the types of constitutive fields for a class?	String
12	“fields_identifier”: What are the identifiers of constitutive fields for a class?	String
13	“fields_amount”: What is the amount of constitutive fields for a class?	int
14	“interacted_field_type”: What are the types of interacted other components?	String
15	“interacted_field_amount”: How many other field-level components interacting with it?	int
16	“ interacted_method_parameter_part_identifier”: What are the identifiers of the methods using this component as the input parameter?	String
17	“ interacted_method_parameter_part_amount”: How many methods using this component as the input parameter?	int
18	“ interacted_method_return_type_identifier”: What are the identifiers of the methods using this component as the return type?	String
19	“interacted_method_return_type_part_amount”: How many methods using this component as the return type?	int
20	“domain_type”: What is the type of the domain of a map type field?	String
21	“range_type”: What is the type of the range of a map type field?	String
22	“element_type”: What is the data type of its element in compound type field?	String
23	“enumeration_value”: What is the enumeration value?	String
24	“parameter_type”: what types are the input parameters of a method?	String
25	“parameter_amount”: How many parameters does a method have?	int
26	“return_type”: What is the return type of a method?	String
27	“interacted_call_identifiers”: What are the identifiers of other methods calling the method?	String
28	“interacted_call_amount”: How many methods calling the method?	int
29	“method_amount”: How many methods does a class have?	int
30	“method_identifier”: What are the identifiers of methods that a class have?	String
31	“method_parameters_types”: What are the types of the parameters of methods that a class have?	String
32	“method_return_type”: What are the return type of methods that a class have?	int
33	“interacted_class_amount”: How many class does it interact?	int

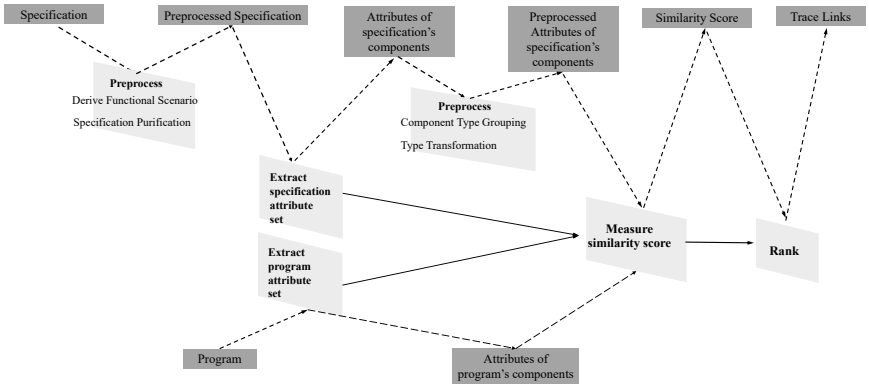

Figure 5. Workflow of establishing SOFL-to-Java trace links.

Table 2. ATTRIBUTES FOR VARIOUS COMPONENTS.

Component	*Attribute Combination*	*Attribute Amount*
constant type data flow / final field in Java	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 33	11
Basic type data flow / basic type field in Java	1,2,3,5,6,7,8,9,10,14,15,33	12
Set type data flow / set type field in Java	1,2,3,5,6,7,8,9,10,14,15,16,17,18,19,22,33	17
Sequence type data flow / list type field in Java	1,2,3,5,6,7,8,9,10,14,15,16,17,18,19,22,33	17
Map type data flow / map type field in Java	1,2,3,5,6,7,8,9,10,14,15,16,17,18,19,20,21,33	18
Enumeration type data flow / enumeartion type field in Java	1,2,3,5,6,7,8,9,10,14,15,16,17,18,19,23,33	17
Composite type or producte type data flow	1,2,3,7,8,9,10,11,12,13,14,15,16,17,18,19,33	17
Process or function / method in Java	1,2,3,9,10,14,15,24,25,26,27,28,33	13
Module / class in Java	1,2,3,5,6,7,8,9,10,11,12,13,14,15,17,18,19,29,30,31,32,33	23

1.6 Selected Similarity Measurement Techniques

The extracted attribute sets for multilevel components of both SOFL specification and Java code are the basis to measure their similarity. To calculate the similarity score or trace link score, we select four existing similarity measurement techniques including longest common subsequence-both (LCS-B), cosine similarity, the similarity measurement between two numbers and Jaccard coefficient. These four techniques produce a score between 0 and 1 for each extracted attribute.

Longest common subsequence-both (LCS-B) [13] is variant of longest common subsequence and is used to measure the similarity of the "identifier" attribute only. LCS-B calculates the textual similarity between two strings $v_{sofl}$and v_{java} by dividing the length of their longest common subsequence by the greater of the length of them. It can be expressed as follows:

$$score(v_{sofl}, v_{java}) = \frac{|LCS(v_{sofl}, v_{java})|}{\max(|v_{sofl}|, |v_{java}|)} \quad (1)$$

We choose LCS-B as the name similarity technique other than naming conventions or Levenshtein distance etc. This is because LCS-B performs better than others in recall and it produces the same score as naming convention technique even in the extreme situation that the names of components in SOFL are the same as the names of components in Java.

Cosine similarity is mainly used to measure the similarity of these attributes whose value type is string type except "identifier" attribute. Cosine similarity measures the similarity of two strings v_{sofl} and v_{java} through calculating the cosine of the angle between two non-zero vectors A and B which representing these two strings $v_{sofl}$and v_{java}. It can be expressed as follows:

$$score(v_{sofl}, v_{java}) = \frac{A \cdot B}{||A||||B||} \quad (2)$$

String vectorization which is actually counting the frequency of each character in each string is an essential preliminary work before calculating the similarity of two strings. Cosine similarity is usually used in positive space in our application scenario, so the value given is between 0 and 1.

The similarity between two numbers v_{sofl} and v_{java} is measured by dividing the minimum of two numbers by the maximum of two numbers. It is mainly used to measure the similarity of these attributes whose value type is int type. It can be expresses as follows:

$$score(v_{sofl}, v_{java}) = \frac{min(v_{sofl}, v_{java})}{\max(v_{sofl}, v_{java})} \quad (3)$$

We choose ratio of two number as similarity rather than Euclidean distance and Mahalanobis distance as our intention in order to get a similarity score between 0 to 1 for each extracted attribute.

Jaccard coefficient [19] is mainly used to measure the similarity of those attributes who value type is set type. Given two sets $v_{sofl}$and v_{java} , Jaccard coefficient calculates their similarity score by dividing the number of elements in their intersection by the number of elements in their union. It can be formulated as follows:

$$score(v_{sofl}, v_{java}) = \frac{|v_{sofl} \cap v_{java}|}{|v_{sofl} \cup v_{java}|} \quad (4)$$

Jaccard coefficient is selected by considering the multiple value and disorder characteristic of the input data flows of SOFL process and parameters of Java method. It should be pointed out that most set type attributes are transformed into string type and cosine similarity is used to measure their similarity to simply calculation and improve the accuracy of similarity calculation in our

implementation. Besides, it could also satisfy our score range requirement.

1.7 Trace Link Prediction

SOFL-to-Java Prediction. A matrix of similarity score is generated or constructed after the similarity calculation by the integrated application of above similarity measurement techniques:

$$Score\_Matrix \in \mathbb{R}^{|SOFL\_C|\times|JAVA\_C|} \quad (5)$$

Where $SOFL_C$is the set of components extracted from SOFL specification and $JAVA\_C$ is the set of components extracted from the Java implementation. Each element of the score matrix $Score\_Matrix$ is the traceability score for a given SOFL-to-Java component pair$(s_c,\ p_c) \in (SOFL_C \times JAVA_C)$.

Each row of the score matrix is a similarity score vector of a SOFL component to each component of Java program code. This similarity score vector is then ranked and compared with a threshold. If the maximum of this similarity score vector is greater than the threshold, the corresponding SOFL and Java component pair will be linked.

1.8 One-to-Many SOFL-to-Java Trace Links

The above discussion may create an illusion that the components' mapping or linking is a one-to-one corresponding relationship. The realistic SOFL-to-Java traceability link is actually more complicated. This is because that a process in SOFL specifications may be implemented as multiple methods in Java programs. In other words, there exists one-to-many mapping relationships between SOFL process and Java method. There are two reasons contributing to this phenomenon.

The first reason is that a process, especially with multiple ports, allows exclusive input or output data flows and different output data flows depend on different input data flows. Besides, both the single port and multiple-port processes could derive multiple functional scenarios and these functional scenarios cannot be implemented fully in a single Java method sometimes. It should be pointed that a functional scenario is corresponding to a program path in a Java method and a Java method could implement multiple functional scenarios. In this case, we propose to do process decomposition and break process into multiple functional scenarios [9], record the inclusive relationship between process and functional scenarios and use above discussed method to establish the trace links between the functional scenarios of SOFL and the methods of Java program.

The second reason is the personal programming habits of developers. This means that some developers tend to divide a process implementation into several steps and each step is implemented as a method for good logic, readability and comprehensibility of code. For example, some developers may implement the LCS-B algorithm used in this paper as one single Java method while some other developers may first implement a method to obtain the longest common string and then write another method to implement LCS-B algorithm. In this case, a call graph or a call chain will be formed. We propose to use the current static analysis technique like pointer analysis [10] to construct the call graph or call chains, extract attributes for the call chains and establish the trace links between processes to the call chains.

Table 3. THE SUBJECT.

ATM Specification	*ATM Java Code*
module, *data flow*, process	**class**, *field*, method
Module(2), *Composite type(8), Product type(2), const(5), map type(4), sequence type(2), enumeration type(1), basic data flow(2), data store(6) (overlap here)*，process(6), process functional scenario(3), function(1),	**class(12),** *const(5), map(4), seq(2), enumeration(2, 1 noise), basic(2),* method(15, 5 noise)
	951 code of lines
36 multi-level traceability links	

Table 4. EXPERIMENTAL RESULTS FOR RQ1 AND RQ2.

Situation	*Attributes Dimension*	*Grouping Traceability Link*	*Accuracy*
consistent identifiers	Semantical+ Structual+ Relational	Not adopted	94.4%(34/36)
		adopted	97.2%(35/36)
inconsistent identifiers	Structual+ Relational	Not adopted	80.5%(29/36)
		adopted	88.8%(32/36)

Implementation and Evaluation

This section shows the implementation, the research questions, the subject and the performance of our method compared with existing approach, and a discussion of threats to its validity.

The proposed traceability link method and workflow are implemented using Python programming language. To extract attribute sets for components of Java program, we use the open source javalang python library which provides a lexical analyzer and parser for Java source code and make some modifications. Besides, we rely on javalang to tokenize SOFL specification and make some rules based on keywords or modifiers in SOFL language to extract attribute set for components of SOFL specification for a small money-box example which is not used as experimental data here. Lastly, we implement the three used similarity measurement algorithms including longest common subsequence-both (LCS-B), our variant of Manhattan distance, cosine similarity and Jaccard coefficient by ourselves. A tool supporting our work will be released in the future.

For the subject as shown in Table 3, we use a critical ATM system example [14] which was specified using SOFL and was implemented using Java programming language. The SOFL specification modeling ATM system contains totally 36 traceability links. Among them, there are 2 module-level trace links, 8 composite type data flow level trace links, 2 product type data flow level trace links, 5 constant data flow level trace links, 4 map type data flow level trace links, 2 sequence type data flow level trace links, 1 enumeration type data flow level trace links, 2 basic type data flow level trace links, 6 process level trace links, 3 process functional scenario level trace links and 1 function level trace links. The implemented Java program of ATM specification

has the size of 951 code of lines, 12 Java classes, 5 constant fields, 4 map type fields, 2 list type fields, 2 enumeration type fields, 2 basis type fields and 15 methods. It should be pointed that 1 of 2 enumeration type fields and 5 of 15 methods are noise data in the Java implementation.

In order to achieve a comparative analysis, we choose the latent semantic indexing (LSI) [11] from existing information retrieval-based techniques and naming conventions technique [4] as the baseline. They are chosen because they are two of the most common techniques for establishing traceability links and naming conventions is best for accuracy in [13].

To evaluate our work, we are concerned about the following research questions: (1) how effective is our proposed method in modeling SOFL-to-Java traceability links in the situation of consistent identifiers during implementation? (RQ1) (2) How effective is our proposed method in modeling SOFL-to-Java traceability links in the situation of inconsistent identifiers during implementation? (RQ2) We also make a hypothesis that the implemented Java program implements all the specification and so we do not need to consider the measurement like recall, F1 score.

Table 4 illustrates the results for RQ1 and RQ2. The result shows that our method could achieve the precision of 94.4% and finally attain the precision of 97.2% by using type grouping traceability link optimization strategy in the situation of consistent identifiers. The performance of our proposed method is approaching the accuracy of naming conventions techniques which has the highest accuracy. Besides, it is still effective with the final accuracy of 88.8% in dealing with the situation of inconsistent identifiers. This is because proposed method makes use of more dimensional attributes' information. The 88.8% accuracy is usually better than the latent semantic indexing (LSI) [12] from existing information retrieval-based techniques which is one of the most common techniques for establishing traceability links.

The main threat comes from the following aspects. One is the scale of the program used in our experiment. It is not big enough to fully demonstrate the effectiveness of proposed method. However, the experimental results support our proposed method and proved that our method is more powerful than existing two methods in such a small-scale program and makes us believe that it will still perform better than them in a larger program. Besides, the applicability of proposed method will be restricted as currently we manually extract the attributes which may introduce some faults in extracted attribute sets and have not implemented the automatic various dimensional attributes extraction tool.

Conclusion

In this paper, we have presented an approach for establishing multilevel SOFL-to-Java traceability links for both completed programs. The proposed method for SOFL-to-Java traceability link could be extended to model relationships between formal specification artefacts and code artifacts. It enhances the existing establishment techniques by combining an ensemble of new and existing measurement dimensions including semantics, structure, function and relationships. An evaluation of our approach shows that it gives an accurate and fine-grained view of the relationships between the SOFL specification and Java code artefacts. This work mainly serves for automatic specification-based program inspection and software construction monitoring realization in human-machine pair programming to detect requirements-related fault and ensure software quality.

Reference

[1] Liu, S.: Formal engineering for industrial software development: Using the SOFL method. Springer Science & Business Media, 2013.
[2] Liu, S., Chen, Y., Nagoya, F. and McDermid, J.A.: Formal specification-based inspection for verification of programs. IEEE Transactions on software engineering, 38(5), pp.1100-1122 (2012).
[3] Liu, S.: Software Construction Monitoring and Predicting for Human-Machine Pair Programming. In International Workshop on Structured Object-Oriented Formal Language and Method, pp. 3-20, Springer, Cham (2018).
[4] Van Rompaey, B. and Demeyer, S.: Establishing traceability links between unit test cases and units under test. In 13th European Conference on Software Maintenance and Reengineering, pp. 209-218, IEEE (2009).
[5] Liu, S. and Nakajima, S.: Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. IEEE Transactions on Software Engineering (2020).
[6] Hayes, I.: VDM and Z: A comparative case study. Formal Aspects of Computing, 4(1), pp.76-99 (1992).
[7] Liu, S.: A survey on the use of SOFL based on four projects. Technical Report HCIS-2004–01, CIS (2004).
[8] Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N. and Mezini, M.: MUBench: A benchmark for API-misuse detectors. In Proceedings of the 13th International Conference on Mining Software Repositories, pp. 464-467 (2016).
[9] Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N. and Mezini, M.: MUBench: A benchmark for API-misuse detectors. In Proceedings of the 13th International Conference on Mining Software Repositories, pp. 464-467 (2016).
[10] Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N. and Mezini, M.: MUBench: A benchmark for API-misuse detectors. In Proceedings of the 13th International Conference on Mining Software Repositories, pp. 464-467 (2016).
[11] Li, Y., Tan, T., Møller, A. and Smaragdakis, Y.: A Principled Approach to Selective Context Sensitivity for Pointer Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), 42(2), pp.1-40 (2020).
[12] Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K. and Harshman, R.: Indexing by latent semantic analysis. Journal of the American society for information science, 41(6), pp.391-407 (1990).
[13] White, R., Krinke, J. and Tan, R.: Establishing multilevel test-to-code traceability links. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 861-872 (2020).
[14] Liu, S., "A Case Study of Modeling an ATM Using SOFL", technical report, 2013.

Acknowledgments The research was funded by China Scholarship Council (CSC No. 202108050145) and supported by ROIS NII Open Collaborative Research 2021-(21FS02).