TRACEABILITY MEASUREMENT BETWEEN A DESIGN MODEL AND ITS SOURCE CODE

Hiroki Itoh†, Hiroyuki Tanabe‡, Rieko Namiki‡, Hironori Washizaki† and Yoshiaki Fukazawa†
† Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
email: hitoh@fuka.info.waseda.ac.jp, {washizaki, fukazawa}@waseda.jp
‡ Osaka Gas Information System Research Institute CO., LTD.
Tokyo, Japan
email: {Tanabe_Hiroyuki, Namiki_Rieko}@ogis-ri.co.jp

ABSTRACT

Although researchers have recently investigated how to use and preserve traceability because it is an important issue for software maintainability, the degree of traceability is difficult to recognize objectively and precisely even if traceability links are recovered. Herein we propose a semi-automatic approach to measure the traceability between a design model and its source code via the Goal-Question-Metric approach. The original algorithm which maps the elements of design and implementation is also proposed for accurate measurements. We discuss performance of the mapping algorithm and usage of measurement results. The results suggest that our approach may elucidate the condition for traceability and aid in traceability maintenance.

KEY WORDS

traceability, GQM approach, metrics, mapping, software maintenance

1 Introduction

Traceability, which is the degree a relationship can be established between two or more products of the development process [1], is crucial for software maintainability. Researchers have investigated utility of traceability in model driven development. One benefit of traceability is change impact analysis; Hammad et al. have presented a system to detect changes in the source code and to translate them into design changes [2]. Olsen et al. have proposed coverage and orphan analyses, which can effectively verify the adequacy of documents [3]. Furthermore, other usages of traceability have been reported in the survey of Winkler et al. [4] (e.g. supporting design decisions, understanding artifacts, and reusing software).

To take advantage of the benefits described above, traceability must be identified. Although there are several techniques to visualize traceability links, it remains difficult to objectively comprehend the level of traceability. Due to this hurdle, traceability may not be recognized correctly, leading to problems with maintenance such as inadequate modification of software artifacts and determination of reusing design components. Hence, both visualization and an indicator are required to properly understand it.

We propose a semi-automatic approach to measure traceability between a design model described in UML and its source code written in an object-oriented program language. Our numeral results enable developers and evaluators objectively to recognize traceability. Then developers can confirm the validity of the documents and determine how to treat the documents with regards to maintenance. To measure traceability between a design model and its source code, traceability links must be created because current software development rarely maintains these links. Hence, we also present an original mapping algorithm and discuss its effectiveness.

This paper is structured as follows. Section 2 describes the motivation and the total image of our approach. Section 3 details the mapping procedure between design and implementation, and Section 4 measures the traceability. Section 6 discusses our methods, while Section 7 presents related works. Finally, Section 8 concludes the paper.

2 Overview of Our Approach

2.1 Motivation

When software needs to be modified, developers will consult its design model to specify the locations of the change. However, if traceability is not established, it becomes complicated to confirm which elements will be impacted. Thus, a care of traceability can be significant even during an activity of software maintenance.

Figure 1 represents that several differences are detected between a design model and source code. When maintainers plan to add a new function to the system, they have some options to handle the problem. For example, they may remove the divergence by changing

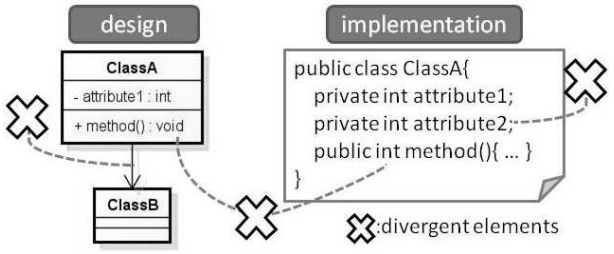

Figure 1. Example of Divergence between Design and Implementation

design information before maintenance. They may recover the design model by reverse engineering, considering that the huge divergence occurs. Additionally, they may decide to suspend the remediation temporally. To determine the action fitting for the actual situation, a whole viewpoint is required besides information about different elements.

On the other hand, recognition of traceability tends to be subjective as demonstrated by the fact that divergence visualization itself cannot provide an objective viewpoint. If members of a project cannot reach a consensus about the condition of traceability, then they cannot control it and make decisions for maintenance. Additionally, divergence visualization is not applicable to a design model in the early stage because the design structure may be changed (as illustrated in 3.1). As a result, it is difficult to understand validity of a design model.

As outlined above, employing only visualization is insufficient to determine the direction of software maintenance and obtain the common recognition about traceability. Therefore, we propose a method of quantitative traceability measurement for objective recognition or comprehensive understanding along with visualization.

2.2 Process

Figure 2 depicts our overall approach. The system input requires two artifacts: a design model written by UML and its object-oriented source code. Our process consists of the following five steps and runs semi-automatically.

Step 1: Extract the structure of the design model and its source code
Structural information is required to analyze traceability. We can conclude the step by using existing tools. Some modeling tools can provide a way to access elements of a design model and a reverse engineering tool can extract the static structure of the source code. Taking advantage of these functions, design and implementation elements can be compared.

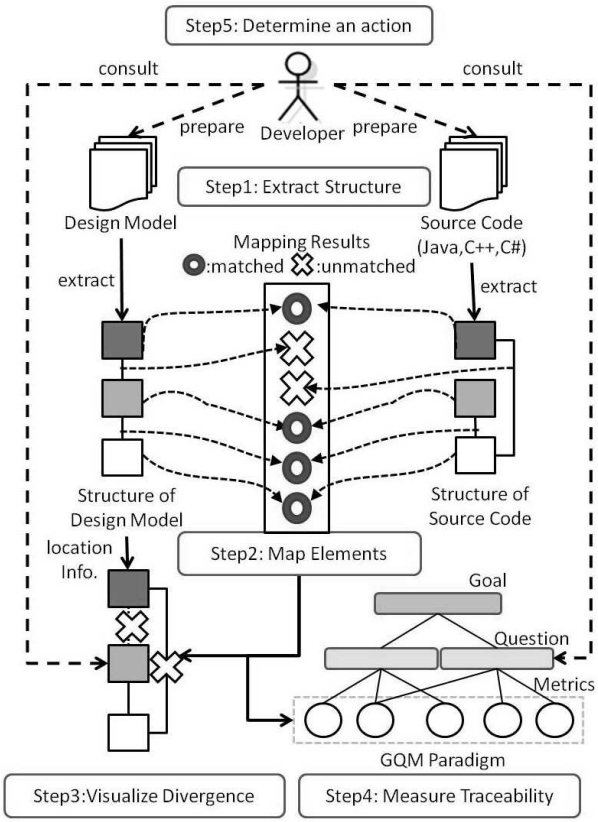

Figure 2. Total Image of Our Approach

Step 2: Map the elements between a the design model and its source code
The second step evaluates which design elements correspond to that of the source code. Our mapping algorithm consists of four mapping rules and creates traceability links semi-automatically. Section 3 describes the algorithm in detail.

Step 3: Visualize divergence
Of course, divergence information should be provided to modify the design and its source code. Several papers have proposed various methods to visualize the differences between two models [6] [10] [11]. We adopt a model that visualizes three types of differences with color or stereotype. The definitions of the differences are:

- Add: an element in the source code that is not described in a design model
- Remove: an element in a design model that is not described in the source code
- Modify: an element described in both a design model and the source code with varying content

Step 4: Measure traceability
We define two paradigms based on the GQM approach [9] for the measurement, which treat traceability from system and class perspectives. Section 4 provides the details.

Step 5: Determine an action for maintenance
Finally, developers determine how to remove differences using the results of divergence visualization and traceability measurement. We offer guides for an action toward maintenance in Section 5.

3 Mapping Elements between a Design Model and Source Code

Currently software development makes few attempts to establish the traceability links, which are required to analyze the degree of traceability between a design model and its source code. However, the mapping algorithm cannot be a simple name-matching method because the structure of the source code may change as the design model evolves from the early phase. Sections 3.1 and 3.2 explain obstacles in employing an early design model and the details of our mapping algorithm, respectively.

3.1 Difficulty of Mapping a Design Model in the Early Stage

Mapping becomes challenging when the source code undergoes structural changes without violating the software intentions. For example, after refactoring, new classes may appear by class divisions, and these additional items are not related due to the unavailability of their identifiers for matching.

Figure 3 shows an example of mapping elements between a design model and its source code where refactoring induces a structural change in the source code. If a simple algorithm, which searches for matching of class names, is applied, the class *Person* in the design is only mapped to the class with the same name in the implementation. Consequently, the class *TelephoneNumber* in the source code will not be linked with *Person* in the design. Therefore, a new algorithm must be prepared to address structural changes in the source code, which cause an incorrect measurement.

3.2 Mapping Algorithm

Our mapping algorithm consists of four rules. Each rule proposes candidates for a traceability link.

Rules 1 and 2 are simple mapping rules using the class name as an identifier, and provide comparatively precise results. Definitions are described as below.

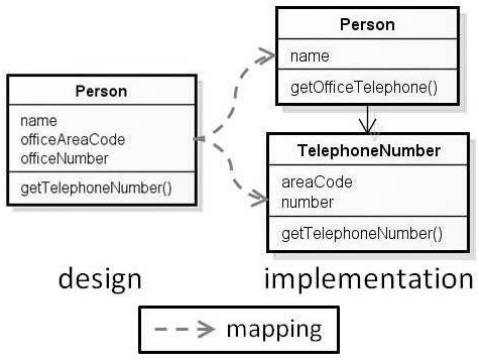

Figure 3. Extraction with Association Matching

Rule 1: Path Matching
Map the classes in the design model and source code whose names (considering namespace) are equal to each other.

Rule 2: Name Matching
Map the classes in the design model and its source code which are not mapped by Rule 1, if their names (not considering namespace) are equal to each other.

Unfortunately, Rules 1 and 2 are insufficient for proper mapping because the structure of a design model does not necessarily equate to that of the source code. Hence, Rules 3 and 4 are defined to resolve this issue.

Rule 3 create links in accordance with the value of the cosine similarity between class names. It assumes that additional classes created through refactoring have names similar to those of the original classes.

Rule 3: Cosine Similarity Matching
Maps classes (C_d in design model and C_i in source code) that satisfy two conditions:

1. C_i is not mapped with any classes upon applying Rules 1 or 2.
2. Cosine similarity between class names of C_d and C_i is equal or greater than 0.75.

Rule 4 links classes by summing textual information and structural characteristics. It aims to detect new classes, which are created by refactoring with class divisions introduced in [5], e.g., "Extract Class" and "Extract Superclass". Actually, Figure 3 is an example of refactoring "Extract Class", which is covered by Rule 4.

Rule 4: Extraction with Association / Generalization Matching
Maps classes (C_d in design model and C_i in source code) that satisfy three conditions:

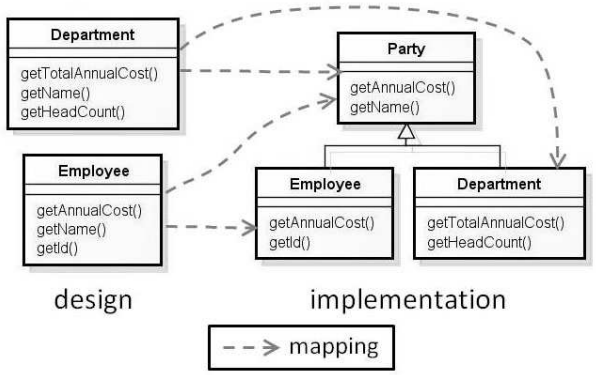

Figure 4. Extraction with Generalization Matching

1. C_i is not mapped with any classes upon applying Rules 1 or 2.

2. C_i has a generalization relationship or a navigable node of an association with a class in the source code which has already been mapped with C_d.

3. Vocabulary Coverage Metric (VCM) between C_d and C_i is equal or greater than 0.4.

VCM is defined as follows. V_d is a set of vocabulary included in a design class and V_i is one included in the source code.

$$VocabularyCoverageMetric(VCM) = \frac{|V_d \cap V_i|}{|V_i|}$$

Figure 4 illustrates refactoring of "Extract Superclass" and explains the application of Rule 4. According to the first and second conditions, the class *Department* in the design and the class *Party* in the implementation have a possibility to be mapped; the classes *Department* are linked and *Department* in the implementation is a child of *Party*. VCM of *Department* in the design with *Party* in the implementation is computed as below.

$$V_d = \{department, get, total, annual, cost, name, head, count\}$$
$$V_i = \{party, get, annual, cost, name\}$$
$$VCM = \frac{|V_d \cap V_i|}{|V_i|} = \frac{|\{get, annual, cost, name\}|}{|\{party, get, annual, cost, name\}|} = 0.80$$

As a result, Rule 4 maps these two classes which satisfy the three conditions.

After completing a search using the four mapping rules, users select the correct candidates. In this way, traceability links between a design model and its source code are recovered semi-automatically, which allows traceability to be precisely measured.

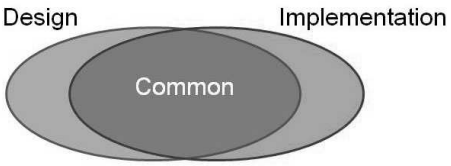

Figure 5. The Image of Element Classification

4 Measurement of Traceability

In our approach, traceability is measured from two viewpoints: system and class perspectives. Traceability from a system perspective verifies the number of common classes and relationships between a design model and its source code, whereas that from a class perspective evaluates whether design elements such as attributes or operations match the source code elements and vice versa.

To measure traceability, we adopt the GQM approach with two goals: "traceability from a system perspective" and "traceability from a class perspective". The following subsections detail the GQM paradigms of the two goals.

After metrics are measured, the scores of the goals and questions are obtained by calculating the average of its own holding metrics[1].

4.1 Traceability from a System Perspective

Target elements for measurement are the contents of a design model and its source code, and some of these belong to the both artifacts (see Figure 5). We define Questions 1 and 2 for traceability from a system perspective. Question 1 asks, "Are classes and relationships in the design model described in the source code?" That is, the question observes Common per Design. On the other hand, Question 2 asks, "Are classes and relationships in the source code described in the design model?", which observes Common per Implementation. Finally, after defining the questions, the metrics of this paradigm are derived. The results are useful when confirming the validity of a model or its source code. Table 1 shows the GQM paradigm of traceability from a system perspective.

4.2 Traceability from a Class Perspective

Similar to traceability from a system perspective, we define two questions and develop metrics for each question. Developers can check the values when considering reusing a component or reengineering. Table 2

[1] If the denominator of a metric becomes zero, a value of it will be "NaN" and is not used to calculate scores of the goal and question

Goal	Question	Metrics
S-G. Traceability from a system perspective	S-Q1. Are classes and relationships in the design model described in the source code?	S-M1. Ratio (%) that a class of a design model could be mapped to that of the source code
		S-M2. Ratio (%) that a class belongs to the same namespace between the design model and source code
		S-M3. Ratio (%) that a relationship of a design model could be mapped to that of the source code
	S-Q2. Are classes and relationships in the source code described in the design model?	S-M4. Ratio (%) that a class of the source code could be mapped to that of the design model
		S-M2. Ratio (%) that a class belongs to the same namespace between the design model and source code
		S-M5. Ratio (%) that a relationship of the source code could be mapped to that of the design model

Table 1. GQM Paradigm of Traceability from a System Perspective

Goal	Question	Metrics
C-G. Traceability from a class perspective	C-Q1. Are contents of a class in the design model described in source code?	C-M1. Ratio (%) that attributes of the design model could be mapped to those of the source code
		C-M2. Ratio (%) that operations of the design model could be mapped to those of the source code
		C-M3. Ratio (%) that mapped attributes and operations have common visibility, types, and arguments
		C-M4. Ratio (%) that relationships between classes of the design model could be mapped to those of the source code
	C-Q2. Are contents of a class in the source code described in the design model?	C-M5. Ratio (%) that attributes of the source code could be mapped to those of the design model
		C-M6. Ratio (%) that operations of the source code could be mapped to those of the design model
		C-M3. Ratio (%) that mapped attributes and operations have common visibility, types, and arguments
		C-M7. Ratio (%) that relationships between classes of the source code could be mapped to those of the design model

Table 2. GQM Paradigm of Traceability from a Class Perspective

shows the GQM paradigm of traceability from a class perspective.

5 Determine an Action for Maintenance

		Score of S-Q2	
		high	low
Score of S-Q1	high	I. Good Traceability	II. Consider modification of the divergence about the added classes
	low	III. Consider discarding and restructuring the design model	

Table 3. An action from the Scores of S-Q1 and S-Q2

		Score of C-Q2	
		high	low
Score of C-Q1	high	i. Good Traceability	ii. Consider modification or refactoring of the model and source code
	low	iii. Review the validity of the class	

Table 4. An action from the Scores of C-Q1 and C-Q2

This section indicates how to determine a direction for maintenance from the results of traceability measurement. To accomplish the purpose, the question scores play an important role.

Traceability from a system perspective represents how software artifacts are validated. Table 3 shows the relationship between the score of question 1 (S-Q1) and 2 (S-Q2). An S-Q1 score will be high if correct software development was conducted. Conversely, if a design model has the low S-Q1 score, a manager of the project should consider discarding and restructuring this because it indicates that the source code has not succeeded the intentions in the design phase. A score of S-Q2 will be lower as new functions are added to the system. Thus, the model and the source code with a poor S-Q2 score should be modified if detected additional functions violate the design intentions.

Traceability from a class perspective suggests whether the information about a class is up-to-date. Table 4 shows an action for remedying the divergence about classes. It is effective for modification to prioritize classes based on those traceability scores and significance in the system.

As above stated, developers can improve trace-

ability by consulting the question scores. The application examples are presented in the next section.

6 Discussions

We use two experiments to validate our approach; the first inspects performance of our mapping method and the second reveals usage of traceability measurement through the interviews with the developers. To investigate these issues, eight pairs of design models and source codes are examined. Table 5 illustrates the scale of these projects.

Proj.	Design Model		Source Code		
	Stage	#Classes	Lang.	#Classes	LOC
A	Early Stage	31	C++	54	2703
B	Early Stage	23	C++	29	2148
C	Late Stage	22	C++	26	1497
D	Late Stage	31	C++	32	2239
E	Early Stage	31	Java	139	8399
F	Late Stage	25	Java	28	3554
G	Early Stage	11	Java	13	319
H	Early Stage	19	Java	23	478

Table 5. Eight Projects for the Case Study

6.1 Performance of Our Mapping Method

Herein, we describe the performance results using our mapping approach. Table 6 represents the number of total and correct candidates against each mapping rule. In the table, "p" denotes mappings proposed by the approach, "c" corresponds to correct links found by the approach, and "c(manual)" indicates correct links extracted via a human decision.

Rules 1 and 2 can map elements for almost all candidates in all projects; there is one mistake in 199 candidates. Hence, Rules 1 and 2 play a role in precisely bridging a design model and its source code. In contrast, Rules 3 and 4 create a few candidates for mapping in some projects; although few candidates are found in Projects F and G, many are detected in the early stage of the design model for Projects A and E.

Actually, Table 4 indicates that there are the two doubtful precision values of Rules 3 and 4. One is the value of precision for Rule 3 in Project E, which is caused by the emergence of many classes with similar names. The other is that for Rule 4 in Project A, which occurs because the rule regards classes created by new functions as transformations produced by refactoring. The results indicate that Rules 3 and 4 do not provide as consistent results as Rules 1 and 2.

However, the value of recall is more important than that of precision because our approach is semi-automatic. Even if a faulty traceability link is reported, users can dismiss it. On the other hand, the addition of a new link is a time-consuming process. Thus, recall, which reflects how much manual recovery is required, has precedence over precision. In our algorithm, Rules 3 and 4 aim at improving a recall value by combining these rules and overcoming structural changes.

The experimental results suggest that a user only needs to add a few links per project after applying our algorithm, indicating that a little work is sufficient to initiate the measurement.

6.2 Usage of the Results for Maintenance

This section presents two examples that apply our approach to traceability maintenance. We conducted measurements for Projects C and D and deliberated the implications of the results. After that, real software development situations are confirmed via interviews with the developers who understand the condition of traceability well. Here, we present the results of the comparison between the measurements and the perception of developers about traceability.

First, we illustrate the inspection results of traceability measurement for Project C. The score of traceability from a system perspective stands at a high level (see Figure 6). With regard to the class perspective, most goal scores are high as seen in Figure 7, whereas two classes have problems (their scores are 47.0 and 52.4). Thus, we judged that the design model keeps valid (I in Table 3) and only a few classes which have the poor scores require to be modified for the next action (ii in Table 4). After inspection, we interviewed the developer of Project C about the situation of traceability. Consequently, it is apparent that the members had placed great importance to traceability through the development and revised information every time the structure changed. The developer also indicated that the two problem classes have relatively low traceability and plans to address them. The facts are consistent with the measurement results, and confirm that our approach can speculate the condition of traceability and an appropriate action in Project C.

The score of traceability from a system perspective for Project D was the highest of eight projects, whereas the project has many classes with the poor goal scores from a class perspective (see Figure 6 and 7) which are accompanied with an unsatisfactory C-Q2. Thus, we evaluated that it is better to keep the structure and modify the class members based on the notion described in Section 5. Similar to Project C, we asked the developer of Project D about the adequacy of the results. The results got a positive response regarding the system perspective and the decision for the divergence modification, whereas those of the class perspective were negative. We found that this paradox is due to omissions of elements in the design model. If attributes or operations are omitted from the class de-

Project	Total				
	#c(manual)	#p	#c	precision	recall
A	43	49	40	81.6%	93.0%
B	16	16	16	100.0%	100.0%
C	21	22	21	95.5%	100.0%
D	30	31	30	96.8%	100.0%
E	82	91	73	80.2%	89.0%
F	24	25	24	96.0%	100.0%
G	13	11	11	100.0%	84.6%
H	16	16	16	100.0%	100.0%

Project	Rule 1			Rule 2			Rule 3			Rule 4		
	#p	#c	precision	#p	#c	precision	#p	#c	precision	#p	#c	precision
A	28	28	100.0%	1	1	100.0%	10	10	100.0%	9	1	11.1%
B	0	0	—	14	14	100.0%	2	2	100.0%	0	0	—
C	21	21	100.0%	0	0	—	1	0	0.0%	0	0	—
D	30	30	100.0%	0	0	—	0	0	—	1	0	0.0%
E	0	0	—	55	55	100.0%	29	14	48.3%	19	16	84.2%
F	0	0	—	24	23	95.8%	1	1	100.0%	0	0	—
G	10	10	100.0%	0	0	—	1	1	100.0%	1	1	100.0%
H	16	16	100.0%	0	0	—	0	0	—	0	0	—

Table 6. Performance of Our Mapping Algorithm

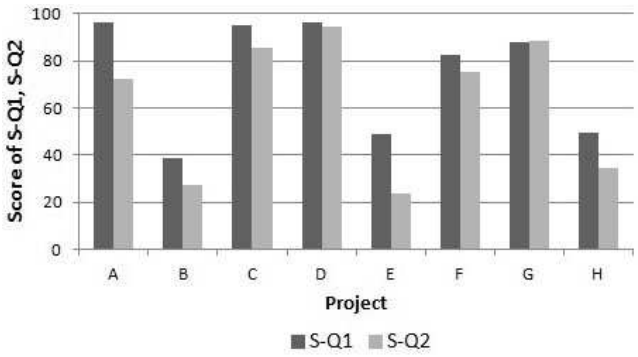

Figure 6. The Scores of S-Q1 and S-Q2

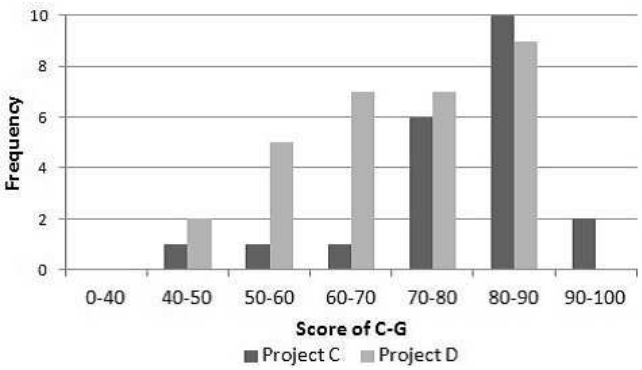

Figure 7. The Scores of C-G

scriptions, they will be regarded as added elements in source code, which results in the poor scores.

We confirmed that the results are helpful to speculate the condition of traceability even if the system is unfamiliar to an evaluator. However, omission of elements should be noticed because it may prevent from recognizing traceability correctly.

7 Related Works

There exist researches proposing the differencing algorithm between models, such as UMLDiff [10] and SiDiff [11]. UMLDiff compares the models created by extracting the structure of version-controlled object-oriented source code, whereas SiDiff detects differences using an original weight algorithm with similarity calculation. Although these approaches are helpful to compare the model structures of consecutive versions, it is insufficient simply to compare elements constituting the source code structure. When evaluating traceability between a design and its implementation, structural changes must also be considered.

One hurdle in traceability research is recovering traceability links. Antoniol et al. proposed a method to establish traceability links from the requirement to the source code based on information retrieval (IR) technique [7]. Gethers et al. integrates several standalone IR methods, aiming to improve previous recovery approaches [12]. In addition, the technique of Zhang et al. has been applied to ontology to link software artifacts semantically [13].

The approaches showed in this section can recover traceability links with their original methods. However, few works discuss traceability specializing in a design and its source code. Antoniol et al. tackled the issue in [14] but it is rather old. Moreover, as mentioned before, links themselves cannot be a director for an exact action, which creates a need to provide

quantitative indicator.

8 Conclusion

Herein we propose an approach to measure the level of traceability between a design model and its source code. To achieve this object, two methods are presented. The first is a mapping algorithm, which bridges a design model and its source code. This semi-automatic approach presents candidates of traceability links, and the user extracts the correct ones. The second is the GQM paradigms to measure traceability, which are defined from two perspectives, system and class perspectives. Finally, the paper verifies the performance of our mapping algorithm and employs the measurement results in traceability maintenance.

For future works, our approach could be improved by applying previous research as described below. The technique of Eaddy et al. [8] would enable our mapping method to be more precise because it uses not only IR but also dynamic analysis and program analysis to trace the requirement to the source code. Additionally, it should be a significant activity for traceability to be analyzed from the aspect of behavior as well as the static structure. Several methodologies have been proposed to extract the behavior of system and generate UML dynamic diagrams (such as sequence diagram [15] and collaboration diagram [16]), which prepares for comparing elements before measurement. Egyed et al. have reported an automatic technique to generate trace information and have applied it to software documents, including a state chart diagram in its case study [17]. Traceability analysis focusing on a behavioral aspect would provide furthermore information to verify the validity of artifacts.

References

[1] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, (New York, IEEE, 1990).

[2] M. Hammad, M. L. Collard and J. I. Maletic, Automatically identifying changes that impact code-to-design traceability during evolution, *ICPC '09*, 2009, 35-64.

[3] G. K. Olsen and J. Oldevik, Scenarios of Traceability in Model to Text Transformations, *ECMDA-FA '07*, 2007, 144-156.

[4] S. Winkler and J. V. Pilgrim, A survey of traceability in requirements engineering and model-driven development, *Software and Systems Modeling*, 9(4), 2010, 529-565.

[5] M. Fowler, *Refactoring: Improving The Design Of Existing Code*, (Boston, Addison-Wesley, 1999).

[6] G. C. Murphy, D. Notkin and K. J. Sullivan, Software Reflexion Models: Bridging the Gap between Design and Implementation, *IEEE Transactions on Software Engineering*, 27(4), 2001, 364-380.

[7] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia and E. Merlo, Recovering Traceability Links between Code and Documentation, *IEEE Transactions on Software Engineering*, 28(10), 2002, 970-983.

[8] M. Eaddy, A. V. Aho, G. Antoniol and Y. G. Guéhéneuc, CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis, *ICPC '08*, 2008, 53-62.

[9] V. R. Basili, G. Cladiera and H. D. Rombach, The goal question metric approach, in J. J. Marciniak (Ed.) *Encyclopedia of Software Engineering*, (New York, John Wiley & Sons, Inc., 1994), 528-532.

[10] Z. Xing and E. Stroulia, UMLDiff: An Algorithm for Object-Oriented Design Differencing, *ASE '05*, 2005, 54-65.

[11] C. Treude, S. Berlik, S. Wenzel and U. Kelter, Difference computation of large models, *ESEC-FSE '07*, 2007, 295-304.

[12] M. Gethers, R. Oliveto, D.Poshyvanyk and A. D. Lucia, On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery, *ICSM '11*, 2011, 133-142.

[13] Y. Zhang, R. Witte, J. Rilling and V. Haarslev, Ontological approach for the semantic recovery of traceability links between software artefacts, *IET Software*, 2(3), 2008, 185-203.

[14] G. Antoniol, B. Caprile, A. Potrich and P. Tonella, Design-code traceability for object-oriented systems, *Annals of Software Engineering*, 9(1-2), 2000, 35-58.

[15] L. C. Briand, Y. Labiche and J. Leduc, Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software, *IEEE Transactions on Software Engineering*, 2006, 642-663.

[16] R. Kollmann and M. Gogolla, Capturing dynamic program behavior with UML collaboration diagrams, *CSMR '01*, 2001, 58-67.

[17] A. Egyed and P. Grunbacher, Automating requirements traceability: Beyond the record & replay paradigm, *ASE '02*, 2002, 163-171.