TOWARDS AN INTEGRATED SUPPORT FOR TRACEABILITY OF QUALITY REQUIREMENTS USING SOFTWARE SPECTRUM ANALYSIS

Haruhiko Kaiya, Kasuhisa Amemiya, Yuutarou Shimizu and Kenji Kaijiri
Shinshu University, 4-17-1 Wakasato, Nagano City, 380-8553, Japan

Keywords: Traceability, Quality characteristics, Requirements specification, Source codes.

Abstract: In actual software development, software engineering artifacts such as requirements documents, design diagrams and source codes can be updated and changed respectively and simultaneously, and they should be consistent with each other. However, maintaining such consistency is one of the difficult problems especially for software quality features such as usability, reliability, efficiency and so on. Managing traceability among such artifacts is one of solutions, and several types of techniques for traceability have been already proposed. However, there is no silver bullet for solving the problem. In this paper, we categorized current techniques for managing traceability into three types: traceability links, central model and projection traceability. We then discuss how to cope with these types of techniques for managing traceability for software quality features. Because projection traceability seems to be suitable for quality features and there are few implementations of projection traceability, we implement a method based on projection traceability using spectrum analysis for software quality. We also apply the method to an example to confirm the usefulness of projection traceability as well as traceability links and central model.

1 INTRODUCTION

During a software development, we cannot directly write source codes that satisfy large and complex requirements because of our cognitive and intellectual limitation. Therefore, we have to stepwise refine abstract requirements into more concrete artifacts. As a result, a lot of artifacts are developed such as a requirements document, design diagrams, test case documents, source codes and so on during the development. In addition, an abstraction level of each artifact is different from a level of another artifact. Ideally, such artifacts are developed and completed in turn along with refining one artifact to another, i.e., based on waterfall model. However, each artifact is updated and changed simultaneously in fact. For example, some codes are modified and then corresponding design documents are updated.

We thus have to manage the relationships between parts in an artifact and those in another for avoiding inconsistencies between two different artifacts. For example, inconsistencies can occur if we modified codes but forgot updating corresponding design documents. Traceability is one of approaches for avoiding such inconsistencies, and a lot of techniques have been already proposed. However, most of them focus on functional requirements.

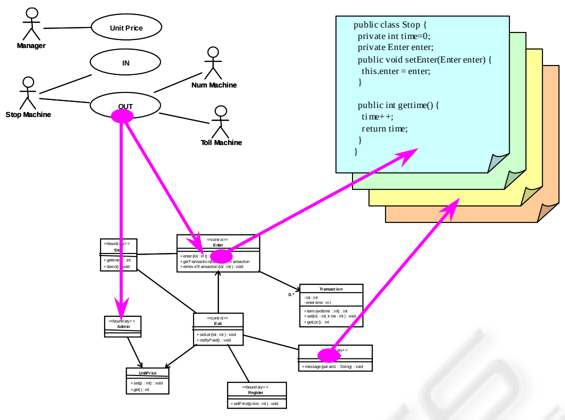
Recently, quality requirements are focused as well as functional requirements because completion among different systems usually depends on the differences of quality requirements. For example, most stakeholders prefer to more usable system. Existing traceability techniques can be of course applied to quality requirements, but they are sometimes inefficient. The main reason is that most quality requirements are scattered features over a software engineering artifact. For example, usability is realized in a design based on harmony among a lot of interaction points such as GUI in a system. Other types of quality requirements such as efficiency, reliability and so on are also realized in the same way. We thus have to have another type of traceability techniques in addition to existing current techniques. In this paper, we introduce a traceability approach called “projection traceability” for analyzing scattered features like quality requirements. The idea of projection traceability is as follows. Instead of explicit links between artifacts, we first make each artifact to project its quality features on a model in the same way that a tree projects its shadow on the ground. We then compare one "shadow" of an artifact to another "shadow" of another artifact to find inconsistencies between these artifacts. We call such a "shadow" of a software engineering artifact as QSM (Quality Specific Model) in this paper. If some inconsistencies about some quality features are found through such comparison, we may put emphasis on the features and use existing traceability techniques only for the parts of the artifacts where the features are related.

The rest of this paper is organized as follows. In the next section, we review current techniques for traceability and categorize them into two types. We also review techniques for measuring quality requirements. To complement problems of current traceability techniques, we propose another type of traceability techniques called "projection traceability" in section 3. In section 4, we concretely implemented a method that can be categorized as projection traceability based on spectrum analysis for quality requirements. An example for applying the method is shown in section 5 to confirm the usefulness of the method. Finally, we summarize our current results and show the future issues.

2 RELATED WORKS

Software quality requirements are widely focused in the field of software, and we can find its special issue in 2008 IEEE Software (Blaine and Cleland-Huang, 2008). In the issue, importance and challenges about software quality requirements were summarized, and one of challenges is measurement and traceability for software quality requirements. In this section, we briefly review researches about measurement and traceability for software quality requirements to clarify the importance of software quality requirements. We use NFR (Non-Functional Requirements) as a synonym of quality requirements even though NFR contains more things than quality requirements (Blaine and Cleland-Huang, 2008), (Glinz, 2008).

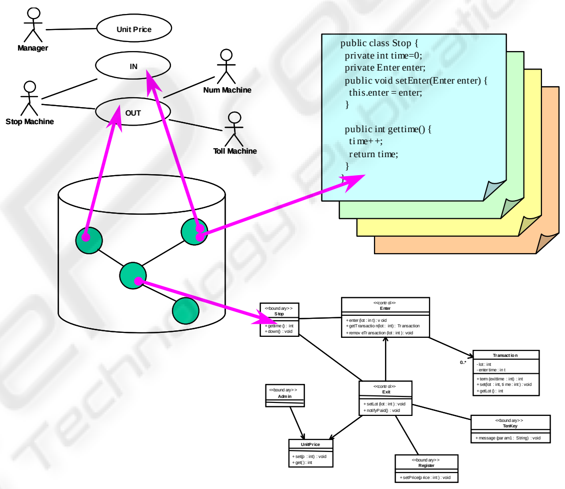
First, we focus on *traceability links* among different kinds of software engineering artifacts. Managing explicit links among different artifacts is normal idea. For example, links between a section in a requirements document and classes and packages in design documents help us to find change impacts on design. Figure 1 shows a general example using traceability links. However, maintaining such links takes a lot of efforts in general. To mitigate such effort, several kinds of ideas are proposed. Lucia et al. used an information retrieval technique to mange such links (Lucia et al., 2009). Mandelin et al. used a probability model



```
public class Stop {
  private int time=0;
  private Enter enter;
  public void setEnter(Enter enter) {
   this.enter = enter;
  }

  public int gettime() {
   time++;
   return time;
  }
}
```

Figure 1: Traceability links.



```
public class Stop {
  private int time=0;
  private Enter enter;
  public void setEnter(Enter enter) {
   this.enter = enter;
  }

  public int gettime() {
   time++;
   return time;
  }
}
```

Figure 2: Central model.

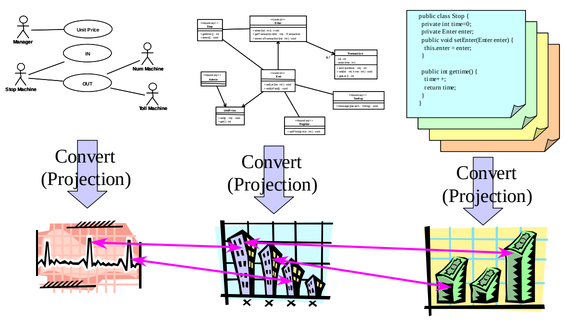


Figure 3: Projection traceability.

to do that (Mandelin et al., 2005). Ratanotayanon et al. used differences between different versions of artifacts to manage traceability links efficiently (Ratanotayanon et al., 2009). Lopez et al. used techniques of NLP (Natural Language Processing) and ML (Machine Learning) to trace quality requirements to architecture (Gokyer et al., 2008).

Second, we focus on *a central model* shared by software engineering artifacts. By using a central model, we can easily trace an artifact to another via the model. Figure 2 also shows a general example using a central model for traceability. Jane et al. proposed a method called "Goal Centric Traceability" for quality requirements (Cleland-Huang, 2005), (Cleland-Huang et al., 2008). In the method, a goal model plays the role of a central model. Thesaurus and ontology are popular notations for a central model. Daneva et al. proposed an ontology for NFR (Kassab et al., 2009), (Kassab et al., 2008). However, how to make links between ontology and software artifacts was not mentioned. Saeki et al. used a domain ontology for traceability between documents and source codes (Yoshikawa et al., 2009).

Finally, we briefly review researches about measurement on quality requirements. One of the famous catalog for software quality requirements is ISO9126 standard (International Standard ISO/IEC 9126-1, 2001), and the standard contains about 20 subcharacteristics such as accuracy, reliability and so on. Washizaki et al. provided measurement methods using usual metrics on source codes and design diagrams such as LOC (Lines of Codes) and CC (Cyclomatic Complexity) for each subcharacteristic (Washizaki et al., 2008). Jane et al. proposed a method to detect and to categorize NFR contained in a document using IR (Information Retrieval) and NLP (Natural Language Processing) techniques. (Cleland-Huang et al., 2006). To count and normalize the number of NFR in a document, we can visualize a distribution of NFR. Kaiya et al. proposed a technique to summarize such distribution and to visualize it based on a metaphor about spectrum analysis in optics (Kaiya et al., 2009). They used the technique to identify domain specific commonality by directly comparing one spectrum of a system to another. These kinds of researches are useful for tracing quality requirements indirectly.

3 PROJECTION APPROACH FOR QUALITY REQUIREMENTS TRACEABILITY

As mentioned in the previous section, traceability link approach and central model approach are used actively and seem to be effective. However, they have several drawbacks respectively. Different kinds of approaches thus seem to mutually mitigate such drawbacks. Because both approaches focus on the units in software artifacts, it is not easy to handle scattered features in artifacts. Quality requirements are typical scattered features in software artifacts, thus another approach suitable for such scattered features will be helpful in addition to current two approaches.

In the same way as Figures 1 and 2, we depict a general example called *projection traceability* in Figure 3. According to projection traceability, each software engineering artifact is converted to (or projected on) some specific model respectively. In Figure 3, the specific model is written in bar charts. By comparing such specific models, we can easily recognize scattered features such as quality requirements. We never think projection traceability solves all problems in quality requirements traceability. Projection traceability can cope with another kinds of approaches such as traceablity links or central model. For example, we first use projection traceability to find what kind of quality features is inconsistent between requirements documents and design diagrams. We then use traceablity links to browse concrete units of artifacts such as requirements sentences and classes with inconsistency.

In projection traceability approach, how to project each artifact on the model is important. As mentioned in the previous section, we have already had some techniques for measuring quality requirements. We can use such techniques to construct *projectors* for each type of software engineering artifacts.

4 TRACEABILITY USING QUALITY SPECTRUM ANALYSIS

In this section, we concretely implement a method based on projection traceability approach in section 3 using spectrum analysis for software quality requirements (Kaiya et al., 2009).

4.1 Quality Spectrum Analysis

Spectrum analysis for quality requirements is one of measurement techniques to summarize the scattered quality features over a requirements document. The technique enables us to visualize quality characteristics in the same way as sound waves. We explain how to perform spectrum analysis for quality requirements by using an example in Figure 4. In the figure, there is a list of requirements and a list of quality characteristics. These two lists are inputs of the technique. An analyst then decides quality characteristics related to each requirement. For example, a requirement #3 is related to both resource efficiency and changeability.

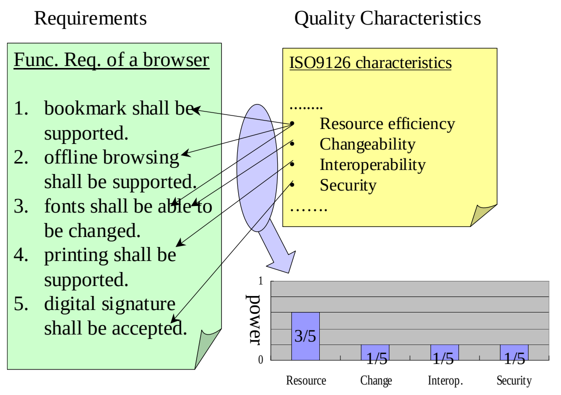


Figure 4: An example of spectrum analysis for requirements.

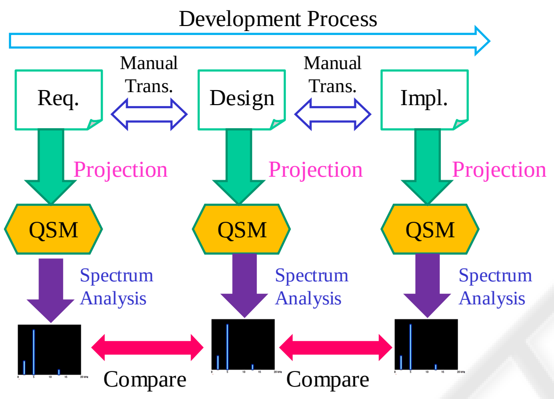


Figure 5: Projection traceability using spectrum analysis.

After that, the analyst counts the number of requirements related to each quality characteristic. For example, resource efficiency is related to 3 out of 5 requirements. The counted numbers of each quality characteristic lets us know how often each quality characteristic is mentioned in a requirements document, and we assume the number is the importance of each quality characteristic. We call the number as "power" of a quality feature such as resource efficiency according to a metaphor of waves. For example in this figure, resource efficiency is more important than others because resource efficiency is frequently mentioned in the list of requirements. We may give some weights to each requirement based on the priority among the requirements. We may also reformat a requirements document (Ncube et al., 2007) if the document is highly structured like i* or KAOS goal models.

4.2 Projection Traceability over a Development Process: An Implementation

Although the spectrum analysis for quality requirements is a simple but a powerful technique to visualize quality requirements over a requirements document, the technique is applied only to a list of sentences like documents. We find we can implement a concrete method of projection traceability mentioned in section 3 based on the spectrum analysis for quality requirements if the spectrum analysis can be applied to other types of software engineering artifacts such as design diagrams and source codes.

We implement projection traceability in Figure 3 by generalizing the idea of spectrum analysis in Figure 4. The outline of the method of projection traceability using the spectrum analysis is shown in Figure 5. Boxes at the top of the figure show software engineering artifacts such as requirements specification, design documents and source codes. During a software development process, each artifact is transformed into another if we regard the process as the transformation from abstract artifacts (e.g., requirements) to concrete artifacts (source codes). Such transformation is performed manually in general, but it is sometimes performed automatically thank to model-driven architecture. To establish projection traceability, we project each artifact into QSM (Quality Specific Model), and apply spectrum analysis to each QSM. QSM is a sub-structure of an artifact, and the sub-structure can specify quality features of the artifact. We expect such projection is achieved automatically or semi-automatically. How to achieve such projection is discussed in the rest of this section. After obtaining QSM of each artifact, we may simply apply spectrum analysis to each QSM, and compare a spectrum to another. If there are some differences between two spectra, there can be some inconsistencies among artifacts related to the spectra.

The major advantages of this method are as follows. First, traceability can be always checked even if each artifact can be maintained and changed independently. Second, the method does not restrict the ordering of software development. For example, we may change source codes and then update design document. Projection traceability merely tells trends of inconsistencies, thus it does not directly point out inconsistency parts in artifacts. It is an inherent drawback of projection traceability.

4.3 QSM for Requirements Specification

In the case of requirements, the original spectrum analysis mentioned in section 4.1 can be directly used, and QSM of requirements is the list of requirements sentences.

We have no fixed idea of QSM for design documents. However, we will use sequence diagrams and activity diagrams for QSM.

4.4 QSM for Source Codes

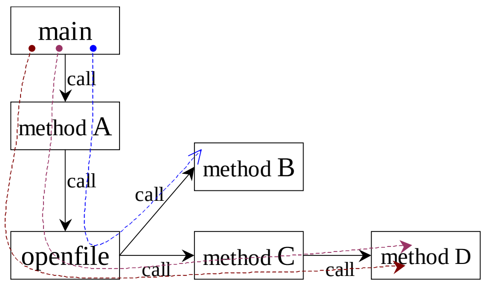


Figure 6: An example of path based weight on a call graph.

In the case of source codes, we focus on a call graph among methods or functions as a candidate of QSM. The reasons are as follows. First, a call graph can be generated automatically based on source codes. In the case of object-oriented programs, we cannot generate complete call graphs because of dynamic binding. However, approximate call graphs can be obtained. Second, a sequence of methods linked by call relationships (we simply call such a sequence as "*path*" in this paper) almost corresponds to a use case, thus the path corresponds to a requirement.

By using Figure 6, we explain how to obtain a quality spectrum from a call graph. In the figure, there is a simple call graph and it contains five methods except main method. We then make relationship between each method and quality characteristics in the same way as a requirement in Figure 4 according to terms in each method and its comments. For example, a method "openfile" may be related to usability if it includes GUI for opening files. Three paths can be identified in the call graph, and each method is contained in several paths. The number of paths on a method is used as a weight of the method, and such weights are used to calculate powers in a quality spectrum. Suppose "openfile" method is contained three paths, the method is related to usability, and no other methods are related to usability. A power of usability is then three.

5 EXAMPLE

To confirm the usefulness of a projection traceability method mentioned in section 4, we apply the method to an example. The main goal of this example is to confirm whether the method can help us to find inconsistencies of some quality requirements between requirements and source codes.

5.1 Overview of the Example

To achieve the main goal above, we prepared the following three software engineering artifacts.

- R: a requirements list of a system where some quality features are important.
- X: source codes for the system and the codes are consistent with R. In other words, all quality features are implemented in the codes X. Actually, codes X did not exist but we virtually developed codes X. We will explain this point in section 5.3.
- Y: source codes for the system and the codes are *not* consistent with R with respect to quality features. In other words, all quality features are not implemented in the codes Y.

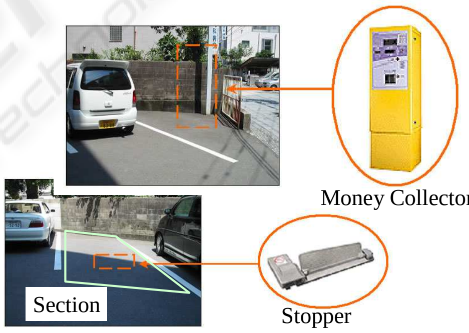


Figure 7: A car park.

We concretely used a system for controlling stoppers and a money collector in a car park as shown in Figure 7. There is a car park, and it has several sections for each car and one machine for collecting money. In each section, there is a stopper to fix a car until its payment is finished. The system controls the money collector and stoppers automatically.

In this system, following quality features are especially important because of each reason.

- Performance (Pe): Because more than one car can enter several sections simultaneously, the system should control several stoppers at the same time. The system is thus required enough performance.
- Usability (Us): The system interacts with a manager as well as hardware devices because the

manger set the unit price for example. The system is thus required usability.

- Reliability (Re): Because the system is a kind of embedded systems, reliability is important.
- Accuracy (Ac): Accuracy is also important because the system handles the amount of money.
- Changeability (Ch): We can easily assume changes about hardware devices. For example, the number of money collectors increases for better availability of payment. The system thus has to take changeability into account.

We simply abbreviate the name of each feature to its first two letters like "Pe" in the following part of this paper.

5.2 Requirements projected on QSM

We prepare a requirements list i.e., R, and the list contains the following ten requirements.

1. The system shall enable a manager to set a unit of price.
2. The system shall receive an ID of a stopper and the current time when the stopper raises. Note that each stopper raises when a car enters a section where the stopper is equipped.
3. The system shall receive a numerical number put into a money collector. The number specifies a section and its stopper which fixes a car.
4. The system shall know the amount of money (payment) when someone puts money into a money collector.
5. The system shall know whether a car is fixed or not in a section.
6. The system shall know the current time.
7. The system shall decide whether the payment is enough for parking of a car.
8. The system shall release a car if its payment is enough and a stopper fixes the car.
9. The system shall pay back all payment if the payment is not enough.
10. The system shall ignore a numerical number put via a money collector if no car is fixed at a section specified by the number.

According to the spectrum analysis mentioned in section 4.3, we obtained the relationships between each requirement and quality characteristics. For example, req#02 is related to performance and reliability because the requirement is about receiving data from stoppers. Because the system interacts hardware devices frequently, the number of relationships between requirements and reliability is relatively high. We then calculate and visualize their spectrum as shown in Figure 8. For example in the figure, the power of reliability (Re) is 0.8. This value implies reliability is so important in this system.

5.3 Codes X and Y projected on QSM

We developed source codes Y based on requirements R, but Y could not completely satisfy quality requirements in R. The codes contain 8 classes of Java and 31 methods. Concretely, Y could not satisfy the following quality requirements in R.

- Performance
- Changeability

Because we did not have actual hardware devices, the codes Y could run as a console application on general OS. Interactions between the system and devices were represented as standard inputs and outputs on the console. We applied spectrum analysis mentioned in section 4.4 to QSM of Y. Table 1 shows the data for quality spectrum of codes Y. As shown in the table, there are 31 methods in Y, and a method "Admin.inputPrice" is related to usability and accuracy because this method is about UI for setting unit of price for example. The last column in the table shows the number of paths on each method, thus scores in other columns are weighted with the number when powers of each quality characteristics are calculated. For example, the power of usability is four even when usability is related to only two methods. Note that each power is normalized with 46 (the total number of the last column) when we visualize the powers as a spectrum.

We did not actually write codes X that completely satisfy quality requirements R, but we put comments to each method in Y. Such comments specify how to improve a method to satisfy all quality requirements in R. We regard codes Y with such comments as codes X, and applied spectrum analysis mentioned in section 4.4 to QSM of X.

5.4 Discussion

Figure 8 shows spectra of requirements R, codes X and Y. We have following two expectations to these spectra.

1. We expect a spectrum of R is similar to another of X because X was virtually written to satisfy quality requirements R.
2. We expect a spectrum of R is different from another of Y with respect to performance and changeability.

Table 1: Data for spectrum of codes Y.

31 methods	Performance (Pe)	Usability (Us)	Reliability (Re)	Accuracy (Ac)	Changeability (Ch)	Num. of Paths on the method
Admin.inputPrice		1		1		1
Admin.setUnitPrice						1
Enter.enter			1			2
Enter.getTransaction						3
Enter.removeTransaction						1
Enter.setStop						1
Exit.notifyPaid						3
Exit.setEnter						1
Exit.setLot				1	1	6
Exit.setRegister						1
Exit.setStop						1
Exit.setTenKey						1
Exit.setUnitPrice						1
Main.getLotNumber						0
Main.main						0
ReadLn.readln						0
ReadLn.readnat						0
Register.setExit						1
Register.setPrice		1	1			3
Stop.carEntered			1			2
Stop.down			1			1
Stop.gettime			1			1
Stop.setEnter						1
TenKey.driverReturned			1			6
TenKey.message			1			2
TenKey.setExit						1
Transaction.getLot						1
Transaction.set					1	1
Transaction.term						1
UnitPrice.get						1
UnitPrice.set						1
powers (not normalized)	0	4	17	7	7	46

According to expectation 1, a spectrum of R is almost similar to another of X except reliability. As mention in earlier part in this section, the system did not control actual hardware devices but it merely run on a generic OS. Because reliability largely depends on interfaces between the system and hardware devices, we thus overlooked reliability related to statements or comments even in codes X.

According to expectation 2, a spectrum of R is different from another of Y with respect to reliability, performance and changeability. As mentioned in a paragraph above, the difference about reliability was caused because codes did not actually interact

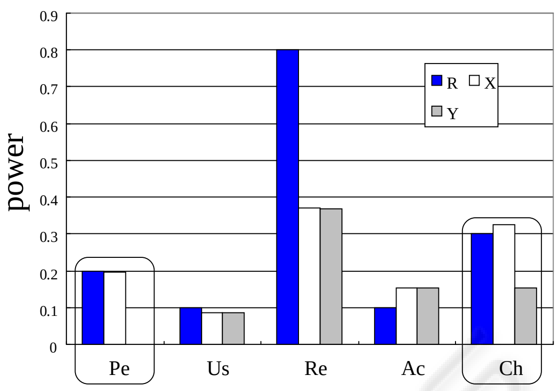


Figure 8: Spectra of requirements, codes X and Y.

with hardware devices. The differences about performance and changeability were intentionally embedded in codes Y, thus projection traceability based on spectrum analysis seems to work well. After finding such differences, we may use or establish traceability links or a central model to browse concrete parts of requirements and source codes.

6 CONCLUSIONS

In this paper, we first review what kinds of traceability techniques exist and discuss what kinds of characteristics each kind of the techniques has. We roughly categorized such techniques into two types: traceability links and central model. As a result, we regard we have to explore another type of techniques other than existing techniques because existing techniques are not good at analyzing scattered features in software engineering artifacts such as quality characteristics. We thus proposed a traceability approach called "projection traceability" based on a concept of a spectrum analysis for software quality requirements (Kaiya et al., 2009). Projection traceability helps us to find the tendency of inconsistencies among different types of artifacts such as a requirements document and source codes. After identifying tendency of such inconsistencies, existing techniques such as traceability links can be used efficiently. We finally apply a method of projection traceability to an embedded software system, and we confirmed the method could help us to find the tendency of quality inconsistencies to an extent.

In an actual software development, some types of requirements are added even after release of a system (Nakatani et al., 2008). Therefore, we never complete requirements, design and codes in turn. Projection traceability can be applied even in such a development because software engineering artifacts can be analyzed respectively.

In architecture, design and codes, quality features are implemented by using specific structures of modules, design patterns or algorithms. For example, MVC is a famous design pattern for high changeability. We do not yet focus on such points when we develop projectors for design diagrams and codes, thus we want to take them into account in the future. In addition, existing software components, e.g., libraries and application frameworks are used in most of all software systems, and such components off course give effects on whether quality requirements are satisfied or not. We also want to focus on how to handle impacts by reused components when we analyze traceability about quality requirements.

ACKNOWLEDGEMENTS

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (C), 20500032, 2010.

REFERENCES

Blaine, J. D. and Cleland-Huang, J. (2008). Software quality requirements: How to balance competing priorities. *IEEE Software*, 25(2):22–24.

Cleland-Huang, J. (2005). Toward Improved Traceability of Non-Functional Requirements. In *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 14–19.

Cleland-Huang, J., Marrero, W., and Berenbach, B. (2008). Goal-centric traceability: Using virtual plumblines to maintain critical systemic qualities. *IEEE Trans. Software Eng.*, 34(5):685–699.

Cleland-Huang, J., Settimi, R., Zou, X., and Solc, P. (2006). The detection and classification of non-functional requirements with application to early aspects. In *RE*, pages 36–45.

Glinz, M. (2008). A risk-based, value-oriented approach to quality requirements. *IEEE Software*, 25(2):34–41.

Gokyer, G., Cetin, S., Sener, C., and Yondem, M. T. (2008). Non-functional requirements to architectural concerns: Ml and nlp at crossroads. *Software Engineering Advances, International Conference on*, 0:400–406.

International Standard ISO/IEC 9126-1 (2001). Software engineering - Product quality - Part 1: Quality model.

Kaiya, H., Tanigawa, M., Suzuki, S., Sato, T., and Kaijiri, K. (2009). Spectrum Analysis for Quality Requirements by Using a Term-Characteristics. In *21th International Conference Advanced Information Systems Engineering (CAiSE 2009)*, pages 546–560, Amsterdam, The Netherlands. LNCS 5565.

Kassab, M., Daneva, M., and Ormandjieva, O. (2008). A meta-model for the assessment of non-functional requirement size. In *SEAA*, pages 411–418.

Kassab, M., Ormandjieva, O., and Daneva, M. (2009). An ontology based approach to non-functional requirements conceptualization. *Software Engineering Advances, International Conference on*, 0:299–308.

Lucia, A. D., Oliveto, R., and Tortora, G. (2009). Assessing ir-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 14(1):57–92.

Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61.

Nakatani, T., Hori, S., Ubayashi, N., Katamine, K., and Hashimoto, M. (2008). A case study: Requirements elicitation processes throughout a project. In *RE*, pages 241–246.

Ncube, C., Lockerbie, J., and Maiden, N. A. M. (2007). Automatically generating requirements from * models: Experiences with a complex airport operations system. In *REFSQ*, pages 33–47.

Ratanotayanon, S., Sim, S. E., and Raycraft, D. J. (2009). Cross-artifact traceability using lightweight links. In *TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 57–64, Washington, DC, USA. IEEE Computer Society.

Washizaki, H., Hiraguchi, H., and Fukazawa, Y. (2008). A metrics suite for measuring quality characteristics of javabeans components. In *PROFES*, pages 45–60. LNCS 5089.

Yoshikawa, T., Hayashi, S., and Saeki, M. (2009). Recovering traceability links between a simple natural language sentence and source code using domain ontologies. In *ICSM*, pages 551–554.