Building Traceability for Specification Evolution and Inspection

CAI, Weichen

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
8
(開始ページ / Start Page)
49
(終了ページ / End Page)
54
(発行年 / Year)
2013-03
(URL)
https://doi.org/10.15002/00009543

Building Traceability for Specification Evolution and Inspection

Weichen Cai
Graduate School of Computer and Information sciences
Tokyo 184-8584, Japan
Email: weichen.cai.jh@stu.hosei.ac.jp
Telephone: (080) 3014-8812

Abstract—Requirements traceability has been widely recognized as an indispensable support for software development activity, especially during recent years, with the scale of the software system becoming unprecedentedly larger and the time period of developing a software system becoming longer. This situation brings up more evolution in requirements specifications and makes requirements traceability more attractive. However, due to the informality of the specifications, the traceability of requirements during evolutions is unlikely to be built systematically and precisely. In this paper, we use SOFL specification to establish an effective approach to building requirements traceability systematically and automatically under certain conditions.

Index Terms—Requirements traceability, formal specification, documentation evolution

I. Introduction

Iterative approach for software development has been widely accepted and used in industrial practice for a long time [1]. Using an iterative approach always come with changes in the documents we create and have to maintain during the whole software development life-cycle. These changes in the documents are called documentation evolution. A great challenge in documentation evolution is how to sustain the consistency between different level documents.

Inconsistency caused by documentation evolution will happen between either homogeneous documents or heterogeneous documents Such inconsistency between documents will deteriorate the trust to these documents and make most of the documents less useful [2][3]. Documentation traceability is mainly focusing on solving the inconsistency during the document evolution. By creating links between different documents, the changes in each document are traceable. But the major obstacle we are facing is that most of the software documents are written in natural language while manually creating links will unwillingly cost additional time. Full automation of building a requirements traceability based on informal documents is obviously difficult.

Formal methods are likely to have more advantages in solving traceability problem [4] in an automatic manner, because the formal documents are organized in a well structured style. Of many existing formal methods, the SOFL formal engineering method has proved to be a practical and effective method for its three-step modeling approach [5]. Developing a software system using SOFL (Structure Object-oriented Formal Language), the starting point is to construct an informal specification that includes the requirements for functions, data resources, and constraints. To clarify the semantics of the descriptions, the informal specification is transformed into a semi-formal specification. The requirements for functions, data, and constraints in the semi-formal specification are organized in a well-structured style in terms of modules with several kinds of elements (process, function, type identifier, constant identifier, state variable, invariant). Finally, the semi-formal specification is transformed into a formal design specification in which all of the data and functions are defined using a mathematically-based notation.

To ensure that every requirement presented in the informal specification is correctly defined in the semi-formal specification and then further correctly realized in the design specification, requirements traceability must be built and utilized to rigorously check the consistency between different level specifications, but how to systematically and somehow automatically build the traceability is still an open problem.

In this paper, we focus on this problem and propose an approach to building requirements traceability as a solution. The essential idea of our approach is that every item in the informal specification, which can be a data item, functional description, or a constraint, must be linked to one or more items in the semi-formal specification that are expected to define the informal item, and every item in the semi-formal specification must be linked to one or more items in the formal design specification in the same principle.

The rest of the paper is organized as follows. We first introduce some link constraints based on the natural definition of SOFL specifications in Section 2. Section 3 discusses the information extracted from each document and the structure it will be reorganized. We discuss some rules for link creation in Section 4. In Section 5, a three-step approach is discussed for automatically link creation. A brief discussion on the maintenance of the links is given in Section 6. Related work is introduced in Section 7. Finally, we give conclusions and point out future research directions in Section 8.

II. Traceability between specifications

Creating traceability between every two of the three level specifications means connecting the related elements in differ-

Supervisor is Professor Shaoying Liu

ent level specifications and maintain the connection relation during the life cycle of the specification itself. Two elements in two different level specifications are connected if one in the high level specification is "implemented" by the element in the low level specification.

A. Traceability between informal and semi-formal specifications

During the transformation from an informal specification to a semi-formal specification, the elements in the informal specification are usually refined into more precise structures: 1) function in informal specification will be transformed into function, process or module in semi-formal specification, 2) data resource will be transformed into type identifier, constant identifier or state variable, 3) constraint will be transformed into invariant.

Automatically creating trace links between the informal specification and the semi-formal specification is difficult since the informal specification is written in a natural language and its semantics is hard to be understood by computer algorithms. Therefore, the links are usually created manually. Our approach supports the creation of such links by checking whether there is any element in the informal specification that is not linked to any element in the semi-formal specification. If there is such a case, it implies that some "requirement" documented in the informal specification is unlikely to be reflected in the semi-formal specification.

B. Traceability between semi-formal and formal specifications

Due to the commonality of the module structure in both level specifications, the creation of the traceability links is straightforward. This situation also allows for an automatic support for building traceability links. Since this part is the main part of our contribution, we will focus on the discussion of this issue in the rest of this paper.

III. Element Conversion

The elements of interest include process, function, type identifier, constant identifier, state variable and invariant. Their conversion into a pair of concept declaration and structural declaration is described below.

Most kinds of elements declaration in software requirement specification can be divided into two parts — a concept declaration and a structural declaration. The concept declarations are some kinds of notations which are defined all by the user himself. To a rational requirement analyzer, the concept declaration of element will typically indicate the real world concept that the element is willing to represent. The structural declaration goes after the concept declaration typically indicate "how to do" or "how to represent".

Definition 1: Let $\mathbf{C}_e$ denote a set of concept declarations in the specification and S_e denote a set of structural declarations. Then, $E = \{(c, S)|c \in C_e \wedge S \subset S_e\}$ is a set of element defined in the specification. But in some cases, where the concept declaration is missing, we add a special signal "#" to $\mathbf{C}_e$. Such signal indicate that the element do not have a concept declaration. Being a set of structural declaration, S_e is actually a set of pair which has a concept declaration and a value declaration within it. V_e is denoted as a set of value declaration. So we have $S_e = \{(c, v)|c \in C_e \wedge v \in V_e\}$. By applying this definition, we are able to convert the element in SOFL into a common format.

As we have introduced in Section 2, both semiformal and formal specification share the same structure and constitute by same components which are type definition, constant definition, variable definition process definition and invariant definition. We will discuss the judging rules of creating different links for these components respectively in this section.

IV. Criterion for linking elements

In this section, we will discuss several rules used to help determine whether two elements belong to the semi-formal specification and the formal specification respectively should be linked or not. we first discuss the basic principle of all the rules.

In the requirement specification, user may frequently change their mind on how the system should perform by modifying the structural declaration without changing concept declaration. Such characteristic helps us to identify the related element between different level specifications, because whether two corresponded elements change in structural declaration, the concept declaration will remain the same. So in our criterion, we first look into the concept declaration to determine whether two elements should be linked or not. If we can't find the corresponding elements based on the concept declaration, we then look into structural declaration to find the element with same implementation.

Since links created based on the concept declaration and the structural declaration don't have the same trustability, we create four kinds of links as follow.

A. Credible link

The credible link is a kind of link which we strongly believe that the linked elements represent the same concept in the real world. If the two elements within different level specifications have the same concept declaration, a credible link will be created.

Definition 2: Let E denote a set of elements. C_e is a set of concept declarations and e is a an element belonging to E. Then, we define $C(e) : E \rightarrow C_e$.

The criterion for creating credible link is then defined as follow:

Criterion 1: Let MO denote the module defined in the semi-formal specification, and FM denote the module defined in the formal specification. e_1 and e_2 are two element defined in MO and FM respectively. CL is denoted as a set of credible link between MO and FM. $CL = \{(e_1, e_2)|e_1 \in MO \wedge e_2 \in FM \wedge C(e_1) = C(e_2)\}$

B. Suspected link and Incredible link

The suspected link and incredible link are both created automatically by looking into the structural declaration of two

element. But structural declaration of the element has two different format :

Definition 3: Let E denote a set of element. C_e is a set of concept declaration. S_e is a set of structural declaration. e is a an element belongs to E. $S(e) : E \rightarrow S_e$.

Definition 4: Let s denote a structural declaration belongs to S_e. V_e is denoted as a set of value declaration belongs to each structural declaration. $V(e) : S_e \rightarrow V_e$.

Using the function definition above, suspected link and incredible link are created based on the criterion blow.

Criterion 2: Let MO denote the module defined as semi-formal specification, and FM as the module defined in formal specification. e_1 and e_2 are two element defined in MO and FM respectively. SL is denoted as a set of suspected link between MO and FM. $SL = \{(e_1, e_2) | e_1 \in MO \wedge e_2 \in FM \wedge C(e_1) \neq C(e_2) \wedge C(S(e_1)) = C(S(e_2)) \neq \#\}$

Criterion 3: We use the same notation with criterion 2 and let IL denoted as a set of incredible link. $IL = \{(e_1, e_2) | e_1 \in MO \wedge e_2 \in FM \wedge C(e_1) \neq C(e_2) \wedge [C(S(e_1)) = C(S(e_2)) = \#) \vee (C(S(e_1)) \neq C(S(e_2))] \wedge V(S(e_1)) = V(S(e_2))\}$

C. Missing link

The missing link is a slightly different from the other three ones. According to the SOFL specification inspection technology, any element defined in the semi-formal specification should be implemented in the formal specification [6]. So the missing link represents a link between one element in the semiformal specification and a missing element which should have been implemented in the formal specification. Having this kind of link means after the three criterions been applied, the element defined in semi-formal specification still remain unlinked. Mathematically speaking, this criterion is defined as follow:

Criterion 4: We denote $withinCL(e)$, $withinSL(e)$, $withinIL(e)$ as the function showing whether element e in the semi-formal specification has a credible link, a suspected link or a incredible link. Let ML denoted as a set of missing link. $ML = \{e | e \in MO \wedge withinCL(e) \wedge withinSL(e) \wedge withinIL(e)\}$

One thing that we should emphasize here is that all criterions can only be applied to the elements declared within module specifications. We will discuss the linking criterion for the module later since module is has a more complicated structure to which the criterion above may not be appropriate to apply. Making judgment on linking two modules will be discussed in the following section.

V. Automatically Link Creation

In our approach, we will go through three steps to create trace linking automatically. We call these three steps dividing, judging and combining, respectively. In this section, we will introduce the purpose and approach of each step in detail.

A. Step 1: Dividing

To create a link between two elements belong to semi-formal and formal specification respectively, we first pick up an element in the semi-formal specification and figure out its corresponding element in formal specification. It is theoretically acceptable. But unfortunately, figuring out the corresponding element will force us to go through the whole specification in practice. Dividing step is to split the specification into several little pieces so that we do not need to search through the whole specification to find the potential related elements.

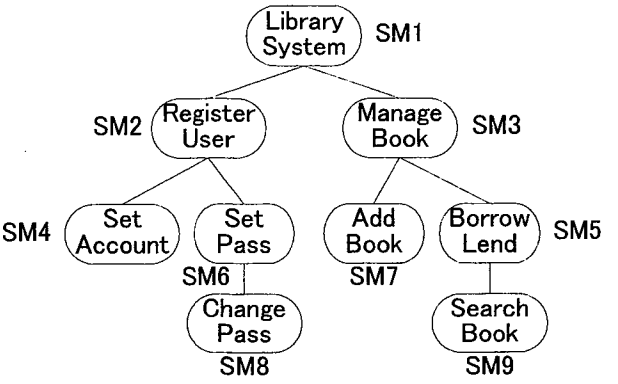

Fig. 1. Tree diagram convert from semi-formal specification

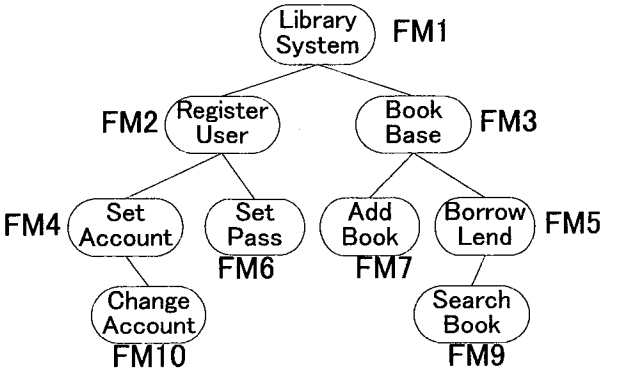

Fig. 2. Tree diagram convert from formal specification

Before we discuss the approach of splitting the tree, another SOFL specification's feature should be introduced here. Both semi-formal and formal specifications are constitute by several module declarations. But these modules are not all on the same level. In SOFL, in case that the process within a module is quite complicated, we may decompose this process into another module and declare that this module is decomposed from a high level module. Having such relationship between modules, we can actually construct a tree diagram against the SOFL specification. In the tree diagram, each node represent a module. A module decomposed from high level module is called a child module, while the high level module is called a parent module. A child module is represented by a child node in the tree belongs to a parent node which represents a parent module. Semi-formal specification's tree diagram is shown in Figure. 1. The tree diagram of the formal specification is shown in Figure. 2.

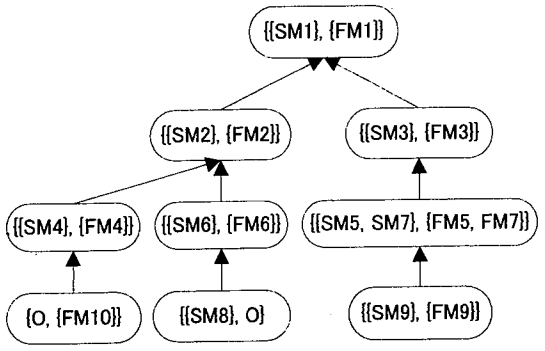

Fig. 3. Derived tree from tree diagram of semi-formal and formal specification

The algorithm we introduced next doesn't look into the definition within each module, so we just show the module name in Figure. 1 and Figure. 2. There are three difference between Figure. 1 and Figure. 2. After converted from semi-formal to formal specification, we miss module Change Pass in semi-formal specification and have a new module Change Account formal specification. Also module Book Manage changes its name to Book Base after the conversion.

After applying our algorithm to both tree diagrams, we will derive a tree diagram with each node represent a pair of sets of modules belong to semi-formal and formal specification respectively.

Definition 5: Let N represent the node in the derived tree. $N = (SM, FM)$. SM and FM are the subset of modules defined in the semiformal and formal specification respectively.

Algorithm 1: Denote SMT as the tree diagram represent the semiformal specification while FMT represent the formal specification. Let DT be the tree derived by the algorithm.

Step 1: Pick up the top node in SMT and FMT. If the module name equal, create N in DT, go to Step 2. Otherwise, put all sm and fm on the same layer into N in DT

Step 2: Find all the child sm and fm of SM and FM within N which have the same module name, create N in DT until there is no child left Apply Step 2 to every new node. If there still child left, go to Step 3.

Step 3: If only fm left, create N with sm missing. If only sm left, create N with fm missing. If both left, put all sm and fm on the same layer into N in DT .

After applying this algorithm to the two tree diagrams in Figure. 1 and Figure. 2, we derive a new tree shown in Figure. 3 whose nodes are either a pair of two modules that have highly potential to related with each other or a pair of sets of modules between some of which relation may exist. We use the notation of each module beside the module name to represent them in Figure. 3.

In the next step, we start with each leaf node in the new tree but not pick up a module in semi-formal specification while have all the module in formal specification as a perspective alternative.

B. Step 2: Judging

Considering the complex structure of the module, such kind of judgment is not quite trustful. In this section we will introduce our approach on making judgment on how close two modules are related by looking into the elements within the modules. In the former section, we have already introduced the judging rules for linking two elements in the semi-formal and formal specification respectively. Based on these rules, linking between two elements can be finished by computer automatically. This section will discuss the way of linking to module automatically. The main idea here is changing the problem of module linking into a problem of quantitative evaluation. The detail will be discussed below.

Firstly, we pick up two potentially related modules and apply the criteria on judging rules for element which we have introduced in the former section to every kind of elements in both module. Recall that in our approach four kinds of links which represent different level trustability will be create by using the judging rules. In the judging step each kind of link will be assigned a quantitative value as a score: 1) credible link =10, 2) suspected link = 6, 3) incredible link = 4, 4) missing link = 0.

Giving each link a score, we turned the level of trustability into a quantitative value. Then we simply calculate the mean trustability of the links between same type elements using the following formula.

Definition 6: Denote C as the number of credible link, S as the number of suspected link, I as the number of incredible link and M as the number of the missing link. Let A_E be the average trustability of element E.

$$A_E = \frac{10 \times C + 6 \times S + 4 \times I + 0 \times M}{C + S + I + M} \quad (1)$$

In the Formula. 1, we have the aggregated value of the all the links in the numerator and the number of the all the links in the denominator. The result from the formula is the mean value of links we have which can be considered as a level of trustability on the whole. From Formula. 1, we can easily find that $A_E \in [0, 10]$.

Having the quantitative value of trustability for each component within one module, we will calculate the level of trustability for the module based on the value of each component. Simply sum of the value of the component together sounds very straight forward. But as the complexity of each component is not equal, having more elements with complicate structure linked will make the link between two modules more trustful. So a weighted sum calculation sounds more precise.

How much weight should be assigned to each component comes to be a problem. The basic idea is that the more complicate component will have more contribution to the trustability which should be assign a larger weight than others. But even the basic idea is the same, there's still a lot different ways to given a set of weight which all of them are actually acceptable for different users. So in this paper, we give function and process a weight equal 25%, while constant identifier, type

identifier, state variable a weight equal 15%, and type invariant 5%.

Based on the given weights, we using the following formula to calculate the WM(weighted mean) of each component's linking trustability as the trustability of the link between two module. In the formula we use the following function named $isDef$ to check whether a given type of component has been defined in the semi-formal specification.

Definition 7: Let SM be a semi-formal specification. P, F, C, T, V, I are denoted a process, function, constant identifier, type identifier, state variable, type invariant, respectively. E = {P | F | C | T | V | I}.

$$isDef(SM, E) = \begin{cases} 1 & if \quad \text{E is defined in SM} \\ 0 & if \quad \text{E isn't defined in SM} \end{cases} \tag{2}$$

$$WM = \frac{A_M}{Total} \tag{3}$$

$$\begin{aligned} A_M &= 25\% \times (A_P + A) + \\ &\quad 15\% \times (A_C + A_T + A_V) + \\ &\quad 5\% \times A_I \end{aligned}$$

$$\begin{aligned} Total &= 25\% \times (isDef(SM, P) + isDef(SM, F)) + \\ &\quad 15\% \times (Def(SM, C) + isDef(SM, T) + \quad (4) \\ &\quad isDef(SM, V)) + 5\% \times isDef(I) \quad (5) \end{aligned}$$

In Formula. 3, we calculate the weighted mean of each component's trustability in the numerator. But leaving some of the components undefined is also acceptable in SOFL specification. In such case, by dividing the sum of weight who are defined in the module, we can automatically adjust the weights used in the numerator.

In this section we calculate the trustability of each component as A_E. Then based on A_E, we using Formula. 3 to calculate WM.As $A_E \in [0, 10]$, we can easily find that $WM \in [0, 10]$.We now convert the value of WM into one kind of link between two modules using function $toLink$.

$$toLink(WM) = \begin{cases} \text{Credible Link} & if \quad WM = 10 \\ \text{Suspected Link} & if \quad WM \in [6, 10) \\ \text{Incredible Link} & if \quad WM \in [4, 6) \\ \text{Missing Link} & if \quad WM \in [0, 4) \end{cases}$$

The type of link between two modules is the final judgement in the judging result. In the next section, based on this result, we will discuss judging not only the leaf node but also the nonleaf nodes in the derived tree from the dividing step and finally link all the modules and elements in the specification.

C. Step 3: Combining

In this section, we will start from the leaf node in the derived tree, push the result up to their parent nodes and finally reach the top node as an end. The main idea here is combining the linking result of two child modules as the just one link between two processes in their parent nodes.

Since the amount of modules within SM and FM might be different from each node. the combining operation applied to each nodes will be a little different. We divide the node into four type of structure. The combining process will be discussed respectively.

1) Type 1 node: Type 1 node is a node defined as follow.

Definition 8: Denote N as a type 1 node. $N = (SM, FM)$ where $SM = \{sm\}$ and $FM = \{fm\}$.

A type 1 node is a most common node that only has one module in both SM and FM. we conduct the judging step and pass the result to its parent node. If the type 1 node is a nonleaf node, we just add the link pass from the all the child nodes as links between process before we apply the judging operation

2) Type 2 node: Type 2 node is the more complicated node defined as follow.

Definition 9: Denote N as a type 2 node. $N = (SM, FM)$ where $SM = \{sm_1, sm_2, ...sm_i\}$ and $FM = \{fm_1, fm_2, ...fm_i\}$

Type 2 node has multiple modules within both SM and FM. Combining operation for leaf node will go through 2 steps. Denote the type 4 node ready to be deal with as $N = \{SM, FM\}$

Step1: Pick up one module denoted as sm in SM, apply the judging operation to sm and each module in FM. Link fm with sm which have the largest WM between them. Pass the link to the parent node in derived tree. Create a sibling node denoted as $N_{sibling}$ of N. Move sm and fm from SM and FM within N to SM and FM within $N_{sibling}$ if $WM \neq 0$. If $WM = 0$, only move sm.

Step2: Move all the sibling nodes of sm and fm in specification tree diagram to $N_{sub} = \{SM_{sub}, FM_{sub}\}$ which is another sibling node of N. Repeat Step1 to N_{sub} until $SM_{sub} = \varnothing$.

3) Type 3 and type 4 node: Type 3 node is a node defined as follow.

Definition 10: Denote N as a type 2 node. $N = (SM, FM)$ where $SM = \{sm\}$ and $FM = \varnothing$.

Type 3 node have a module in SM but no potential related module in formal specification which always cause by the change in the structure of the tree diagram of the specifications. To such kind of node, we pass a missing link to the parent node base on the concept that each definition in semi-formal specification should be implement in formal specification..

Type 4 node is a node defined exactly conserved with type 2 node.

Definition 11: Denote N as a type 2 node. $N = (SM, FM)$ where $SM = \varnothing$ and $FM = \{fm\}$.

Type 4 node has a module in FM but no potential related module in semi-formal specification. This kind of node is probably cause by the decompose operation from turning semi-formal specification to formal specification. To such kind of node, we simply skip this node and move on to its sibling nodes.

Going through the three steps, we finally trace all the elements and the modules.

VI. MAINTENANCE

Maintenance means that after the trace links are created, users are required to eliminate all the missing link and we also strongly recommend user check the suspected link and the incredible link to make sure they are correct. And the manually change to the links will cause the whole links change.

The maintenance of traceability environment also contains maintenance of consistency between each specifications, what means that once an element changed in one specification, the link connect with the corresponding element in the other two specification will be disconnected and replaced by two missing link.

VII. RELATED WORK

A large amount of publications have affirmed that requirement traceability plays an essential role in software engineering [8][7]. However, once the problem come to how the trace information can be create and maintain, the lists become very short.

Works of Gotel and Finkelstein [7] have done an investigation on the traceability problem and discuss the reason for which it still exist. One reason they stated is that the lack of pre-requirements traceability. They argue that tracing requirement is not enough since the requirement may not be capture rationale. We agree with this point, however, we also believe that extra time consumption cause by manually trace information creation also limits the adoption of the current approach in the industrial practice.

The approaches introduced by Haumer et al. [2] and Jackson [9] is a small sample of manual traceability technique. Since even a small system can still cause the traceability become complex. Manual traceability detection, though corrective, may cost too much extra effort.

Easterbrook and Callahan [10] introduced an approach for verification and validation of specification using formal method. In their works, they introduced an AND/OR table as an intermediate to relate textual requirements and SCR model. However, they didn't give the concrete process of the trace information generating.

Despite some deficiencies of the approach we mentioned above, all techniques discussed are useful since they cover the software development approach and requirement modeling techniques outside formal method domain.

VIII. CONCLUSION

In this paper we presented an approach supporting the automatically generation of trace information by using SOFL method. We discuss the corresponding relationship between heterogeneous requirement specification and the way in which we draw out the necessary information from the specification and convert into a structure format. We then go through a three-step approach for creating the trace information automatically. At last we discuss the maintenance of the trace information has been created. A major strength of this approach is that it take the advantage of the structuralized specification in formal method and automate the process of the trace information creation without additional information outside the specification. Also our approach can be adjust to other formal method specification as long as its specification has a modulized structure which can be convert into a tree diagram.

The key contribution of our approach is reduce the enormous efforts and complexity of generating traces by automatically deriving trace information from the existed documentation. This leads to a lower cost for adding traceability into software development process and consequently more benefit will obtained.

Further work will concentrate on a more precise judgment on the potential related module to accelerate the performance of trace generate and an self-adjustable weight for calculate the trustability between two module. Also a tool will be developed to support our approach in the near future.

REFERENCES

[1] Sooriamurthi, R., Introduction to Programming and Software Development: an Interactive, Incremental and Iterative Approach, Proceedings of the 36th Annual Meeting of the Decision Sicences Institute DSI-2005, San Francisco, California, pp 1851-1856

[2] Haumer, P., Pohl, K., Weidenhaupt, K., and Jarke, M., Improving Reviews by Extending Traceability, Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS), 1999.

[3] Clarke, S., Harrison, W., Ossher, H., and Traa, P. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code, Proceedings of 1999 ACM SIGPLAN Conferencer on Object-Oriented Programming, Systems, Languages, and Applications, Dallas, TX, October 1998, pp.325-339

[4] Hughes, T. and Martin, C. Design Traceability of Complex Systems, Proceedings of the 4th Annual Symposium on Human Interaction with Complex Systems, 1998, pp.37-41

[5] S. Liu, Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, 2004.

[6] S. Liu, Tamai. T and Nakajima. S, A Framework for Integrating Formal Specification, Review, and Testing to Enhance Software Reliability, International Journal of Software Engineering and Knowledge Engineering, Volume 21, Number 2, March 2011, pp. 259-288

[7] Gotel, 0. C. Z. and Finkelstein, A. C. W., An Analysis of the Requirements Traceability Problem, Proceedings of the First International Conference on Requirements Engineering, 1994, pp. 94-101.

[8] Watkins R. and Neal M., Why and How of Requirements Tracing. IEEE Sopare 11(4), 1994, 104-106.

[9] Jackson, J., A Keyphrase Based Traceability Scheme, IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 1991, pp.2-1-214.

[10] Easterbrook S.; Callahan J., Formal Method for Verification and Validation of partial specification: A Case Study, Journal of Systems and Software, Volume 40, Number 3, March 1998, pp. 199-210(12)