PAPER *Special Section on Knowledge-Based Software Engineering*

Locating Concepts on Use Case Steps in Source Code*

Shinpei HAYASHI[†a)], ***Member***, **Teppei KATO**[†], ***Nonmember***, ***and*** **Motoshi SAEKI**[††], ***Member***

SUMMARY Use case descriptions describe features consisting of multiple concepts with following a procedural flow. Because existing feature location techniques lack a relation between concepts in such features, it is difficult to identify the concepts in the source code with high accuracy. This paper presents a technique to locate concepts in a feature described in a use case description consisting of multiple use case steps using dependency between them. We regard each use case step as a description of a concept and apply an existing concept location technique to the descriptions of concepts and obtain lists of modules. Also, three types of dependencies: time, call, and data dependencies among use case steps are extracted based on their textual description. Modules in the obtained lists failing to match the dependency between concepts are filtered out. Thus, we can obtain more precise lists of modules. We have applied our technique to use case descriptions in a benchmark. Results show that our technique outperformed baseline setting without applying the filtering.

key words: *concept location, use case description, dependency analysis*

1. Introduction

In software maintenance, it is important to understand features [3] to be maintained [4]. The maintenance process includes the addition of a new feature and the modification of existing features. Such tasks require the understanding of existing source code, which is time-consuming [5]. To support such processes, a technique must be devised or adopted to reduce the cost of understanding existing code when modifying them.

Feature or concept location in the source code [6], [7] is an important task for program understanding, where analysts identify correspondence between features considered in the *problem domain* and modules implemented in the *technical domain*. A problem domain is recognized by stakeholders of a system under discussion, where features are defined in the documentation or in their mind. However, a technical domain is recognized by computers, where features are implemented in modules such as classes, functions, methods, and statements in source code. These two domains must mutually correspond, but the mapping between them is not straightforward. Identifying and understanding the gaps separating these domains are major obstacles and challenges for feature location.

Many existing feature or concept location techniques use a textual query as an input and find relevant modules in the source code as outputs [7], [8]. However, when analysts use a structured description as an input, such as a use case description consisting of multiple use case steps, they might want to know not only the list of modules relevant to the whole feature but also those relevant to each sub-step separately. Even if they have no use case description, they might have already known or imagined that a feature is implemented using a set of smaller concepts based on their skills or domain knowledge. At any of these cases, it is useful to understand the implementation of all features by obtaining a set of modules associated with each sub-concept by regarding a feature as a composed set of sub-concepts[**] [6], [9]. Here, a sub-concept is a concept of which a feature consists, e.g., a step in a use case description.

However, it is difficult to obtain the concept location result of each sub-concept using existing techniques. Most existing concept location techniques require a single query as the input. When using the whole description as the input, it is difficult to recognize which modules in the output are associated with which sub-concepts. When using a short description of each sub-concept as input separately and applying concept location multiple times, the obtained results do not consider the relations among sub-concepts such as data passing or transition of the procedural state. Therefore, the obtained results include irrelevant modules.

As described herein, we propose a feature location technique for sub-concepts based on their mutual relation. We can consider several relations among sub-concepts, including a time-order (a concept works after another concept) or data-dependency (data associated with a concept are transferred to another concept). Our technique checks each module of the output of an existing concept location technique for the ability to satisfy such constraints. By filtering out modules of no place for satisfying the constraints from the obtained result, we obtain a better ranking.

We have automated our technique. By comparing the ranking obtained using it and those using the existing techniques, we demonstrate that it can produce more effective

Manuscript received May 9, 2023.
Manuscript revised October 23, 2023.
Manuscript publicized December 20, 2023.

[†]The authors are with Department of Computer Science, Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
[††]The author is with Department of Software Engineering, Faculty of Science and Technology, Nanzan University, Nagoya-shi, 466–8673 Japan.

[*]This paper is extended based on [1], which appeared in proceedings of the 47th IEEE Annual Computers, Software, and Applications Conference, © 2023 IEEE.

a) E-mail: hayashi@c.titech.ac.jp
DOI: 10.1587/transinf.2023KBP0004

[**]For brevity, in this paper we regard our whole target of location as a feature consisting of sub-concepts. We use *feature location* for locating a feature and *concept location* for locating a sub-concept.

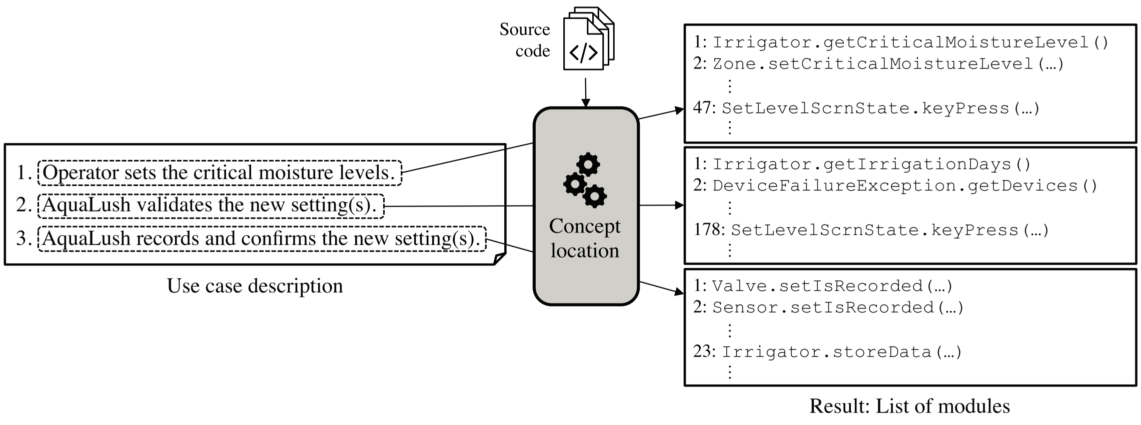

Source code
1. Operator sets the critical moisture levels.
2. AquaLush validates the new setting(s).
3. AquaLush records and confirms the new setting(s).
Use case description
Concept location
1: Irrigator.getCriticalMoistureLevel()
2: Zone.setCriticalMoistureLevel(...)
47: SetLevelScrnState.keyPress(...)
1: Irrigator.getIrrigationDays()
2: DeviceFailureException.getDevices()
178: SetLevelScrnState.keyPress(...)
1: Valve.setIsRecorded(...)
2: Sensor.setIsRecorded(...)
23: Irrigator.storeData(...)
Result: List of modules

Fig. 1 Applying concept location separately for each step. The use case description is from [2] with modification.

ranking. The main contributions of this paper can be summarized as follows:

1. A new technique for supporting the process of understanding existing code in the maintenance phase by locating the sub-concepts of a structured document based on a concept location technique.
2. An empirical evaluation demonstrating that the proposed technique is more effective for locating use case steps compared with existing approaches.

The remainder of this paper is organized as follows. In Sects. 2 and 3, we respectively describe the motivation of this research and the proposed technique. Section 4 presents automation of our approach. Section 5 shows the result of our evaluation to confirm the effectiveness of our approach using an experiment. We discuss related work in Sect. 6. Finally, we conclude our manuscript in Sect. 7.

2. Motivation

Feature location is the process of finding the modules relevant to the given feature in source code for understanding implementation of it [6], [7]. Most existing feature location techniques mainly find modules to the given *single* feature. These accept the single description of a feature and output a ranking of modules relevant to the given feature description.

For software development, we sometimes use structured documents for feature specification. An example of structured documents is a use case description. We can regard each use case as a feature, and therefore, we regard a use case description as a feature description. Because use case descriptions include rich information of the target use case (feature), they can be regarded as good inputs for feature location techniques.

Given a use case description as a feature description, more sophisticated locations of the features are desired. Each step in the use case description clearly specifies the procedural behavior of the target feature. Therefore, understanding the whole feature by understanding each subsequent behavior step-by-step is useful for analysts. For such effective understanding, obtaining the set of modules related to each step is necessary.

However, using existing techniques, it is difficult to obtain the concept location result of each sub-concept effectively. Most existing concept location techniques require a single query for the input and output of a single list of modules. When using the whole description as a single input, it is difficult to recognize which modules in the output are associated with which sub-concepts. When using a short description of each sub-concept as input separately and applying concept location multiple times, the obtained results do not reflect relations among sub-concepts such as data passing or transition of the procedural state. Therefore, the obtained results include irrelevant modules.

An example of application of the existing technique is shown in Fig. 1. Consider that an analyst regards the use case description shown in the figure as a feature description. This use case description is a simplified version of the basic flow of the use case "Set Irrigation Parameters" in the AquaLush example system introduced in the book by Fox [2]. This feature is implemented as a sequence of three steps. Each sentence of the steps is useful for inputs of an existing lexical-based concept location technique. Each obtains the list of modules relevant to each step. The right side of Fig. 1 shows three lists of modules, which are the result of concept location based on the vector space model (VSM) [10]. The highest modules implementing Steps 1, 2, and 3 that appeared in the obtained lists of modules are, respectively, `SetLevelScrnState.keyPress(...)`, `SetLevelScrnState.keyPress(...)`, and `Irrigator.storeData()`. Unfortunately, they appeared in a low rank so that the analyst must check many modules to reach the actually relevant modules.

Interactions and dependencies exist between steps, e.g., data passing or execution. These dependencies must be realized in the implementation environment. The located modules should be followed based on the dependencies. A module implementing the focused concept should not only be relevant in terms of behavior of the concept but also be coordinated with other modules following the specified

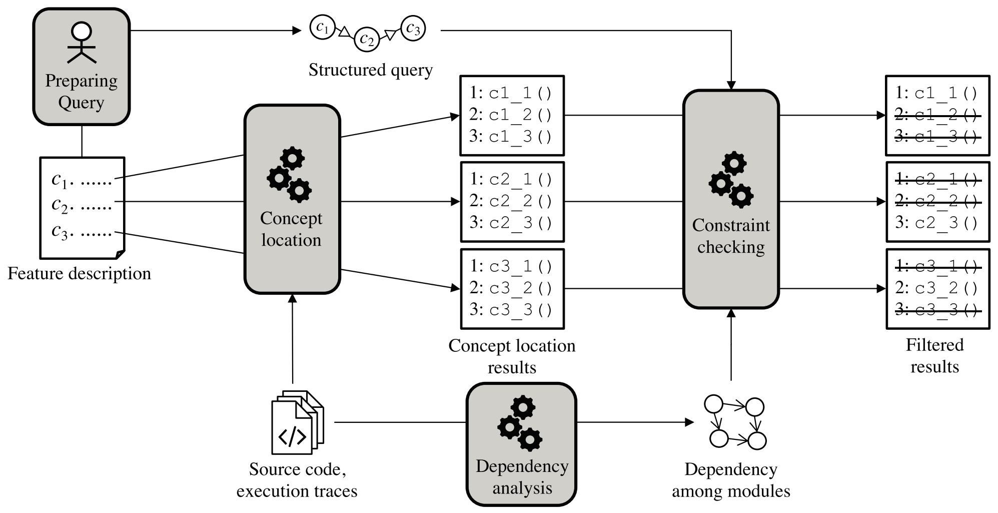


Fig. 2 Overview of our technique.

dependency. Therefore, when applying concept location to each sub-concept, a technique is desired to output the list of modules that is not only relevant to the target sub-concept but which also satisfies the dependencies among sub-concepts.

3. Proposed Technique

3.1 Overview

In this paper, we propose a module extraction technique that is relevant to each sub-concept of the target feature. Our technique uses dependency relations between sub-concepts as constraints on the extracted modules.

An overview of our technique is portrayed in Fig. 2. Gray nodes in this figure represent the main processes of the technique. Processes shown with a stickman are performed manually, whereas others are automated. The input of our technique is the description of the target feature, in addition to data related to the target software product such as source code or execution traces. First, the analyst prepares a *structured query* by extracting sub-concepts and their dependency relations from the given feature description. Our structured query is a directed graph of which the nodes and edges respectively represent sub-concepts of the target feature and their relations; a more formal definition will be provided in the next subsection. This query is used for the constraint checking as described below. Next, we apply an existing concept location technique using descriptions corresponding to the extracted sub-concepts as inputs. Thereby, we obtain the rankings of modules as outputs. We extract dependencies between modules by analyzing the source code and/or execution traces of the target product. We use the structured query as constraints on the extracted modules for each sub-concept. The dependency relations specified in the query are regarded as the constraints by which the extracted modules obtained using the base concept location should hold. Modules that do not satisfy the constraints are filtered out from the list of modules. The analyst can obtain an improved list of modules.

The main steps of our technique for improving the results of existing concept location techniques are query preparation, dependency analysis, and constraint checking. More detailed procedures for the respective steps are explained below.

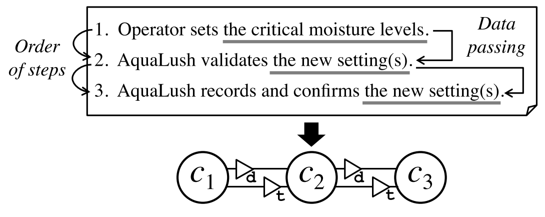
3.2 Preparing Query

First, an analyst extracts sub-concepts from a feature description. In the extraction, analysts should specifically examine the noun in the procedural sentence. As described in this paper, a procedural description such as use case description is used as a feature description as an example. In such a description, the procedure of the target feature is described as a sequence of *procedural steps*. We can regard each step as a sub-concept of the feature. If the analyst cannot find such descriptions including detailed steps, then the analyst can imagine what the features consist of and can then prepare sub-concepts independently.

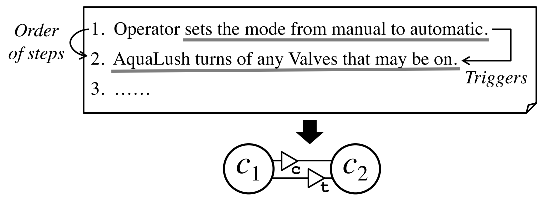
Next, the analyst extracts dependency relations between the extracted sub-concepts and writes them as a structured query. A structured query in this paper is represented as a labeled directed graph: $Q = (C, \rhd)$, of which nodes C and edges $\rhd \subset C \times C \times L$ are sub-concepts and constraints between them. Here, C and L denote sets of sub-concepts and a set of labels specifying the type of constraint $\rhd$. We write a constraint between the concepts c_1 and c_2 labeled with l as $c_1 \rhd_l c_2$.

We have defined the following constraints of three types:

$\rhd_{\mathtt{t}}$: **Time constraint.** We write a *time constraint* $c_1 \rhd_{\mathtt{t}} c_2$ if the concept c_2 should work after the concept c_1 works. In a description, we can extract time constraints by the sequential order of steps because the steps are enumerated in a time order. We often extract time constraints of c_1 to c_2, c_2 to c_3, ... because use case steps in a flow are expected to be executed in the order of top to


Order of steps
1. Operator sets the critical moisture levels.
2. AquaLush validates the new setting(s).
3. AquaLush records and confirms the new setting(s).
Data passing
c_1 c_2 c_3
d t

(a) Use case: Set Irrigation Parameters.


Order of steps
1. Operator sets the mode from manual to automatic.
2. AquaLush turns of any Valves that may be on.
3. ……
Triggers
c_1 c_2
c t

(b) Use case: Toggle Irrigation.

Fig. 3 Extracting structured queries. The descriptions are from [2] with modification to (a).

bottom.

$\rhd_{\mathtt{c}}$**: Invocation constraint.** We write an *invocation constraint* $c_1 \rhd_{\mathtt{c}} c_2$ if c_2 should be invoked during the execution of c_1. In a description, if a step in the basic flow has its alternative flow, then we can extract the invocation constraint between the sub-concept in the basic flow and those in the alternative flow. Users can also extract this constraint when a step is expected to trigger another step, which leads to an invocation from one to another. The invocation constraint can be regarded as a subclass of the the time constraint because an invocation leads to time elapsed between the invocation of the caller and that of the callee.

$\rhd_{\mathtt{d}}$**: Data constraint.** We write a *data constraint* $c_1 \rhd_{\mathtt{d}} c_2$ if data modified during the execution of c_1 should be read during the execution of c_2. In a description, a step such as "An actor inputs x" suggests the data written of x. The steps mentioning x suggest the data read of x. Data constraints can be specified between sub-concepts associated with these steps.

Examples of a structured query are presented in Fig. 3. The top of Fig. 3 (a) is the use case description, which is the same as that in Fig. 1, as the source of the query. This use case description has three steps. These three steps can be sources of the sub-concepts of the target feature. The sub-concepts c_1, c_2, and c_3 can be extracted respectively from the steps. For extraction of the constraints between these sub-concepts, we also use the given use case description. By observing the order of steps, two time constraints $c_1 \rhd_{\mathtt{t}} c_2$ and $c_2 \rhd_{\mathtt{t}} c_3$ are extracted. Next, we can find the fact that the data "critical moisture levels" input by the user in Step 1 is read in Step 2 for confirmation. Then, a data constraint $c_1 \rhd_{\mathtt{d}} c_2$ is extracted. Finally, another data transfer suggests another data constraint $c_2 \rhd_{\mathtt{d}} c_3$. In this data transfer, the data "new settings" confirmed in Step 2 are recorded, and are then used in Step 3. As a result, a structured query shown at the bottom of Fig. 3 is obtained: $Q = (\{c_1, c_2, c_3\}, \{(c_1, c_2, \mathtt{t}), (c_2, c_3, \mathtt{t}), (c_1, c_2, \mathtt{d}), (c_2, c_3, \mathtt{d})\})$. The fact that an analyst prepares this query means that he/she expects that the target feature consists of three concepts c_1 ("Operator sets . . . "), c_2 ("AquaLush validates . . . "), and c_3 ("AquaLush records . . . "). They are executed in order, and data passing occurs in order. The query does not include $\rhd_{\mathtt{c}}$ because the analyst could not find possibilities of invocations between steps. In contrast in Fig. 3 (b), the analyst extracted an invocation constraint between c_1 and c_2 because he/she expected the turning by AquaLush in Step 2 could be triggered by the set of the mode in Step 1.

Cost of query preparation. It is obvious that our technique requires an additional cost of preparing the structural query representing the constraints between use case steps. However, the preparation of queries is mostly not a heavy task because the preparation strategy explained in this section is easy to follow. We asked a student developer to prepare six structured queries for the use case descriptions used in our evaluation in Sect. 5. As a result, the average time used for a use case description (of 4.8 steps in average) was 175 s, and the maximum time used was 363 s for a use case description of eight steps. These results should be considered within an allowance. Although we have not conducted any controlled experiments of preparing queries, we believe that the cost of preparing queries does not lead to a considerable expenditure of time. The exact time spent to prepare each structured query by this developer is shown in Table 1.

Of course, the cost of query preparation may increase with a large use case description, as demonstrated by the above results. Therefore, implementing automated support to reduce the cost of query preparation for large use case descriptions would be beneficial. For example, a tool that identifies data constraints through recognizing word relationships within use case steps, indicated by terms such as "it" or "the X", and subsequently suggests relevant words as the candidate of constraints would be helpful. Following the methodology that identifies patterns in data constraints within descriptions [11], extracting constraints based on natural language patterns stands as a viable approach to enhance query preparation.

3.3 Concept Location

A concept location technique is used for each sub-concept $c \in C$ for obtaining the ordered list of modules relevant to c: $R_c \subseteq M$ (ordered set). Here, M denotes a set of modules in the source code. Hereinafter, we simply call the ordered list of modules relevant to a concept *a ranking for the concept.*

The key idea in this process is reduction of the costs of extracting modules which satisfy given constraints. To verify the constraints among sub-concepts, we analyze the whole set of modules in the source code. However, it includes a large amount of modules that are irrelevant to the target concept, which engenders inefficiency of detection. Our approach is the first to detect modules related to each sub-concept separately to exclude low-relevance modules in advance and then to make the search space a reasonable size.

By checking the constraints among modules that are relevant to each sub-concept, we expect the resulting ranking of relevant modules of high accuracy.

Our approach allows some flexibility as to which base concept location technique is used. Several perspectives when selecting the base concept location technique embedded in our approach are as follows.

- The technique must use natural language descriptions as its inputs because the description of a sub-concept is a part of a structured document.
- The technique is encouraged to specifically examine textual information rather than dependencies between modules. Since the proposed technique performs filtering based on dependency analysis, the base technique can benefit more from the proposed technique if it focuses on text processing; in that case, two techniques can work complementary, resulting in improved performance.

3.4 Dependency Analysis

In our technique, we extract dependencies among modules by checking the constraint among sub-concepts. They can be examined by static and/or dynamic analyses, depending on their type.

Time dependency. We extract a time dependency $m_i \rightarrow_{\mathtt{t}} m_j$ if a module m_i should be executed after the module m_j is executed. As described in this paper, we extract a weaker relation by which the modules are executed in the following order:

$$m_i \rightarrow_{\mathtt{t}} m_j \iff \exists t_i \leq t_j \bullet m_i@t_i \land m_j@t_j.$$

Here, $m@t$ denotes the fact that the module m is executed at time t in an execution trace. As a natural way of checking the order of two method invocation events, we use dynamic analysis to collect when they were executed.

Call dependency. We extract a call dependency $m_i \rightarrow_{\mathtt{c}} m_j$ if a module m_i invokes another module m_j. Call dependencies can be extracted either via static or dynamic analyses. If we use static analysis in case of dynamic analysis inapplicable, then all the possible invocations are collected; invocations occurring in source code are collected. In contrast, using dynamic analysis, an invocation is collected only if the invocation is actually observed.

Data dependency. We extract data dependency $m_i \rightarrow_{\mathtt{d}} m_j$ if a module m_i modifies a variable and another module m_j reads the variable after that. Our technique extracts data dependencies using both static and dynamic analyses. A module reads/writes a variable by referencing the variable identifier or by assignment of the variable. We extract such expressions in source code via static analysis[†]. Furthermore, we deduce additional access patterns such as invocations of collection utilities or accessor methods as data access.

[†]This is just one of the approaches; extraction via dynamic analysis is also possible.

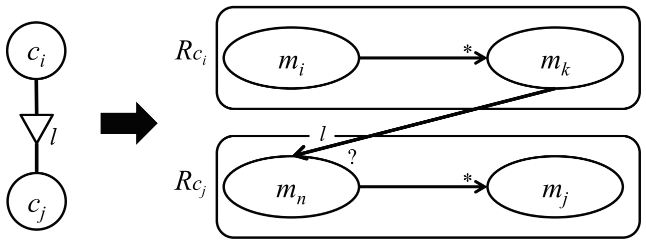


Fig. 4 Conversion from constraints to dependencies.

$$m_i \rightarrow_{\mathtt{d}} m_j \iff w^v_{m_i} \land r^v_{m_j} \land m_i \rightarrow_{\mathtt{t}} m_j \land$$
$$\nexists m, t_i < t < t_j \bullet m_i@t_i \land m_j@t_j \land m@t \land w^v_m.$$

Here, r^v_m and w^v_m respectively denote the occurrence of read and write accesses for the variable v in module m. It holds if and only if a data transfer from m_i to m_j via variable v exists, and there is no method that writes to the variable v after the write access of m_i and before the read access of m_j.

3.5 Checking Constraints

Using the dependencies obtained from source code analysis and/or execution trace analysis, we check the constraints between sub-concepts. A constraint between sub-concepts can be regarded as a constraint between sets of modules relevant to the sub-concepts. That is to say, we check whether a dependency between a module relevant to a sub-concept and another module relevant to the other sub-concept satisfies the constraint between the sub-concepts. We convert a constraint among sub-concepts into another constraint among modules. A constraint between two sub-concepts c_i and c_j can be converted into a constraint among modules m and m', where m and m' are modules that are respectively included in R_{c_i} and R_{c_j}. The resulting rankings of concept location of sub-concepts c_i and c_j consist only of modules that satisfy the converted constraints among modules.

Sometimes a concept is realized not only by a module but by the interaction of multiple modules. The interaction can be achieved by method invocation or access of variables. Therefore, we also allow modules that do not directly satisfy constraints but which have an association with other modules and which can satisfy the constraint via the associated modules. Furthermore, a module in both rankings of the sub-concepts of which a constraint targets can be regarded as satisfying the constraint because it can realize both sub-concepts by itself.

Based on the observations presented above, we convert constraints among sub-concepts into those among modules. For example, consider a constraint $c_i \rhd_l c_j$ shown in Fig. 4. If we write a pair of modules m_i, m_j satisfying the constraint as $m_i \rhd_l m_j$, then this constraint can be decomposed as a sequence of conditions as follows:

$$m_i \rhd_l m_j \triangleq \exists m_k \in R_{c_i}, m_n \in R_{c_j} \bullet$$
$$m_i \rightarrow^* m_k \rightarrow^?_l m_n \rightarrow^* m_j.$$

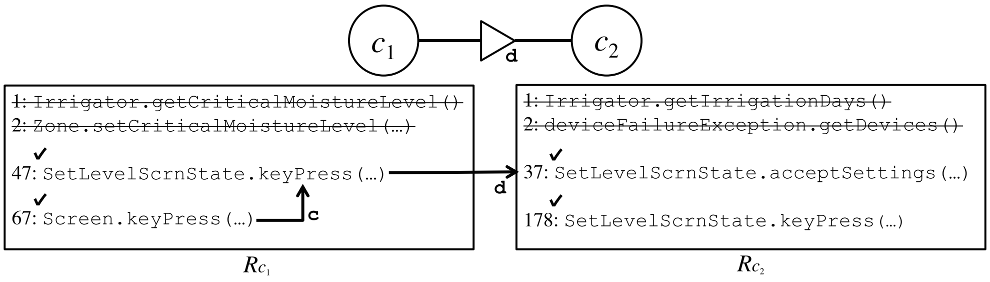

c_1
d
c_2
~~1: Irrigator.getCriticalMoistureLevel()~~
~~2: Zone.setCriticalMoistureLevel(...)~~
✓
47: SetLevelScrnState.keyPress(...)
✓
67: Screen.keyPress(...)
c
d
~~1: Irrigator.getIrrigationDays()~~
~~2: deviceFailureException.getDevices()~~
✓
37: SetLevelScrnState.acceptSettings(...)
✓
178: SetLevelScrnState.keyPress(...)
R_{c_1}
R_{c_2}

Fig. 5 Constraint checking.

Here, $\rightarrow^*$ denotes a zero or greater transitive iteration of data dependency relations (limiting those among the modules in a specific ranking). The symbol $\rightarrow_l^?$ denotes a zero-or-one iteration of dependency of type l.

Figure 5 presents an example of constraint checking. Using the constraint $c_1 \rhd_{\mathtt{d}} c_2$ shown at the top of the figure, we filter the rankings of the concepts c_1 and c_2: R_{c_1} and R_{c_2}. Here, by dependency analysis among modules, we have data dependency: `SetLevelScrnState.keyPress(...)` $\rightarrow_{\mathtt{d}}$ `SetLevelScrnState.acceptSettings(...)`. This dependency satisfies the constraint. The 47th of R_{c_1} and the 37th of R_{c_2} pass the constraint. Additionally, if we have call dependency `Screen.keyPress(...)` $\rightarrow_{\mathtt{c}}$ `SetLevelScrnState.keyPress(...)`, then the 67th of R_{c_1} will be a candidate. Moreover, `SetLevelScrnState.-keyPress(...)` is included both in R_{c_1} and R_{c_2}; the 178th of R_{c_2} holds the condition. Similarly, we can detect the modules not to satisfy the constraint. For example, the first and second ranked modules in R_{c_1} and R_{c_2} can be candidates for removal from the ranking because these modules do not satisfy the constraint.

In our technique, we check all the constraints in the structured query Q and remove modules that do not satisfy the constraints from the rankings. The outputs of our technique are the saturated list of modules. Any modules in the rankings satisfy all the constraints.

Note that our filtering strategy is rather on the conservative side. The interpretation of constraints ($\rhd_{\mathtt{c}}$ and $\rhd_{\mathtt{d}}$) in the relation between methods, we relax the constraint matching to consider transitive relationships between methods. This decision might lead to keep some irrelevant methods in the ranked lists because there might be long transitive relations between the methods. This is an intended decision because our aim is to filter out clearly inappropriate elements. Rather than keeping the methods only satisfying strong conditions, our strategy prefers to remove very irrelevant methods using weak conditions.

4. Implementation

We have implemented a tool chain for automating technique for Java programs at the method level. In this toolchain, we use static information extracted from source code and/or dynamic traces obtained from executing the target program as the dependency among methods. The architecture of the

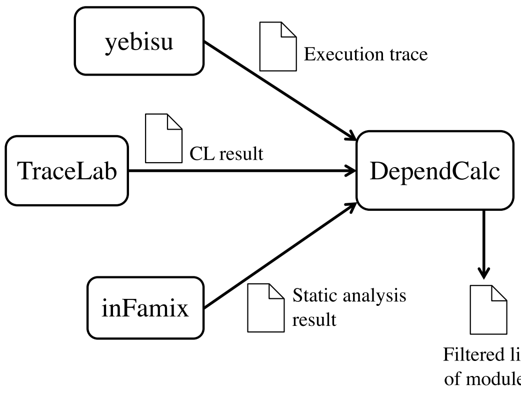


Fig. 6 Architecture of automated implementation.

toolchain is shown in Fig. 6, consisting of existing components: yebisu, inFamix, and TraceLab. The arrows and their labels in the figure respectively represent input–output relations among components and the types of data sent.

In the toolchain, time and call dependencies are extracted using a dynamic analysis tool yebisu [12]. The yebisu outputs the execution traces of method invocation, which include information related to the actually executed methods and the call relation between methods. Using them, dependencies of two types can be extracted. Data dependency is extracted from the static analysis result of inFamix[†], including the occurrences of method invocations and field references in source code. As a basic concept location technique for filtering, we use a lexical technique based on VSM, which is a fundamental of existing feature location techniques, and use TraceLab [10] for implementation of the technique. The inputs of TraceLab-concept location are the whole source code of the target project and the natural language description of the target concept to locate. Using the description, one can obtain the ordered list of methods for each subconcept. Finally, the outputs of three components and the structured query prepared by the analyst are provided to a newly implemented component named DependCalc. Then we obtain filtered lists of methods. This tool checks the dependency among methods in the given list according to the constraints in the given query, and filters out methods that

[†]http://www.intooitus.com/products/infamix (accessed at 2012–02–03). Because the production company has already been closed, it is no longer available.

do not satisfy the constraints.

Different software components can be used as alternatives for the implementation of the proposed method. For example, the dynamic analysis component (yebisu) can be replaced by other implementations that can obtain execution traces, such as SELogger [13]. The base concept location component (TraceLab) produces a ranking based on the textual similarity between the use case description and source code, which is substitutable to other systems that can have similar ability, such as Gensim [14]. The static analysis component (inFamix) extracts the relationship among methods and fields, and such an analysis can also be performed by other systems such as JxPlatform [15].

5. Evaluation

We have evaluated our technique using an experiment. The purpose of this experiment is to ascertain whether our approach can improve the rank of correct modules in the resulting ranking of concept location compared with the baseline.

5.1 Setup

In order to find the target of evaluation, we set up the following criteria:

- Openness: The product materials are available on the web for reproducibility.
- Existence of use case descriptions: The requirements documents including use case descriptions are provided.
- Ease of oracle preparation: The associated modules for each use case step were already specified, or enough information to prepare them is provided.

As our evaluation target, We used AquaBench [16], a benchmark for traceability based on the AquaLush case study [2]. We chose AquaBench because each implemented feature has a use case description. Furthermore, the traceability among use case descriptions, other specifications, and source code are maintained in the benchmark. Therefore, it offers ease of preparation of sub-concepts, structured queries, and the controlled set for evaluating the detected results. AquaBench consists of 119 classes and 463 methods. AquaBench has eight use case descriptions. Because two of them were not associated with automated test cases, we used the remaining six of them as targets of evaluation. All steps in the basic flow of the use case descriptions are regarded as sub-concepts to locate. The name of each use case, the size of each use case and the prepared query, and the time spent to prepare the structured query as mentioned in Sect. 3.2 are shown in Table 1. The structured queries used for these use case descriptions are shown in Fig. 7.

We compared our approach with the baseline approach. For comparison, we prepared both types of results: only static information and using dynamic traces. We prepared resulting lists of modules of four types:

(1) using an existing technique,
(2) using our approach,
(3) using an existing technique with filtering via dynamic trace, and
(4) using our approach with filtering via dynamic trace.

Table 1 Use cases used in this study

		#	# constraints			Time
	Name in the book [2]	steps	$\rhd_{\mathtt{t}}$	$\rhd_{\mathtt{c}}$	$\rhd_{\mathtt{d}}$	[s]
UC1	Toggle Irrigation	3	2	1	1	40
UC2	Set Irrigation Parameters*	3	2	0	2	63
UC3	Manually Control Irrigation	8	8	1	4	363
UC4	Make Repairs	7	8	0	4	249
UC5	Report Failures	3	2	0	1	24
UC6	Irrigate	5	4	1	2	313

*The modification shown in Figs. 1 and 3 was applied to the original version in the book [2].

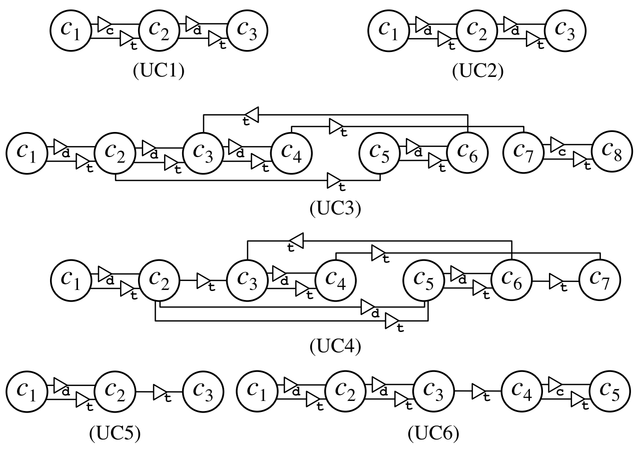


Fig. 7 Structured queries.

Baselines (1) and (3) were intended, respectively, to be compared with our approaches (2) and (4). We used VSM as the baseline concept location technique, which can also be used for the base concept location technique embedded in our technique. This means that our technique produces the same results as the baseline technique before applying the filtering process. VSM was selected because it is well-known, enough simple to clarify the effectiveness of our technique, and is regarded as a good approach to measure the textual similarity for IR-based software engineering tasks [17]. We used TraceLab [18] for computing the VSM score. For (1) and (2), we used static analysis results for call dependencies. For (3) and (4), we prepared a dynamic trace by executing each use case and filtered out the modules that did not appear in the trace.

For comparison of the obtained rankings, we used several measures. First, we used the effectiveness measure [19], defined as the rank of the highest correct module in the given ranking. We compared the average of effectiveness measure values of all the sub-concepts for each feature. To calculate the average, we used only sub-concepts for which rankings include at least one correct module. Additionally, we used the mean average precision (MAP) and the mean reciprocal rank (MRR), which are the standards for comparing ranks.

We prepared the *oracle*, the correct set of modules for each use case, to evaluate the result of the proposed technique. We first extracted traceable design information from the use case descriptions of AquaBench. Since AquaBench provides the traceability information between use case descriptions and modules, we succeeded to obtain the candidate modules to be located. Next, we obtained dynamic traces by executing automated tests associated with respective use case descriptions. The modules in the intersection of this information were the oracle candidates. We manually searched relevant modules from the candidates. Two of the authors prepared the relevant modules separately and compared them. We finalized a set of modules by removing some modules prepared by each author based on a discussion of differences between the results.

The structured queries among sub-concepts were also prepared by one author, precisely following the strategies shown in Sect. 3.2.

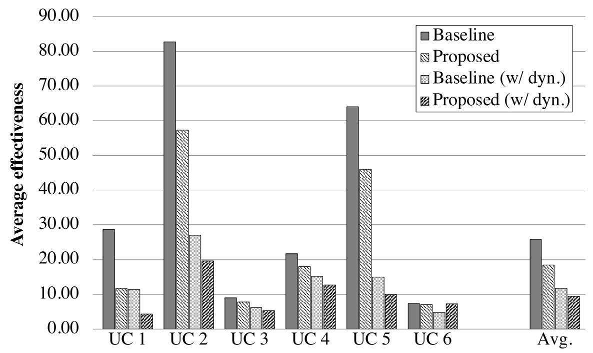
5.2 Results

The summarized results are presented in Fig. 8, for the effectiveness measure, MAP, and MRR. The bar chart in these figures represents the average values of the measure for each sub-concept obtained using several approaches. The four bars associated with a sub-concept respectively denote the values using (1) existing technique, (2) our approach, (3) existing technique filtered out using dynamic trace, and (4) our approach filtered out using dynamic trace. The rightmost bars represent the average/mean of all the values of six sub-concepts. Results must be compared based on their data used: we compared the results of (1) with (2), and (3) with (4). We respectively designate the results of first and second comparisons as static results and dynamic results.

Results show that our approach was able to reduce the average values of these measures for every sub-concept for the comparison of static results, and for 4 out of 6 use cases (features) for comparison of dynamic results. This fact indicates that our approach can improve the rank of correct modules by filtering out irrelevant modules from the results obtained using baseline approaches. Completely, on average we were able to obtain 7.33 percent reduction of effectiveness measure for comparisons using static information, and 2.33 percent reduction for the comparison of dynamic information. The most effective result was that UC 1, which produced reduction of 17 percent for the comparison using static information and 7 percent for the comparison using the dynamic information. However, our technique with dynamic traces produced a worse result than the baseline for UC 6, mainly because the top correct module was filtered out by the constraint checking.

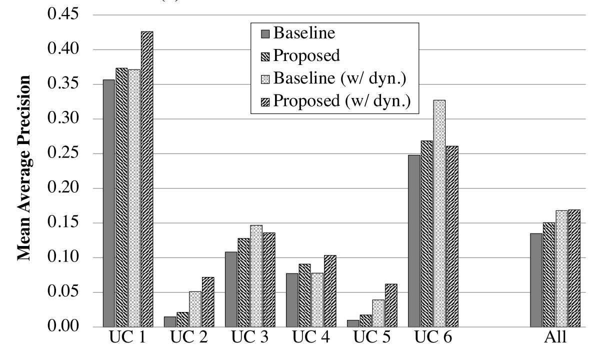
5.3 Discussion

As described above, some correct modules were filtered out by checking constraints. We verified the reason why these modules were filtered out. Results show that they were under another type of dependency we had not expected. For example, a data constraint exists between sub-concepts c_1 and c_2 of UC 6. Actual dependencies among modules in the rankings are shown in Fig. 9. In this figure, with data obtained by *moduleB* c_1, the correct module of c_1, is transferred to *moduleC* via *moduleA*. Because our constraint does not hold in this situation, the correct modules were filtered out. We must improve the pattern of conversion from constraints among sub-concepts to those among modules by fixing this



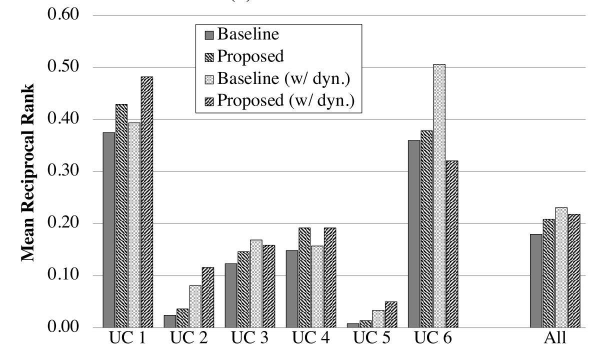
	UC 1	UC 2	UC 3	UC 4	UC 5	UC 6	Avg.
Baseline	28.67	82.67	9.00	21.67	64.00	7.40	25.79
Proposed	11.67	57.33	7.83	18.00	46.00	7.00	18.46
Baseline (w/ dyn.)	11.33	27.00	6.17	15.17	15.00	4.80	11.75
Proposed (w/ dyn.)	4.33	19.67	5.33	12.67	10.00	7.20	9.42

(a) Effectiveness measure.



	UC 1	UC 2	UC 3	UC 4	UC 5	UC 6	All
Baseline	0.36	0.01	0.11	0.08	0.01	0.25	0.13
Proposed	0.37	0.02	0.13	0.09	0.02	0.27	0.15
Baseline (w/ dyn.)	0.37	0.05	0.15	0.08	0.04	0.33	0.17
Proposed (w/ dyn.)	0.43	0.07	0.14	0.10	0.06	0.26	0.17

(b) MAP.



	UC 1	UC 2	UC 3	UC 4	UC 5	UC 6	All
Baseline	0.37	0.02	0.12	0.15	0.01	0.36	0.18
Proposed	0.43	0.04	0.15	0.19	0.01	0.38	0.21
Baseline (w/ dyn.)	0.39	0.08	0.17	0.16	0.03	0.51	0.23
Proposed (w/ dyn.)	0.48	0.12	0.16	0.19	0.05	0.32	0.22

(c) MRR.

Fig. 8 Experimental results.

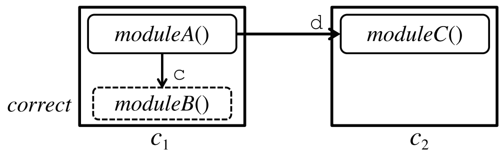


Fig. 9 Example structure by which the technique fails to detect.

problem.

5.4 Threats to Validity

The possibility exists that the creation process of the oracle modules entails a bias. AquaBench, the system we used, a benchmark of software traceability recovery, and includes sufficient information for creating the oracle by us. Additionally, we carefully prepared the candidates of oracles by separated environments by two of the authors and discussed our conclusions well. We believe that our efforts at minimizing the bias can contribute to the validity of the experiment.

In our experiment, we applied our technique only to AquaBench. To demonstrate validity, we must apply our technique to other systems. However, finding appropriate systems for evaluation of our techniques is not easy. Most open systems do not include well-written structured documents. We also checked some benchmark projects of related area, but could not find appropriate projects at all. For example, although TraceLab [18] project provided some benchmark projects, they did not fit our criteria, e.g., difficulty in preparing oracle because the use case descriptions are written in non-English language. As the first step for this research, we believe that our evaluation in this paper is valuable although it targeted only one system, while maintaining the criteria of evaluation targets: openness, existence of use case descriptions, and ease of oracle preparation.

A gap separating the use case steps and the concepts should be identified. Actually, use case descriptions describe a dialog between an actor and systems. Fundamentally, odd steps in use case descriptions begin with “Actor inputs . . . ” and the other even steps “The system does . . . ”. The former steps might not be associated with implementation. However, most software systems have user interfaces. They can be a trigger of the internal features.

6. Related Work

Existing feature location techniques are classifiable into static analysis, dynamic analysis, information retrieval, and their combinations, according to the type of analysis they use to relate features and modules in source code. Recently, Dit *et al.* provided a comprehensive survey of this area [7]. Because our approach is based on dynamic analysis, we emphasize feature location techniques similar to ours in this section. Other kinds of feature location techniques are explained in the literature [7].

6.1 Hybrid Approaches

Marcus *et al.* compared feature location techniques of three kinds based on static analysis: pattern matching using regular expression, dependency search, and latent semantic indexing. They pointed out that each technique has its own benefits and shortcomings [20]. Several hybrid approaches combine textural and dynamic analyses, but their purposes and approaches differ from ours.

The first approach is the combination of ranking. By combining the ranking obtained using textual analysis and that using dynamic analysis, several approaches yielded more accurate feature location results [19], [21]. These approaches apply two techniques in combination, whereas our approach uses textural analysis to filter out the resulting ranking obtained.

Another approach is to reduce the textual analysis search space by limiting the target as the modules included in the given dynamic trace [22]. This approach is intended to reduce the number of applied dynamic analyses, whereas our approach uses dynamic analysis for checking the constraints among sub-concepts.

Several approaches similar to our approach exist to use the dependencies between concepts [23], [24]. However, these approaches differ from ours from the viewpoint of the construction of the approach. They do not filter out the results of existing concept location techniques.

A filtering-based approach for supporting the comprehension of software systems using use cases has also been proposed [25]. This approach filters out entities that are common to a high percentage of use cases for simplifying the complex structure of the whole system. However, they did not use the dependencies between two concepts as constraints for filtering the results of concept location.

6.2 Use of Structured Queries

Several approaches exist for seeking some information for modules using structured queries, such as [26]–[29]. However, most approaches are undertaken to seek modules or snippets of the whole source code or a set of software products. In contrast, our approach used structured queries to filter the results of concept location.

Hill *et al.* claimed the need for annotating feature location results by their roles [30]. In our technique, the preparation of structured queries for a use case description involves the identification of roles of each use case step and their connections.

6.3 FCA-Based Feature Location Techniques

Eisenbarth *et al.* [31] proposed a feature location technique that combines formal concept analysis (FCA), and dynamic and static analyses to identify the correspondence between features and modules in source code. Following their technique, a concept lattice is obtained from the binary relation between scenarios and modules observed in execution traces. Then the lattice is analyzed using mapping between scenarios and features specified by domain experts. Koschke and Quante [32] improved the dynamic feature location technique proposed by Eisenbarth *et al.* [31] to incorporate scenario-feature mapping into a formal context. Based on the case study of two compilers, they discussed tradeoffs among granularity of modules, information gain, and the cost of feature location, changing a level of granularity from routines to basic blocks. They also showed that their technique works well when the number of features is small, but there is a limitation in the numbers of features and scenarios because the number of concepts increases exponentially. Poshyvanyk *et al.* combined latent semantic indexing (LSI) and FCA [33] to demonstrate that it reduces search effort compared to a simple LSI-based technique, which obtains a ranked list of modules according to the relevance to a user-specified query. They apply FCA to organize modules and domain terms taken from search results filtered by certain relevance criteria. By limiting the number of search results, this technique has maintained a concept lattice on a human-readable scale.

6.4 Interactive Approaches

Several human-assisted incremental feature location techniques have been proposed [34]. However, they have specifically examined the refinement of feature location, and have not assumed that features are ambiguous or unknown to analysts. A typical interactive approach is the use of *relevance feedback* [35], which enables developers to give hints for improving feature location results [8], [36]–[38]. The work by Chen and Rajlich [39] proposes to select and understand each of the code fragments. FEAT [40] explores a structural program model interactively and finds code fragments that are related to a feature. However, they offer no mechanisms for giving feedback to users during intermediate steps of FL.

7. Conclusion

As described in this paper, we have proposed a technique for locating procedural steps using concept location. The technique regards a feature a set of mutually related sub-concepts. By defining and checking the constraints among sub-concepts, the results of concept location for each sub-concept are filtered out. Then we were able to obtain the ranking of each sub-concept. We have defined constraints of three dependency types and automated our technique. The evaluation showed that our approach was able to improve the accuracy of concept location compared with an existing approach.

We summarize our future work as follows:

- **Consideration of other types of constraints**, e.g., one using control dependencies,
- **Improvement of dependency patterns of constraints** to remedy difficulties that occurred in the experiment,
- **Implementation of a semi-automated support of query preparation** to help identify constraints among concepts, and
- **Comparison and selection of base concept location techniques** for comprehensive improvement of the effectiveness of our approach.

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Nos. JP22H03567, JP21H04877, JP21K18302, and JP21KK0179.

References

[1] S. Hayashi, T. Kato, and M. Saeki, “Locating procedural steps in source code,” Proc. 47th IEEE Annual Computers, Software, and Applications Conference, pp.1607–1612, 2023.
[2] C. Fox, Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2, Addison-Wesley, 2006.
[3] The Institute of Electrical and Electronics Engineers (IEEE), “830-1998 - IEEE Recommended Practice for Software Requirements Specifications,” 1998.
[4] G.C. Murphy, M. Kersten, M.P. Robillard, and D. Čubranić, “The emergent structure of development tasks,” Proc. 19th Annual Meeting of the European Conference on Object-Oriented Programming, LNCS, vol.3586, pp.33–48, 2005.
[5] T. Vestdam and K. Nørmark, “Maintaining program understanding: Issues, tools, and future directions,” Nordic Journal of Computing, vol.11, no.3, pp.303–320, 2004.
[6] V. Rajlich and N. Wilde, “The role of concepts in program comprehension,” Proc. 10th International Workshop on Program Comprehension, pp.271–278, 2002.
[7] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: A taxonomy and survey,” Journal of Software: Evolution and Process, vol.25, no.1, pp.53–95, 2013.
[8] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, “On the use of relevance feedback in IR-based concept location,” Proc. 25th IEEE International Conference on Software Maintenance, pp.351–360, 2009.
[9] V. Rajlich, Software Engineering: The Current Practice, CRC Press, 2011.
[10] B. Dit, E. Moritz, and D. Poshyvanyk, “A TraceLab-based solution for creating, conducting, and sharing feature location experiments,” Proc. 20th IEEE International Conference on Program Comprehension, pp.203–208, 2012.
[11] J.M. Florez, L. Moreno, Z. Zhang, S. Wei, and A. Marcus, “An empirical study of data constraint implementations in java,” Empirical Software Engineering, vol.27, no.5, 119, pp.1–46, 2022.
[12] H. Kazato, S. Hayashi, T. Oshima, S. Miyata, T. Hoshino, and M. Saeki, “Extracting and visualizing implementation structure of features,” Proc. 20th Asia-Pacific Software Engineering Conference, pp.476–484, 2013.
[13] K. Shimari, T. Ishio, T. Kanda, N. Ishida, and K. Inoue, “NOD4J: Near-omniscient debugging tool for Java using size-limited execution trace,” Science of Computer Programming, vol.206, 120630, pp.1–13, 2021.
[14] R. Řehůřek and P. Sojka, “Software framework for topic modelling with large corpora,” Proc. LREC 2010 Workshop on New Challenges for NLP Frameworks, pp.45–50, 2010.
[15] K. Maruyama, “JxPlatform3.” https://github.com/katsuhisamaruyama/jxplatform3, 2022.
[16] E. Ben Charrada, D. Caspar, C. Jeanneret, and M. Glinz, “Towards a benchmark for traceability,” Proc. 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM

Workshop on Software Evolution, pp.21–30, 2011.

[17] M.M. Rahman, S. Chakraborty, G. Kaiser, and B. Ray, "Toward optimal selection of information retrieval models for software engineering tasks," Proc. 19th International Working Conference on Source Code Analysis and Manipulation, pp.127–138, 2019.

[18] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, M. Gethers, D. Poshyvanyk, J.H. Hayes, and W. Li, "Toward actionable, broadly accessible contests in software engineering," Proc. 34th International Conference on Software Engineering, pp.1329–1332, 2012.

[19] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," IEEE Trans. Softw. Eng., vol.33, no.6, pp.420–432, 2007.

[20] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," Proc. 13th International Workshop on Program Comprehension, pp.33–42, 2005.

[21] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," Proc. 18th International Conference on Program Comprehension, pp.14–23, 2010.

[22] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering, pp.234–243, 2007.

[23] M. Harman, N. Gold, R. Hierons, and D. Binkley, "Code extraction algorithms which unify slicing and concept assignment," Proc. 9th Working Conference on Reverse Engineering, pp.11–20, 2002.

[24] N. Gold, M. Harman, Z. Li, and K. Mahdavi, "Allowing overlapping boundaries in source code using a search based approach to concept binding," Proc. 22nd IEEE International Conference on Software Maintenance, pp.310–319, 2006.

[25] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Scenario-driven dynamic analysis for comprehending large software systems," Proc. 10th European Conference on Software Maintenance and Reengineering, pp.71–80, 2006.

[26] B.P. Eddy and N.A. Kraft, "Using structured queries for source code search," Proc. 30th IEEE International Conference on Software Maintenance and Evolution, 2014.

[27] C.D. Roover and K. Inoue, "The Ekeko/X program transformation tool," Proc. 14th IEEE International Working Conference on Source Code Analysis and Manipulation, pp.53–58, 2014.

[28] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J.X. Yu, "Matching dependence-related queries in the system dependence graph," Proc. 25th IEEE/ACM International Conference on Automated Software Engineering, pp.457–466, 2010.

[29] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," Proc. 18th Working Conference on Reverse Engineering, pp.119–123, 2011.

[30] E. Hill, D. Shepherd, and L. Pollock, "Exploring the use of concern element role information in feature location evaluation," Proc. 23rd IEEE International Conference on Program Comprehension, pp.140–150, 2015.

[31] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," IEEE Trans. Softw. Eng., vol.29, no.3, pp.210–224, 2003.

[32] R. Koschke and J. Quante, "On dynamic feature location," Proc. 20th IEEE/ACM International Conference on Automated Software Engineering, pp.86–95, 2005.

[33] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," Proc. 15th IEEE International Conference on Program Comprehension, pp.37–48, 2007.

[34] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao, "Improving feature location using structural similarity and iterative graph mapping," Journal of Systems and Software, vol.86, no.3, pp.664–676, 2013.

[35] C.T. Meadow, Text Information Retrieval Systems, Academic Press, 1992.

[36] S. Hayashi, K. Sekine, and M. Saeki, "iFL: An interactive environment for understanding feature implementations," Proc. 26th IEEE International Conference on Software Maintenance, pp.1–5, 2010.

[37] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A.D. Lucia, "When and how using structural information to improve IR-based traceability recovery," Proc. 17th European Conference on Software Maintenance and Reengineering, pp.199–208, 2013.

[38] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," Proc. 22nd IEEE International Conference on Software Maintenance, pp.299–309, 2006.

[39] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," Proc. 8th International Workshop on Program Comprehension, pp.241–247, 2000.

[40] M.P. Robillard and G.C. Murphy, "Concern Graphs: Finding and describing concerns using structural program dependencies," Proc. 24th International Conference on Software Engineering, pp.406–416, 2002.

Shinpei Hayashi is an associate professor in School of Computing at Tokyo Institute of Technology. He received a B.E. degree in information engineering from Hokkaido University in 2004. He also respectively received M.E. and D.E. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008. His research interests include software evolution and software development environment.

Teppei Kato received B.E. and M.E. degrees in computer science from Tokyo Institute of Technology in 2012 and 2014, respectively. His research interests include feature and concept location in the area of software maintenance and evolution.

Motoshi Saeki received a D.E. degree in computer science from Tokyo Institute of Technology in 1983. He was a professor in School of Computing at Tokyo Institute of Technology. He is currently a professor in Department of Software Engineering at Nanzan University. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).