PAPER Special Section on Knowledge-Based Software Engineering

Identifying Core Objects for Trace Summarization by Analyzing Reference Relations and Dynamic Properties

Kunihiro NODA[†a)], ***Nonmember***, **Takashi KOBAYASHI**[†b)], ***and*** **Noritoshi ATSUMI**[††c)], ***Members***

SUMMARY Behaviors of an object-oriented system can be visualized as reverse-engineered sequence diagrams from execution traces. This approach is a valuable tool for program comprehension tasks. However, owing to the massiveness of information contained in an execution trace, a reverse-engineered sequence diagram is often afflicted by a scalability issue. To address this issue, many trace summarization techniques have been proposed. Most of the previous techniques focused on reducing the vertical size of the diagram. To cope with the scalability issue, decreasing the horizontal size of the diagram is also very important. Nonetheless, few studies have addressed this point; thus, there is a lot of needs for further development of horizontal summarization techniques. We present in this paper a method for identifying core objects for trace summarization by analyzing reference relations and dynamic properties. Visualizing only interactions related to core objects, we can obtain a horizontally compactified reverse-engineered sequence diagram that contains system's key behaviors. To identify core objects, first, we detect and eliminate temporary objects that are trivial for a system by analyzing reference relations and lifetimes of objects. Then, estimating the importance of each non-trivial object based on their dynamic properties, we identify highly important ones (i.e., core objects). We implemented our technique in our tool and evaluated it by using traces from various open-source software systems. The results showed that our technique was much more effective in terms of the horizontal reduction of a reverse-engineered sequence diagram, compared with the state-of-the-art trace summarization technique. The horizontal compression ratio of our technique was 134.6 on average, whereas that of the state-of-the-art technique was 11.5. The runtime overhead imposed by our technique was 167.6% on average. This overhead is relatively small compared with recent scalable dynamic analysis techniques, which shows the practicality of our technique. Overall, our technique can achieve a significant reduction of the horizontal size of a reverse-engineered sequence diagram with a small overhead and is expected to be a valuable tool for program comprehension.

key words: *dynamic analysis, reverse-engineered sequence diagram, trace summarization, core object, program comprehension*

1. Introduction

Sufficiently understanding software structures and behaviors is one of the most important objectives in program maintenance. Specification and design documents are helpful for program comprehension. Nevertheless, in most cases, those documents do not accurately reflect the system state owing to many modifications and extensions after the first version has shipped.

To address this issue, execution trace analysis techniques are often used for some purposes, such as program comprehension [1] and specification mining [2]–[5]. For aiding in program comprehension, visualizing object interactions in execution traces as reverse-engineered sequence diagrams is a promising approach [1]. However, the massiveness of information in the diagram causes a scalability issue. Therefore, abstraction techniques are very important for the trace visualization approach.

The vertical size of a reverse-engineered sequence diagram grows in proportion to the execution time, while its horizontal size grows in proportion to the number of objects. Most previous works focused on reducing the vertical size of the diagram [6]–[11] or effectively exploring the diagram [12]–[15]. To cope with the scalability issue, decreasing the horizontal size of the diagram is also very important. Nonetheless, few studies have addressed this point [9], [16]–[18]. Thus, further development and improvement of reduction techniques that focus on the horizontal direction are needed.

In this paper, we present a technique of identifying core objects for trace summarization by analyzing the reference relations and dynamic properties. Visualizing only interactions related to core objects, we obtain a horizontally compactified version of a reverse-engineered sequence diagram that contains the system's key behaviors comprised of messages from among the core objects. Our core identification steps are as follows. First, we detect and eliminate the temporary objects that are generated in large quantities during a program execution [19]. To this end, we analyze reference relations and lifetimes of objects in a similar way to escape analysis. Second, focusing on the frequency of access to non-temporary objects, we estimate the importance of those objects. Objects that survive for long periods and have high access frequencies are expected to play core roles in a system. These objects are the *core objects* we strive to identify.

We applied our technique to traces of various open-source software systems to evaluate its feasibility and effectiveness in terms of trace summarization. The results showed that our technique achieved superior reduction performance compared with the state-of-the-art trace summarization technique. Our compression ratio of the horizontal size of a reverse-engineered sequence diagram was 134.6 on average, while retaining core objects in the resulting diagram that were important to comprehend a design overview. The runtime overhead imposed by our technique was 167.6% on average. This overhead is relatively small

Manuscript received November 8, 2017.
Manuscript revised March 10, 2018.
Manuscript publicized April 20, 2018.

[†]The authors are with Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
[††]The author is with Kyoto University, Kyoto-shi, 606–8501 Japan.
a) E-mail: knhr@sa.cs.titech.ac.jp
b) E-mail: tkobaya@cs.titech.ac.jp
c) E-mail: atsumi.noritoshi.5u@kyoto-u.ac.jp

DOI: 10.1587/transinf.2017KBP0018

compared with recent scalable dynamic analysis techniques, which shows the practicality of our technique. Our technique can achieve a significant reduction of the horizontal size of a reverse-engineered sequence diagram with a small overhead and is expected to be a valuable tool for program comprehension.

The main contributions of this paper are as follows:

- The *Reference Escape Analysis* (REA) technique is proposed to detect temporary objects in a system
- A formula to estimate the importance of objects by focusing on access frequency is presented. Visualizing only interactions related to important objects (i.e., core objects), we can obtain a horizontally compactified version of a reverse-engineered sequence diagram.
- We demonstrate the feasibility and the effectiveness of the proposed technique through experiments with various open-source software systems.

This paper is an extended version of our previous work [20]. The main differences from our previous work are as follows:

- Refinement of the algorithms to identify core objects, which improves the overall reduction performance
- Additional experiments on various software systems to mitigate the external validity
- Additional analysis of the sensitivity of the reduction performance when varying the tunable parameters of our algorithm

The remainder of this paper is organized as follows. Section 2 describes key related works. Section 3 details the proposed technique. Section 4 briefly describes how to visualize a summarized sequence diagram by using the results of our technique. In Sect. 5, we evaluate our technique through experiments, and Sect. 6 discusses threats to validity. Section 7 presents our conclusions.

2. Related Work

2.1 Coping with a Scalability Issue of a Reverse-Engineered Sequence Diagram

The vertical size of a reverse-engineered sequence diagram drastically increases in accordance with the increasing execution time. Many researchers have thus proposed message reduction techniques. These methods primarily involve removing trivial behaviors, such as repetitive behaviors [6], [10], [14], and implementation details [9]. Another approach is to divide an execution trace into several phases [8], [11], [21]–[23] that correspond with the starting points of tasks.

Reducing the horizontal size of a reverse-engineered sequence diagram is a likewise important and challenging task. Compared with message reduction approaches, fewer works have presented object reduction techniques. Dugerdil et al. formed object clusters based on the frequencies of objects' interactions in a unit of time. As inter-cluster interactions, they generated a highly abstracted behavioral view of a system. Some studies extracted design patterns and performed object grouping based on design intentions that were realized with the extracted patterns [17], [18]. Hamou-Lhadj et al. calculated the utilityhood for each method and pruned the implementation details using the utilityhood values, achieving the reduction of the numbers of methods and objects [9].

Several approaches differ from the ones that reduce messages or objects. If the developer focuses only on a specific part of a system, extraction approaches [24], [25] are promising. Another approach is effective visualization and exploration rather than information reduction; effective filtering/zooming functionalities [12], a dedicated view for visualizing static/dynamic information [26], partial trace visualization [14], and interactive visualization [15].

2.2 Identifying Important Classes of a Software System

There exist several techniques to identify important classes of a software system. Most of the techniques exploit network analysis and machine learning [27].

Thung et al. and Yang et al. classified whether each class in a reverse-engineered class diagram was important with their original classifier using various metrics (e.g., design and network metrics) [28], [29]. They used the classifier and thereby obtained a condensed version of a reverse-engineered class diagram that was close to a forward designed one.

Meanwhile, Zaidman et al. and Şora proposed techniques for identifying the most important key classes (i.e., core parts) of a system by using network analysis to facilitate the comprehension of the design overview [30], [31]. Note that the number of the key classes, which are important to comprehend the design overview, is around 5–10; this number is much lower than the number of classes contained in a forward designed class diagram.

Those techniques of identifying key classes [30], [31] are similar to our proposed technique in terms of identifying core parts of a system; however, there is a difference between their techniques and ours as follows. Key classes identified by the techniques by Zaidman et al. and Şora tend to be *abstract classes* or *interfaces* that are useful for understanding the structural view (i.e., static aspect) of a subject system. On the other hand, our technique can identify key *concrete classes* that are instantiated at a runtime and thus valuable for understanding the behavioral view (i.e., dynamic aspect) of a subject system. Due to the difference, it is difficult to directly compare techniques by Zaidman et al. and Şora with our technique. We need to use appropriate one according to the purpose of tasks developers undertake.

2.3 Analyzing Object Reference Relationships

Some studies analyzed object reference relations, which relates to our work. Dufour et al., for example, proposed a technique of "blended escape analysis" to characterize temporary data structures and program regions that create and use them [19]. Meanwhile, Lienhard et al. contended that

analyzing object flow—the way in which objects are passed through a system at runtime—is important for understanding the runtime of an object-oriented system [32]. The object flow graph was used for some purposes, such as architectural risk analysis [33] and impact analysis [34].

2.4 Comparison of Our Work with Existing Trace Summarization Techniques

The technique proposed herein is categorized into object reduction approaches. In this section, we compare our technique with related works that focus on object reduction.

Of the existing techniques that achieve the horizontal reduction of a reverse-engineered diagram, the most promising and closest to our approach is the one proposed by Hamou-Lhadj et al. [9].

Hamou-Lhadj et al. proposed the utilityhood metric based on fan-in and fan-out. With the metric, they detected and pruned implementation details (i.e., non-core parts). Their approach is similar to ours in terms of identifying the core and non-core. However, their technique cannot achieve the appropriate reduction of the horizontal size of the diagram because there tend to exist many methods having the same fan-in and fan-out. In Sect. 5, we conduct a detailed comparison of our technique with their one using traces of open-source software systems. Consequently, we show the advantage of our technique in terms of the horizontal reduction of a reverse-engineered sequence diagram.

The technique by Dugerdil et al. [16] provides a highly abstracted view of a system. The technique is more appropriate when developers intend to understand a coarse-grained architectural behavior (e.g., inter-layer interactions in a layered architecture system). The technique by Dugerdil et al. is valuable in the earliest stage of program comprehension. Compared with the technique by Dugerdil et al., our technique focuses on comprehending finer-grained behavior. Our technique aims at helping comprehension in scenarios in which the developers strive to identify the objects that play key roles in each feature or module and understand how the key objects behave.

Design patterns based techniques [17], [18] rely on static analysis and are more favorable for comprehending design intentions. This is because extracted design patterns provide good clues about key structures and behaviors based on system's design intentions. Static analysis relieves the complexity of dynamic analysis of large-scale execution traces; however, it often incurs a certain amount of false-positives. Thus, relying solely on static analysis is insufficient. Analyzing dynamic information specific to current execution scenarios is also necessary. While these works [17], [18] focused mainly on static analysis, we herein examine the effectiveness of dynamic analysis for trace summarization.

3. Identifying Core Objects for Trace Summarization

Our core identification technique consists of the following two steps.

1. Pruning temporaries by Reference Escape Analysis
2. Importance estimation by analyzing access frequency

The 1st step eliminates temporaries that are generated in large quantities at a runtime but not important for comprehending system's key behavior. The 2nd step estimates the importance of each object by analyzing access frequency and identifies the core objects for a system. Here, it is worth noting that the 1st step plays a role of a noise-reducer for the 2nd step. Because some of the temporaries expect to have high access frequencies, removing such noisy objects before the 2nd step is important to estimate the importance of each object more correctly. (The effect of the noise-reduction can be seen in the result of our experiment described in Sect. 5.3.2.) We elaborate each of the steps in the following sections with a running example shown in Fig. 1.

The left graph in Fig. 1 represents the reference relations among objects that are generated during an execution of a Pac-Man game. The circle and rectangle shapes represent an instance object and static object, respectively. An edge represents a reference direction between objects via a field. In a Pac-Man game, the player strolls to collect gems and attempts to avoid colliding with ghosts on the map. The map object holds blocks and ghosts via vector objects. The ghosts and player have some states (e.g., *NormalState*, *DeathState*). The player object is a singleton and is referenced from the static map instance. At times, the block and ghost instances in the vectors are iterated by iterators for some calculations.

3.1 Pruning Temporaries by Reference Escape Analysis

In an object-oriented system, an object refers to other objects (including itself) to send messages to those objects. The means of referencing other objects are categorized into three types: (1) via a field, (2) via a local variable, and (3) via a return value of a method (e.g., a chain of method invocations). Temporary objects are destroyed without being stored in any field. We detect those temporary objects by reference escape analysis (REA).

In REA, we define following three escape states:

- ***GlobalEscape*:** An object is stored in a class field.
- ***ReferenceEscape*:** An object is stored in an instance field.

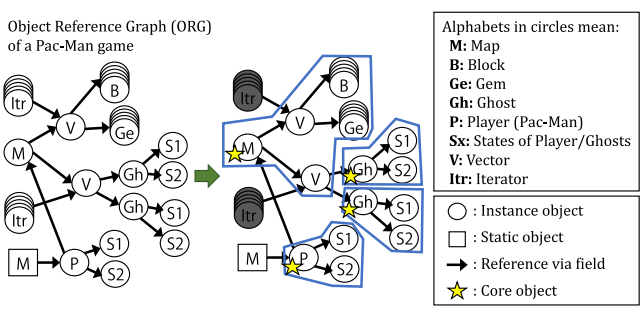

Fig. 1 Running example (Pac-Man game example)

Algorithm 1 Reference Escape Analysis

Input: Execution Trace: $ET = \langle b_1, b_2, \ldots, b_n \rangle$
Output: Escape States of Objects

for all $b_i \in ET$ **do**
  $O_{\text{referenced}} \leftarrow$ all the static objects referenced at b_i
  for all $o \in O_{\text{referenced}}$ **do**
    $EscapeState(o) \leftarrow$ "GlobalEscape"
for all $b_i \in ET$ **do**
  if b_i is ConstructorEntry **then**
    $o_{\text{created}} \leftarrow$ an object created at b_i
    $EscapeState(o_{\text{created}}) \leftarrow$ "Captured"
  else if b_i is VariableDefinition **then**
    $f \leftarrow$ the field assigned at b_i
    $o_{\text{owner}} \leftarrow$ the owner object of f
    $o_{\text{assigned}} \leftarrow$ the value assigned to f
    if $EscapeState(o_{\text{owner}})$ == "GlobalEscape" **then**
      $EscapeState(o_{\text{assigned}}) \leftarrow$ "GlobalEscape"
    else
      $EscapeState(o_{\text{assigned}}) \leftarrow$ "ReferenceEscape"

- ***Captured*:** A state other than those listed above.

REA assigns one of the above states to each object by examining the variable definition and reference events in an execution trace.

In the following, we assume that an execution trace is represented in a form of an event sequence based on the behavior model (B-model) proposed by Noda et al. [25]. B-model represents a behavior of an object-oriented system. B-model consists of event elements, such as *ConstructorEntry* / *ConstructorExit* events, which represent "entry into a constructor" / "exit from a constructor," respectively, and *VariableDefinition* / *VariableReference* events, which denote that "a value is assigned to a variable" / "a value is read from a variable," respectively. An execution trace can be represented in the form $\langle b_1, b_2, \ldots, b_n \rangle$, where b_i is an event element in B-model.

Algorithm 1 shows our REA algorithm. For each static object, we initialize the escape state of a static object to "GlobalEscape." When an object is instantiated, we initialize the escape state of the object to "Captured." After that, we examine *VariableDefinition* events in an execution trace and update escape states of objects.

After REA, we decide whether each object is temporary based on its escape status. Obviously, objects marked as "Captured" are the first candidates of temporaries; however, there exist two types of complicated situations such that we cannot decide whether each object is temporary based solely on its escape state. First, we should not conclude that all objects marked as "Captured" are temporaries. For instance, an object created in the *main* method and that becomes a *root* of successive procedures is not likely to be stored in any field; however, it is not a temporary object. Second, if temporary objects have mutual or cyclic references, those objects are marked as "Reference Escape." Thus, simply pruning all the objects marked as "Captured" is not appropriate for temporaries removal.

To address those situations, we estimate and utilize the lifetimes of objects by analyzing the reference timings. To avoid the first situation, we do not treat long-lived objects as temporaries. As for the second situation, we prune objects that are marked as "Reference Escape" but are short-lived.

Algorithm 2 Lifetime Analysis

Input: Execution Trace: $ET = \langle b_1, b_2, \ldots, b_n \rangle$
Output: Lifetimes of Objects

for all $b_i \in ET$ **do**
  if b_i is ConstructorEntry **then**
    $o_{\text{created}} \leftarrow$ the object created at b_i
    $CreatedAt(o_{\text{created}}) \leftarrow i$
  else
    for all o_k referenced at b_i **do**
      $LastAccessedAt(o_k) \leftarrow i$
for all object o_s **do**
  $Lifetime(o_s) \leftarrow LastAccessedAt(o_s) - CreatedAt(o_s)$

Algorithm 2 shows our lifetime estimation algorithm. We handle the period from object instantiation to the last reference as the object lifetime. Note that we calculate an approximate lifetime based on reference timings, and the approximated lifetime is different from the actual one treated in a garbage collection system. However, since our purpose in estimating the lifetime is to determine whether an object is important for program comprehension, we do not require an accurate lifetime; the approximated value is sufficient for our purpose.

Consequently, if an object o_i satisfies the following condition, we conclude o_i is a temporary and prune it.

$$\begin{aligned}
&(EscapeState(o_i) == \text{"Captured"} \\
&\qquad \wedge Lifetime(o_i) < Lifetime_{\max}(O) \cdot L_{t\text{-long}}) \\
&\vee (EscapeState(o_i) == \text{"ReferenceEscape"} \\
&\qquad \wedge Lifetime(o_i) < Lifetime_{\max}(O) \cdot L_{t\text{-short}})
\end{aligned}$$

Here, O is a set of all objects ($O = \{o_i \mid 1 \leq i \leq n\}$). $L_{t\text{-long}}$ and $L_{t\text{-short}}$ are threshold factors for deciding whether an object is long-lived, short-lived, or not. By default, we set $L_{t\text{-long}}$ and $L_{t\text{-short}}$ as 0.7 and 0.03, respectively. These settings are based on the result of our experiment described in Sect. 5.3.3.

Note that both Algorithm 1 and Algorithm 2 analyze the events of *VariableReference*/*Definition* of fields, *MethodEntry*/*Exit*, and *ConstructorEntry*/*Exit*; we need to weave logging codes to detect those events. Other events in the B-model (e.g., *VariableReference*/*Definition* of local variables, *LoopStart*/*End*, etc.) are not necessary for our technique; we do not record those events.

In Fig. 1, iterator instances and a non-static map object are marked as "Captured," while a static map object and the player object are "GlobalEscape." The other objects are "ReferenceEscape." Though a non-static map object is "Captured," the map object is not reported as a temporary on account of its long lifetime. Meanwhile, iterator objects are reported as temporaries and pruned. The pruned objects are blacked-out in the right of Fig. 1.

3.2 Importance Estimation by Analyzing Dynamic Properties

To find core objects, we estimate the importance of each non-temporary object by analyzing its dynamic properties: access and method-invocation frequencies. An object, which plays an important role in a system, is expected to be heavily accessed and receive many messages from other objects. Thus, we consider an important object to be one that has high access and method-invocation frequencies. We define an importance estimation formula as follows.

$$Importance(o_i) = \frac{w_w + w_r + w_{mi}}{\frac{w_w}{WF(o_i)} + \frac{w_r}{RF(o_i)} + \frac{w_{mi}}{MIF(o_i)}}$$

$Importance(o_i)$ is a weighted harmonic mean of the three frequencies: write access frequency $WF(o_i)$, read access frequency $RF(o_i)$, and method-invocation frequency $MIF(o_i)$. By default, we set w_w, w_r, and w_{mi} as 10, 5, and 1, respectively, based on the result of our experiment described in Sect. 5.3.3.

$WF(o_i)$, $RF(o_i)$, and $MIF(o_i)$ are defined as follows.

$$\begin{aligned}
WC(o_i) &= \text{(the count of write accesses to } o_i\text{)} \\
RC(o_i) &= \text{(the count of read accesses to } o_i\text{)} \\
MIC(o_i) &= \text{(the count of method invocations} \\
&\qquad \text{whose receiver is } o_i\text{)} \\
WF(o_i) &= \frac{WC(o_i) - WC_{\min}(O)}{WC_{\max}(O) - WC_{\min}(O)} \\
RF(o_i) &= \frac{RC(o_i) - RC_{\min}(O)}{RC_{\max}(O) - RC_{\min}(O)} \\
MIF(o_i) &= \frac{MIC(o_i) - MIC_{\min}(O)}{MIC_{\max}(O) - MIC_{\min}(O)}
\end{aligned}$$

$WF(o_i)$, $RF(o_i)$, and $MIF(o_i)$ are unity-based normalized values of $WC(o_i)$, $RC(o_i)$, and $MIC(o_i)$, respectively. Because the ranges of $WC(o_i)$, $RC(o_i)$, and $MIC(o_i)$ are greatly different each other, the unity-based normalization is necessary. Note that if either $WF(o_i)$, $RF(o_i)$, or $MIF(o_i)$ is zero, then $Importance(o_i) = 0$. If the maximum count is equal to the minimum count, then the frequency value is one (e.g., if $WC_{\max}(O) = WC_{\min}(O)$, then $WF(o_i) = 1$).

Consequently, if the importance value of an object is greater than the threshold I_t, we determine the object is a core object.

In Fig. 1, a non-static map, player, and ghost objects are heavily accessed and receive many messages; thus, those are recognized as the core objects. Those objects indeed play key roles in the Pac-Man game. A star symbol in the right graph in Fig. 1 indicates that the object is a core object.

4. Object Grouping and Visualization of Intergroup Interactions

In this section, we briefly describe how to visualize only core objects related behavior. After importance estimation, to visualize interactions related to core objects, we exploit the object grouping and visualization algorithm proposed in our previous work [20].

The algorithm takes our core identification result as its input. Then, for each identified core object, it constructs an object group; each of the object groups consists of an identified core object and objects having the similar lifetimes and references from the core object. In the running example, the object reference graph is decomposed into four object groups by the algorithm, as shown on the right of Fig. 1.

After that, the algorithm visualizes only intergroup interactions between the object groups and thereby obtains a summarized version of a reverse-engineered sequence diagram. An object group is depicted as a single lifeline in the summarized sequence diagram. The horizontal size of the resulting summarized sequence diagram (i.e., #lifelines in the diagram) is equal to the number of identified core objects. (Note that this number is also equal to the number of object groups.)

5. Experiment

We have implemented our technique as a tool. Our tool records runtime information based on the B-model. It then performs trace summarization, as described in the previous section. To weave logging codes into the target system, our tool uses *SELogger* which is a part of *REMViewer* [35]. *SELogger* takes *.class files as its input, then performs byte-code processing to insert logging codes into the *.class files.

We evaluated our technique on traces of various open-source software systems. We chose the utilityhood-based trace summarization proposed by Hamou-Lhadj et al. [9], which is the most promising approach for identifying core-behaviors among the existing techniques, as the baseline technique.

5.1 Research Questions and Evaluation Approaches

> ***RQ*$_1$:** Compared with the baseline technique, how effective is our approach in terms of the horizontal reduction of a reverse-engineered sequence diagram?

Motivation: Reducing the horizontal size of a reverse-engineered sequence diagram is our primary concern and objective. In RQ_1, we investigate the effectiveness of our proposed technique in terms of the performance of the horizontal reduction of a reverse-engineered sequence diagram.
Evaluation Approach: In both of the baseline and our technique, the size of the resulting summarized sequence diagram varies according to the thresholds that decide whether each object (or method) is important. The baseline technique deletes objects that did not receive any methods whose utilityhood values are smaller than the utilityhood threshold U_t. Our technique removes objects whose importance values are smaller than the importance threshold I_t.

Table 1 Subject systems and execution scenarios

Project	Ver.	KLOC	Execution scenario
jpacman [1]	rev.53	6.0	launch the application; start a new game; move the Pac-Man to the right; have the Pac-Man obtain a power cookie and change its state into the power state; quit the application
wro4j [2]	1.7.7	34.0	execute the *wro4j-runner* with specifying test resources (*.js and *.css files) as target files, 'cssMin' as a pre-processor, and 'jsMin' as a post-processor
JHotDraw [3]	7.6	138.9	launch a demo application; draw a rectangle; draw a rounded-rectangle on the right side; quit the application
jEdit [4]	5.3.0	186.7	launch the application; type some sentences including line breaks; quit the application
JMeter [5]	3_1	187.7	execute the application from the command line (Non-GUI mode) with the following settings: sending HTTP requests to a web page of a university; #threads=5, #ramp-up=2, and #loop=2; saving the results into report files
Ant [6]	1.9.8	224.3	build Ant itself (i.e., execute the 'build' task defined in the build.xml)

[1] `http://code.google.com/p/jpacman/` [2] `http://wro4j.github.io/wro4j/` [3] `http://www.jhotdraw.org/` [4] `http://www.jedit.org/`
[5] `http://jmeter.apache.org/` [6] `http://ant.apache.org/`

For each subject system, we define the ground truth of core classes that are important to comprehend the design overview of the system. Varying the thresholds U_t and I_t, we evaluate the trade-off relationship between the horizontal size of the resulting sequence diagram and the coverage of the ground truth (i.e., what percentage of the ground truth classes are contained in the resulting diagram). We denote the coverage by C_{GT}. The coverage C_{GT} indicates the usefulness of the resulting sequence diagram to comprehend the design overview of a subject system. To be able to generate a smaller size of a summarized sequence diagram with higher coverage C_{GT} means the summarization technique is more effective.

> ***RQ*$_2$:** Do both of the temporaries removal and the non-core objects elimination contribute to the performance of the horizontal reduction of a reverse-engineered sequence diagram?

Motivation: Our technique consists of two types of reduction steps: pruning temporaries and eliminating low-importance objects. In RQ_2, we would like to further investigate the individual contribution of the two reduction steps to the performance of our technique.
Evaluation Approach: For each step, we investigate the performance of reducing the number of objects since the number of objects directly affects the horizontal size of the resulting sequence diagram.

> ***RQ*$_3$:** What is the effect of varying the tunable parameters of our technique on the reduction performance?

Motivation: Our technique has two types of tunable parameters: a set of weights used in our importance estimation formula; a set of lifetime thresholds for detecting long-lived and short-lived objects. In RQ_3, we would like to investigate the effect of varying the parameters on the reduction performance.
Evaluation Approach: We analyze the sensitivity of the performance when varying the two types of parameters individually: first, fixing the lifetime threshold values, we vary the weights and investigate the reduction performance; second, we fix the weight values and vary the lifetime thresholds.

> ***RQ*$_4$:** How much runtime overhead does our technique involve?

Motivation: Our proposed technique need to weave logging codes, which causes a runtime overhead. In terms of practicality, it is highly important to be able to analyze with a small runtime overhead. In this research question, we investigate the overhead imposed by our technique.
Evaluation Approach: We measure and compare the execution times of two states: with and without weaving. We evaluate the overhead by comparing with those of recent scalable dynamic analysis techniques.

5.2 Experimental Setup

5.2.1 Subject Systems and Execution Scenarios

The subject systems and execution scenarios are as shown in Table 1. For each subject, we selected a representative execution scenario.

5.2.2 Ground Truths

For each execution scenario, we predefine a ground truth of core identification. Extracting core classes, which are important to comprehend the design overview, from execution scenarios, source code comments, and documents, we define the extracted classes as the ground truth *GT*. Table 2 shows the ground truth *GT* for each subject system.

For the *jpacman* case, we consider that the actors and keywords that appear in the description of the execution scenario (i.e., player, ghost, power cookie, and state) are important for program comprehension. In addition, we also consider the *SimpleLevel* class as a core class because the *SimpleLevel* controls many key parts of the game system (e.g., sends periodic update/render-messages to the game entities according to the FPS).

As for *JHotDraw*, since the documentation and source code comments written by its developers provide the description of the important classes for its architecture, we define the important classes as our ground truth.

For other subject systems, we use the definition of important classes provided by Şora et al. [31]. Şora et al. extracted the classes that are important to comprehend design

Table 2 Core classes that are important to comprehend the design overview

Project	Core classes (i.e., ground truth *GT*)
jpacman	*Player, Ghost, EatGem, Player$NormalState, Player$PowerState, Ghost$NormalState, Ghost$EatableState, SimpleLevel*
wro4j	*WroModel, WroModelFactory, Group, Resource, WroManager, WroManagerFactory, ResourcePreProcessor, ResourcePostProcessor, UriLocator, UriLocatorFactory, ResourceType*
JHotDraw	*DrawingView, Drawing, DrawingEditor, Tool, Figure*
JEdit	*jEdit, View, EditPane, Buffer, JEditTextArea, Log*
JMeter	*JMeterEngine, JMeterThread, Sampler, SampleResult, TestCompiler, TestElement, TestStateListener, TestIterationListener, TestPlan, ThreadGroup*
Ant	*Project, Target, UnknownElement, RuntimeConfigurable, Task, IntrospectionHelper, ProjectHelper2*

overviews from the documentation or developer tutorials of the subject systems. Note that Şora et al. also defined the important classes for JHotDraw; however, we re-extract important classes from the source code comments because the definition by Şora et al. was for an older version of JHotDraw.

Of the sets of important classes extracted from execution scenarios, documents, and the definitions by Şora et al., we exclude some of the classes because those are not used in our execution scenario. For example, although the *Handle* interface is described as important in the source code comments in JHotDraw, our execution scenario has no operations using the *Handle* interface (e.g., resizing); thus, we exclude the *Handle* from the ground truth.

While most of the core classes in *GT* are abstract classes or interfaces, classes of the objects appeared in a reverse-engineered sequence diagram are sub-classes or implementation-classes of the core classes in *GT*. From a perspective of comprehending a behavioral aspect with a reverse-engineered sequence diagram, identifying not abstract classes or interfaces but concrete classes are important. Therefore, when calculating the coverage C_{GT}, for each class c defined in Table 2, we consider the class c is covered if at least one object of the sub-type of c (including itself) is contained in the resulting sequence diagram. Formally, the coverage C_{GT} is calculated as follows:

$$C_{\mathrm{GT}} = \frac{\Sigma_{c \in GT}\, \mathrm{isCovered}(c, SD)}{\mid GT \mid}$$

If at least one object of the sub-type of c (including itself) is contained in the resulting sequence diagram SD, $\mathrm{isCovered}(c, SD) = 1$; otherwise, $\mathrm{isCovered}(c, SD) = 0$.

5.2.3 Weaving Extent

We aim to help developer comprehend the behavior specific to the target domain. For this reason, we weave logging codes only into classes defined in the target application. We do not insert logging codes into libraries (i.e., our tool passes only *.class files defined in the target application to *SELogger*). This weaving condition is commonly and widely used in the area of dynamic analysis and is realistic in terms of avoiding a heavy logging overhead.

Of the library codes, we exceptionally weave logging codes only into Java collection libraries. Our proposed method analyzes the object reference relations. Because developers often use the collection libraries to manipulate collection data, not analyzing the collection libraries may significantly reduce the accuracy of the analysis. To avoid this situation, we weave logging codes only into the Java collection libraries; libraries except for the collection libraries are not our logging targets. Note that objects of collection libraries are used only for analyzing reference relations. Since those objects of collection libraries are not core objects in the target application, we ignore those objects when importance estimation and visualization.

From a technical point of view, normally, *SELogger* does not weave logging codes into the runtime-core classes of Java (i.e., classes contained in *rt.jar*) because *SELogger* needs some of the non-woven classes in the *rt.jar* to write an execution trace onto a disk. To weave logging codes into the Java collection libraries, we have collection classes within our tool that are independent on the *rt.jar*, then we pass the collection classes within out tool to *SELogger* explicitly; thereby, we achieve that the target application uses the woven collection libraries and *SELogger* uses the non-woven collection libraries.

5.2.4 Settings of Tunable Parameters

Our technique has two types of tunable parameters as described in the description of RQ_3 (see Sect. 5.1). In all the experiment except for RQ_3, we use the default values of them. In the experiment for RQ_3, we investigate the effect of varying the parameters on the reduction performance.

5.2.5 Settings of Baseline Technique

For comparison, we apply the baseline technique [9] to subject systems. A utilityhood value is calculated for each method m by the following formula:

$$U(m) = \frac{\#\text{fan-in}(m)}{N} \times \frac{\log(N/(\#\text{fan-out}(m) + 1))}{\log(N)}$$

Here, N stands for the number of methods in a call graph. We calculate the utilityhood values with the same settings as described in the paper of Hamou-Lhadj et al. [9]: Polymorphic calls are resolved by Rapid Type Analysis [36] during a call graph construction to measure the fan-in and fan-out; The utilityhood values are calculated for only non-private methods (excluding constructors and accessor methods) invoked in the execution scenario. Note that although Hamou-Lhadj et al. ignore methods of inner classes, we do not ignore those. This is because some of the core classes shown

in Table 2 are implemented as inner classes.

5.3 Results

5.3.1 Answer to RQ_1

The recorded runtime information is shown in Table 3. The numbers of messages and objects affect the size of the reverse-engineered sequence diagram. The number of lifelines (i.e., the horizontal size) of the diagram is equal to the number of objects.

When varying the thresholds I_t and U_t that decide whether each object (method) is important, the trade-off relationships between the horizontal size of the resulting diagram (i.e., #lifelines) and the coverage C_{GT} are as shown in Fig. 2. The dashed line in Fig. 2 is a random classifier that randomly classifies whether each object is important.

As shown in Fig. 2, for all the subject systems, our technique reaches $C_{GT} = 1$ with a much smaller number of lifelines, compared with the baseline. The numbers of lifelines and the compression ratios at $C_{GT} = 1$ are as shown in Table 4. Table 4 shows the compression ratio of our technique is much higher than that of the baseline: our compression ratio ranges 14.5–390.9 (134.6 on average), whereas that of the baseline ranges 3.6–22.0 (11.5 on average).

Our technique could achieve a significant reduction of the horizontal size of a reverse-engineered sequence diagram, while retaining the core objects of *GT* in the diagram. From the result described above, we confirm that our technique is more effective in terms of the horizontal reduction of a reverse-engineered sequence diagram, compared with the baseline technique.

Moreover, there is another finding. In the baseline technique, there are many ties in the utilityhood-based ranking because many methods have the same utilityhood values (i.e., there exist many methods having the same #fan-in/out). Meanwhile, in our proposed technique, the number of ties in the importance-based ranking is relatively small.

Table 3 Recorded runtime information

Project	Total events	Messages	Loaded classes	Objects[1]
jpacman	24,415,941	10,709,679	57	1,289
wro4j	327,214	154,928	295	1,504
JHotDraw	436,926	219,858	276	2,736
jEdit	2,529,506	908,104	497	7,030
JMeter	608,182	320,258	372	5,245
Ant	44,864,241	23,176,454	271	50,828

[1] the number does not include objects of collection libraries

Table 4 The compression ratio at $C_{GT} = 1$

	ours		baseline	
Project	#lifelines[1]	Comp.[2]	#lifelines[1]	Comp.[2]
jpacman	27	47.7	148	8.7
wro4j	104	14.5	421	3.6
JHotDraw	7	390.9	313	8.7
jEdit	71	99.0	492	14.3
JMeter	222	23.6	238	22.0
Ant	219	232.1	4,378	11.6
Average	108	134.6	998	11.5

[1] #lifelines in the resulting sequence diagram at $C_{GT} = 1$

[2] Compression ratio is calculated by (#lifelines in the original diagram)/(#lifelines in the resulting diagram). Note that #lifelines in the original diagram is equal to #objects in Table 3.

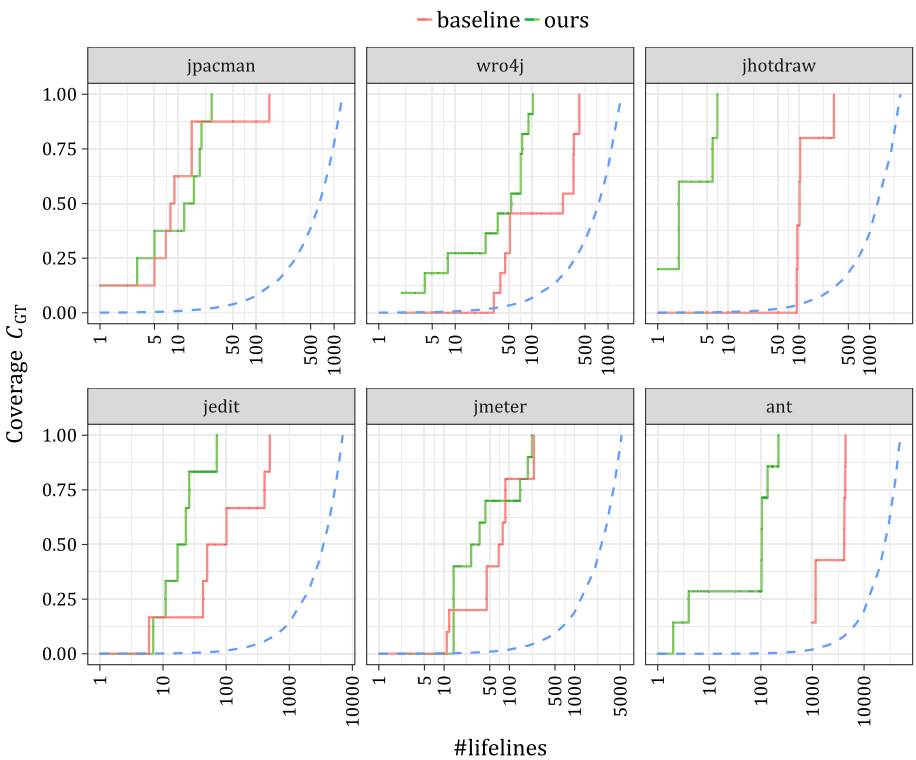

Fig. 2 The reduction performance

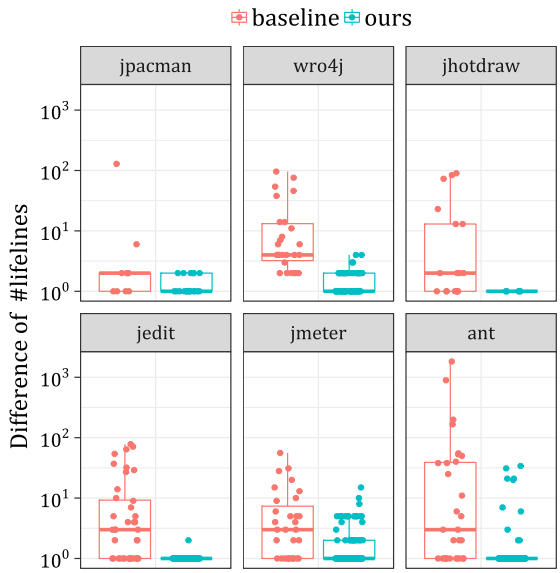

Fig. 3 The distribution of the difference of #lifelines between two adjacent threshold values (i.e., the difference between #lifelines($U_t = U_{M_i}$) and #lifelines($U_t = U_{M_{i+1}}$); the difference between #lifelines($I_t = I_{O_j}$) and #lifelines($I_t = I_{O_{j+1}}$))

This fact affects the degree of the change of #lifelines when varying the threshold to raise/reduce the abstraction level of the diagram. We elaborate this finding in the following paragraphs.

Let $U_{M_1}, U_{M_2}, \ldots, U_{M_M} (s.t., U_{M_i} < U_{M_{i+1}})$ be all the possible values of utilityhood, and $I_{O_1}, I_{O_2}, \ldots, I_{O_N} (s.t., I_{O_j} < I_{O_{j+1}})$ be all the possible values of importance. U_{M_i} means the utilityhood value for a set of methods M_i, and I_{O_j} means the importance value for a set of objects O_j. All the methods (objects) in M_i (O_j) have the same utilityhood (importance) value. Additionally, let #lifelines($U_t = U_{M_i}$) (#lifelines($I_t = I_{O_j}$)) be the #lifelines such that $U_t = U_{M_i}$ ($I_t = I_{O_j}$).

Figure 3 shows the distribution of the difference of #lifelines between two adjacent threshold values (i.e., the difference between #lifelines($U_t = U_{M_i}$) and #lifelines($U_t = U_{M_{i+1}}$); the difference between #lifelines($I_t = I_{O_j}$) and #lifelines($I_t = I_{O_{j+1}}$)). From the Fig. 3, moving the threshold values U_t, I_t to the adjacent values (i.e., from $U_t = M_i$ to $U_t = M_{i+1}$; from $I_t = O_j$ to $I_t = O_{j+1}$), the degree of the change of the #lifelines is much smaller in our technique, compared with the baseline (the #lifelines is changed by 1–34 in our technique and by 1-1826 in the baseline).

Changing the threshold U_t causes a drastic change of the abstraction level of the diagram; it is difficult to change the threshold value flexibly in the baseline technique. Meanwhile, the degree of the change of #lifelines when varying the threshold I_t is much smaller. This indicates that our technique enables a developer to choose the threshold value more flexibly depending on the kind of the maintenance task that the developer undertakes. (e.g., It is appropriate to choose the higher-importance point as the threshold value in the earlier stage of program comprehension. Along with increasing the degree of understanding, the developer can change the threshold value to the lower-importance point.)

Table 5 Results of pruning temporaries

Project	#temporaries	#non-temporaries	Comp.[1]
jpacman	733	556	2.3
wro4j	1,120	384	3.9
JHotDraw	2,157	579	4.7
jEdit	5,765	1,265	5.6
JMeter	914	4,331	1.2
Ant	48,055	2,773	18.3
Average	9,791	1,648	6.0

[1] Compression ratio is calculated by #(all objects)/#(non-temporaries)

> Our technique is more effective in terms of the horizontal reduction of a reverse-engineered sequence diagram. The compression ratio of our technique was much higher than that of the baseline: our compression ratio was 134.6 on average, whereas that of the baseline was 11.5.
> Moreover, our technique is capable of changing the abstraction level more flexibly, compared with the baseline technique.

5.3.2 Answer to RQ_2

Table 5 shows the number of (non-)temporaries and the compression ratios by pruning temporaries. Temporaries removal reduced a significant number of objects: on average, 9,791 objects were removed and the compression ratio was 6.0. On the other hand, the reduction performance of eliminating the non-core objects is shown in Table 4. The non-core objects elimination further improves the compression ratios from 6.0 up to 134.6 on average. Both of the two reduction steps reduce a great number of objects as shown in Table 5 and Table 4; therefore, both of the steps contribute to the reduction performance.

To investigate the individual contribution of eliminating the non-core objects without temporaries removal, we calculated the importance values for all objects including temporaries. By manual samplings of the calculation result, we found some temporary objects had high access frequencies and thereby temporaries could be noise in our importance estimation technique: If developers decrease the threshold I_t because the desired behaviors are not contained in the resulting diagram, a lot of temporaries will tend to appear in the diagram, which greatly impairs the usefulness of the diagram. Therefore, to enable developers to change the abstraction level of the resulting diagram flexibly, it is necessary to combine the temporaries removal and the non-core objects elimination.

> Both pruning temporaries and eliminating the non-core objects contribute to the reduction performance. Both of the reduction steps are essential to enable developers to flexibly change the abstraction level of the resulting sequence diagram.

5.3.3 Answer to RQ_3

First, fixing the lifetime thresholds $L_{t\text{-long}}$ and $L_{t\text{-short}}$ as their

default values, we vary the weights w_w, w_r, and w_{mi}.

The results of varying the weights w_w, w_r, and w_{mi} are shown in Fig. A· 1, Fig. A· 2, and Fig. A· 3, resp. Enlarging the weight w_w, the reduction performance rises for four projects and decreases for two projects (Fig. A· 1). Increasing the weight w_r makes the reduction performance better for two projects and worse for two projects (Fig. A· 2). As for w_{mi}, increasing the weight makes the reduction performance worse for two projects and has few positive effects; there are almost no performance improvements by increasing the w_{mi} (Fig. A· 3).

Enlarging the weight for write access w_w leads a good effect on the reduction performance. Write accesses reflect state changes of objects. Core objects that play important roles in a system are expected to tend to change their states frequently. Thus, the good effect on core identification by increasing the weight for write access is a plausible result.

From these observations, fixing the w_w as 10 (i.e., high value) and the w_{mi} as 1 (i.e., low value), we varied the w_r and tried to seek for the best weight value. We obtained the best performance with $(w_w, w_r, w_{mi}) = (10, 5, 1)$; thus, we chose those values as the default values of the weights.

Second, Fig. A· 4 shows the effect of varying the lifetime threshold for short-lived objects $L_{t\text{-short}}$ with fixing other parameters as default values. From Fig. A· 4, we can see that removing short-lived objects whose lifetimes are less than 3% of the whole execution period leads a good effect on the reduction performance. We consider removing short-lived objects reduces noise in our core identification technique and contributes to the improvement of the performance. Since the setting $L_{t\text{-short}} = 0.3$ leads the best performance, we set the default value of $L_{t\text{-short}}$ as 0.3.

Finally, fixing the weights and the lifetime threshold $L_{t\text{-short}}$ as their default values, we investigated the effect of varying the threshold $L_{t\text{-long}}$ from 0.7 up to 1.0 at 0.1 intervals. We could not see any notable effect of varying $L_{t\text{-long}}$. For five projects of the total six projects, the reduction performance did not vary. For only one project, the reduction performance varied; however, the change was very slight and ignorable (#lifelines at $C_{\mathrm{GT}} = 1$ varied by only 1). We consider this result is attributed to the situation such that there were no important objects that were the *root*s of successive procedures as described in Sect. 3.1. Although there was no need to prevent long-lived captured objects from being removed by our REA for our subjects, the prevention of removing long-lived captured objects is intrinsically necessary. As no effect was observed, we just chose the first value in our experiment (i.e., 0.7) as the default value of $L_{t\text{-long}}$.

Enlarging the weight for write access leads the improvement of the reduction performance. To attach great importance to state changes of objects is expected to be helpful for core object identification.
Removing short-lived objects whose lifetimes are less than 3% of the whole execution period reduces noise in our core identification and improves the reduction performance. We could not see any notable effect of varying the $L_{t\text{-long}}$ for our subjects; however, preventing long-lived captured objects from being removed is intrinsically necessary.

5.3.4 Answer to RQ_4

Table 6 shows the runtime overhead. We measured the runtime overhead 5 times for each execution scenario, and calculated the average overhead. The 'Base' and 'With logging' columns show the execution time without and with logging codes, resp. We used an Intel Xeon E5-2620 v4 2.10GHz machine and assigned 16GB of RAM to the heap of the Java VM. We set the options of *SELogger* as follows: using four background threads for writing a trace data into a disk; recording the trace data in an uncompressed format.

Our technique involved 39.9%–344.1% runtime overhead (167.6% on average). This overhead is relatively small compared with the recent scalable dynamic analysis techniques. For example, the feedback-directed instrumentation technique for computing crash paths [37] imposed 100%–800% overhead in most cases, and the record and replay system [38] involved 480% overhead on average.

Developers need to execute an instrumented application only once for behavioral visualization. Thus, in many cases, the overhead of our technique is expected to be acceptable not in a production phase but in a development phase.

Our technique imposes 167.6% runtime overhead on average, which is relatively small compared with the recent dynamic scalable analysis techniques. In many cases, this overhead is expected to be acceptable for practical use.

Finally, to promote the understandings of readers, we show an example of a part of the resulting (summarized) sequence diagram of the *jpacman* case in Fig. 4.

From the diagram shown in Fig. 4, for example, developers can understand the following key behavior. A *SimpleLevel* object periodically sends *update()* messages to actor objects such as *Map*, *Player*, and *Ghost* objects. If a collision between the *Player* object and another object (e.g., a ghost object) occurs, the *onCollision()* method of *Player*'s *StateManager* is invoked. Then, the *Player* object changes its state from *NormalState* to *PowerState*. Therefore, for example, if developers want to change the behavior on a collision between the player object and other actor objects (e.g., to implement new states of the player; to move the

Table 6 Execution time

Project	Base (sec)	With logging (sec)	Overhead (%)
jpacman	9.84	13.76	39.9
wro4j	4.59	6.16	34.0
JHotDraw	6.84	22.79	233.2
jEdit	7.48	26.92	259.8
JMeter	4.97	22.06	344.1
Ant	52.91	102.91	94.5
Average	14.44	32.43	167.6

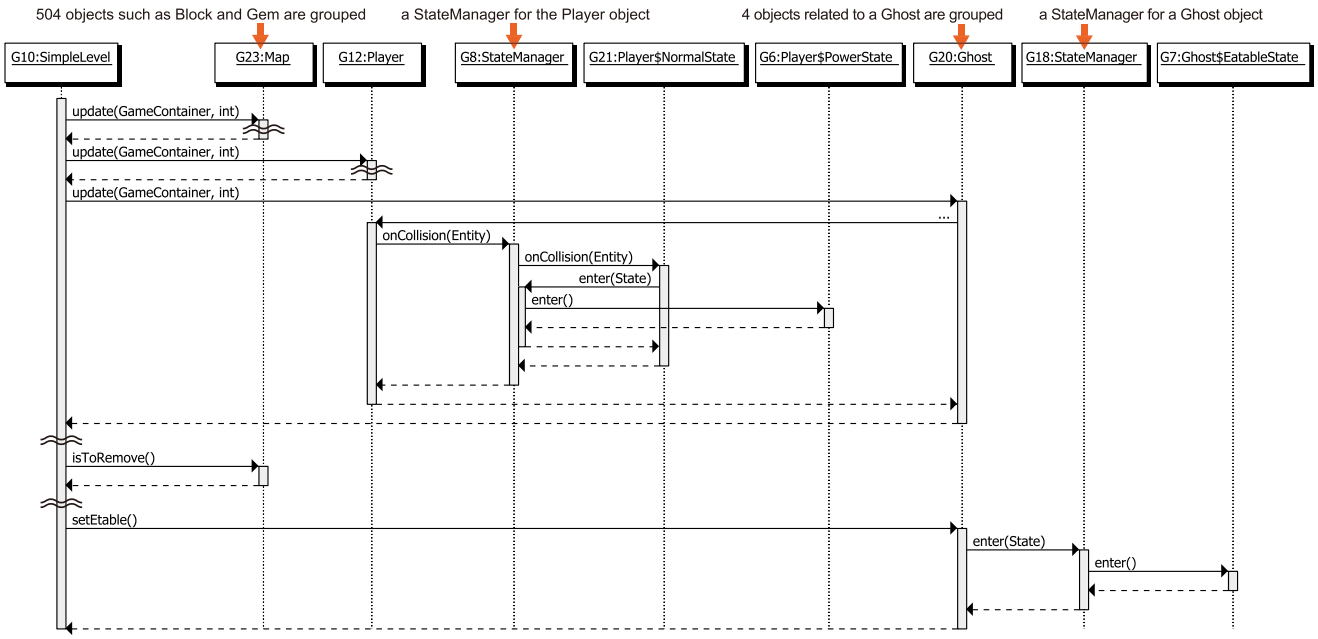

Fig. 4 A part of the resulting diagram of the *jpacman* case

player backward on the collision), developers can acquire the knowledge necessary for the modification from the resulting diagram.

6. Threats to Validity

In the experiment, each execution scenario for each application is short. However, this is not an unnatural setting; even though there exist only a long execution trace, we can easily extract a small block of interest from the long trace based on the absolute time recorded in each event in the trace. We can also use the phase detection techniques mentioned in Sect. 2 to divide the long trace.

The ground truth of the core objects for *jpacman* were defined by authors. Although authors have conducted some studies by using *jpacman* over two years and have lots of knowledge about the application, the *GT* for *jpacman* might be incorrect. However, for other subject systems, we extracted the ground truths from their documents, and our technique showed superior reduction performance compared with the baseline for all the subjects; thus, we consider the effectiveness of our technique has been confirmed.

We did not make it clear how much of developers' time were saved in actual maintenance tasks by using the resulting summarized sequence diagram, which was out of scope of this paper, since we focused on the identification of core objects in this paper. User studies need conducting in future studies to evaluate and validate the usefulness of the resulting summarized diagram (including the object grouping and visualization algorithm described in Sect. 4) in actual maintenance tasks.

7. Conclusion

Facilitating program comprehension with a reverse-engineered sequence diagram is a promising technique. However, it incurs a scalability issue. In this paper, we presented a technique to identify core objects for trace summarization by analyzing reference relations and dynamic properties. Our technique first detects and prunes temporary objects, and then ranks objects by estimated importance to identify core objects. Visualizing only interactions related to identified core objects, our technique obtains a summarized version of a reverse-engineered sequence diagram.

We evaluated our technique on traces of various open-source software systems. The results showed that our technique had superior reduction performance of a reverse-engineered sequence diagram, compared with the state-of-the-art trace summarization technique. The horizontal compression ratio of our technique was 134.6 on average, whereas that of the state-of-the-art technique was 11.5. The runtime overhead imposed by our technique was 167.6% on average, which was small and showed the practicality of our technique. Our technique can achieve a significant reduction of the horizontal size of a reverse-engineered sequence diagram with a small overhead and is expected to be a valuable tool for program comprehension.

Acknowledgments

We would like to thank an ex-colleague Tatsuya TODA for his contribution in the early stage of this work. This work was partly supported by MEXT KAKENHI Grant Numbers JP24300006, JP26280021, JP15H02683, and JP15K15973.

References

[1] C. Bennett, D. Myers, M.A. Storey, D.M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," J. Softw. Maint. Evol., vol.20, no.4, pp.291–315, 2008.

[2] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," Proc. International Conference on Automated Software Engineering, pp.443–453, 2013.
[3] M. Brünink and D.S. Rosenblum, "Mining performance specifications," Proc. International Symposium on Foundations of Software Engineering, pp.39–49, 2016.
[4] M. Pradel and T.R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," Proc. International Conference on Software Engineering, pp.288–298, 2012.
[5] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," Proc. International Symposium on Foundations of Software Engineering, pp.178–189, 2014.
[6] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," Proc. International Workshop on Principles of Software Evolution, pp.148–154, 2005.
[7] D. Myers, M.A. Storey, and M. Salois, "Utilizing debug information to compact loops in large program traces," Proc. European Conference on Software Maintenance and Reengineering, pp.41–50, 2010.
[8] Y. Watanabe, T. Ishio, and K. Inoue, "Feature-level phase detection for execution trace using object cache," Proc. International Workshop on Dynamic Analysis, pp.8–14, 2008.
[9] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," Proc. International Conference on Program Comprehension, pp.181–190, 2006.
[10] M. Srinivasan, J. Yang, and Y. Lee, "Case studies of optimized sequence diagram for program comprehension," Proc. International Conference on Program Comprehension, pp.1–4, 2016.
[11] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection," Proc. Companion of the International Conference on Software Engineering, pp.969–970, 2008.
[12] R. Sharp and A. Rountev, "Interactive exploration of uml sequence diagrams," Proc. IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp.1–6, 2005.
[13] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," IEEE Trans. Softw. Eng., vol.37, pp.341–355, 2011.
[14] D. Myers and M.A. Storey, "Using dynamic analysis to create trace-focused user interfaces for IDEs," Proc. International Symposium on Foundations of Software Engineering, pp.367–368, 2010.
[15] H. Grati, H. Sahraoui, and P. Poulin, "Extracting sequence diagrams from execution traces using interactive visualization," Proc. Working Conference on Reverse Engineering, pp.87–96, 2010.
[16] P. Dugerdil and J. Repond, "Automatic generation of abstract views for legacy software comprehension," Proc. India Software Engineering Conference, pp.23–32, 2010.
[17] T. Toda, T. Kobayashi, N. Atsumi, and K. Agusa, "Grouping objects for execution trace analysis based on design patterns," Proc. Asia-Pacific Software Engineering Conference, pp.25–30, 2013.
[18] K. Noda, T. Kobayashi, and K. Agusa, "Execution trace abstraction based on meta patterns usage," Proc. Working Conference on Reverse Engineering, pp.167–176, 2012.
[19] B. Dufour, B.G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive java applications," Proc. International Symposium on Foundations of Software Engineering, pp.59–70, 2008.
[20] K. Noda, T. Kobayashi, T. Toda, and N. Atsumi, "Identifying core objects for trace summarization using reference relations and access analysis," Proc. Annual Computer Software and Applications Conference, pp.13–22, 2017.
[21] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, and A. Shafiee, "Stratified sampling of execution traces: Execution phases serving as strata," Sci. Comput. Program., vol.78, no.8, pp.1099–1118, Aug. 2013.
[22] H. Pirzadeh and A. Hamou-Lhadj, "A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension," Proc. IEEE International Conference on Engineering of Complex Computer Systems, pp.221–230, 2011.
[23] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah, "Exploiting text mining techniques in the analysis of execution traces," Proc. IEEE International Conference on Software Maintenance, pp.223–232, 2011.
[24] S. Munakata, T. Ishio, and K. Inoue, "OGAN: Visualizing object interaction scenarios based on dynamic interaction context," Proc. International Conference on Program Comprehension, pp.283–284, 2009.
[25] K. Noda, T. Kobayashi, S. Yamamoto, M. Saeki, and K. Agusa, "Reticella : An execution trace slicing and visualization tool based on a behavior model," IEICE Trans. Inf. & Syst., vol.95, no.4, pp.959–969, 2012.
[26] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J.J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," J. Syst. Softw., vol.81, no.12, pp.2252–2268, 2008.
[27] I. Şora and D. Todinca, "Using fuzzy rules for identifying key classes in software systems," Proc. International Symposium on Applied Computational Intelligence and Informatics, pp.317–322, May 2016.
[28] F. Thung, D. Lo, M.H. Osman, and M.R.V. Chaudron, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," Proc. 22nd International Conference on Program Comprehension, pp.110–121, 2014.
[29] X. Yang, D. Lo, X. Xia, and J. Sun, "Condensing class diagrams with minimal manual labeling cost," Proc. Annual Computer Software and Applications Conference, pp.22–31, 2016.
[30] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," J. Software Maintenance and Evolution: Research and Practice, vol.20, no.6, pp.387–417, 2008.
[31] I. Şora, "Helping program comprehension of large software systems by identifying their most important classes," Proc. International Conference on Evaluation of Novel Approaches to Software Engineering Revised Selected Papers, pp.122–140, Springer International Publishing, 2016.
[32] A. Lienhard, S. Ducasse, and T. Gîrba, "Taking an object-centric view on dynamic information with object flow analysis," Comput. Lang. Syst. Struct., vol.35, no.1, pp.63–79, April 2009.
[33] R. Vanciu and M. Abi-Antoun, "Ownership object graphs with dataflow edges," Proc. Working Conference on Reverse Engineering, pp.267–276, 2012.
[34] M. Abi-Antoun, Y. Wang, E. Khalaj, A. Giang, and V. Rajlich, "Impact analysis based on a global hierarchical object graph," Proc. International Conference on Software Analysis, Evolution, and Reengineering, pp.221–230, 2015.
[35] T. Matsumura, T. Ishio, Y. Kashima, and K. Inoue, "Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java," Proc. International Conference on Program Comprehension, pp.253–257, 2014.
[36] D.F. Bacon and P.F. Sweeney, "Fast static analysis of c++ virtual function calls," Proc. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp.324–341, 1996.
[37] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller, "Feedback-directed instrumentation for deployed javascript applications," Proc. International Conference on Software Engineering, pp.899–910, 2016.
[38] J. Huang, "Scalable thread sharing analysis," Proc. International Conference on Software Engineering, pp.1097–1108, 2016.

Appendix: Result of Sensitivity Analysis

As mentioned in Sect. 5.1 RQ_3, we analyzed the sensitivity of the performance when varying the two types of tunable parameters individually. We show the resulting charts of the sensitivity analysis in the following pages.

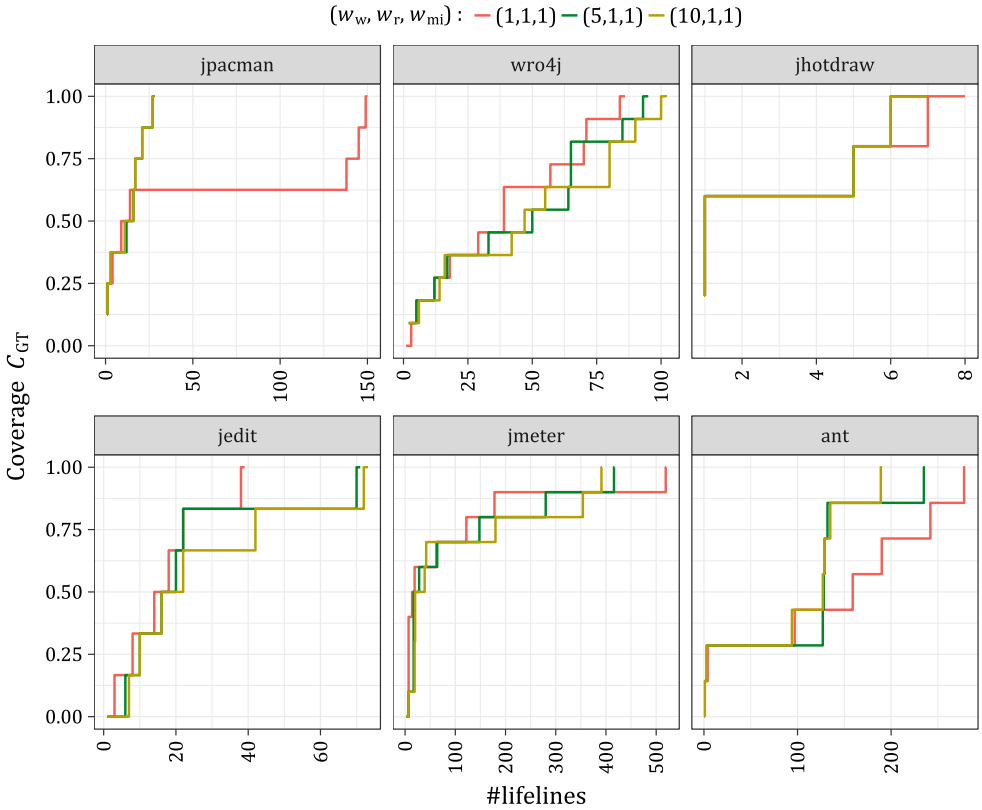

Fig. A· 1 The effect of varying the value of w_w

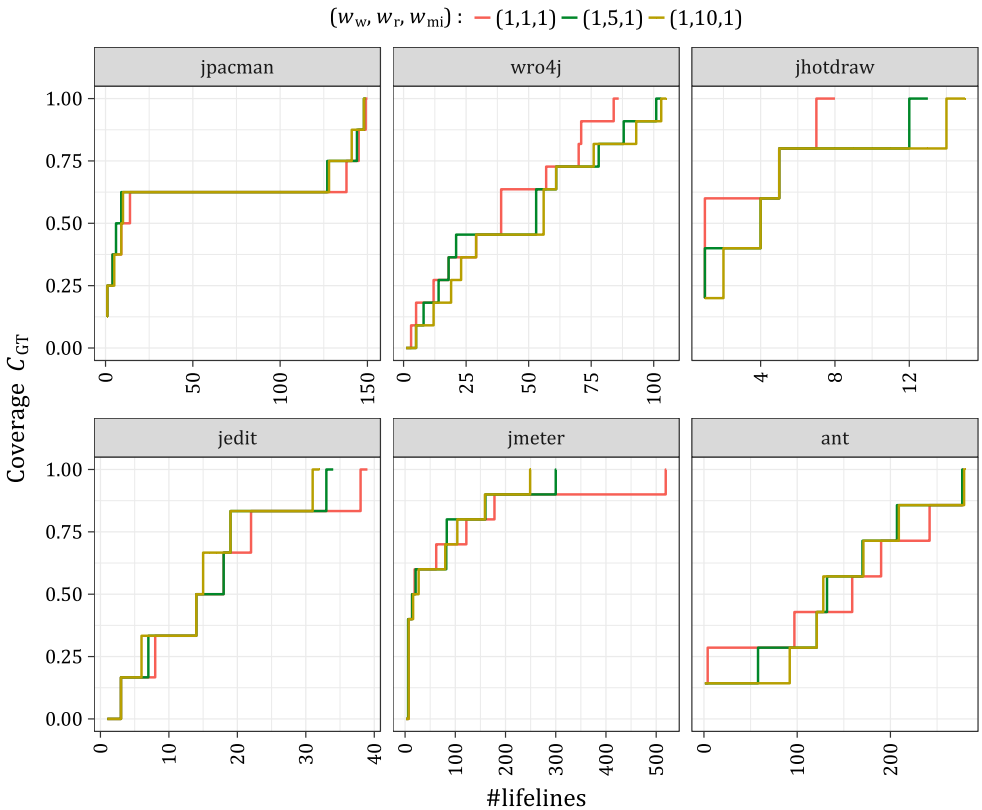

Fig. A· 2 The effect of varying the value of w_r

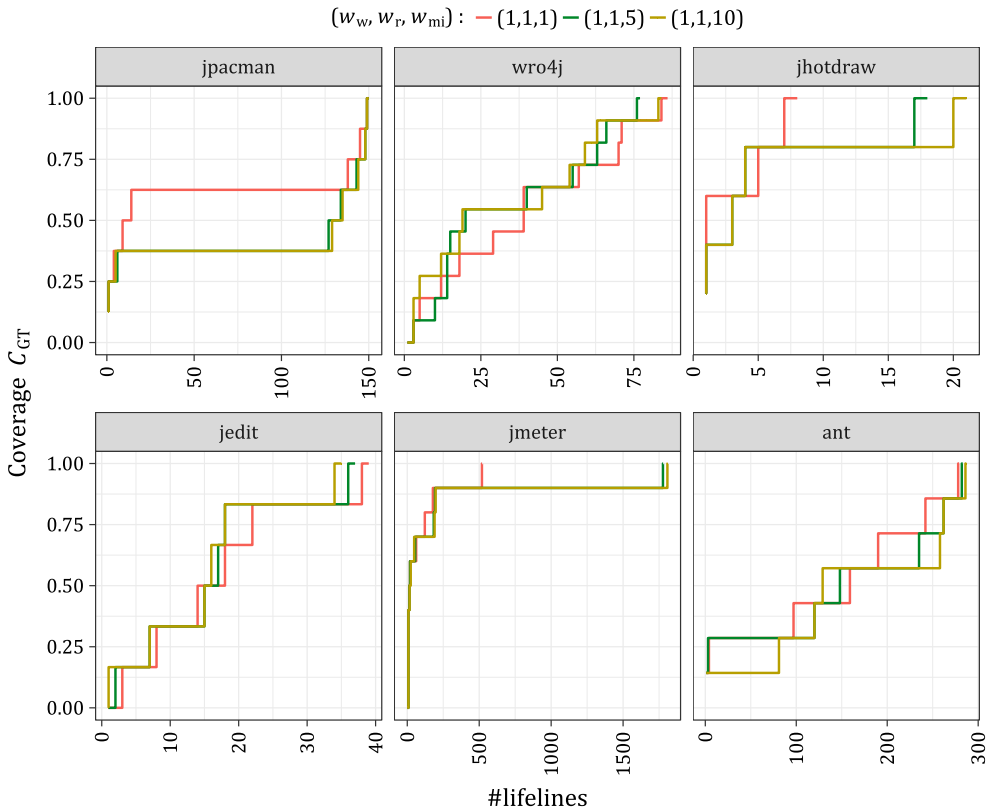

Fig. A·3 The effect of varying the value of w_{mi}

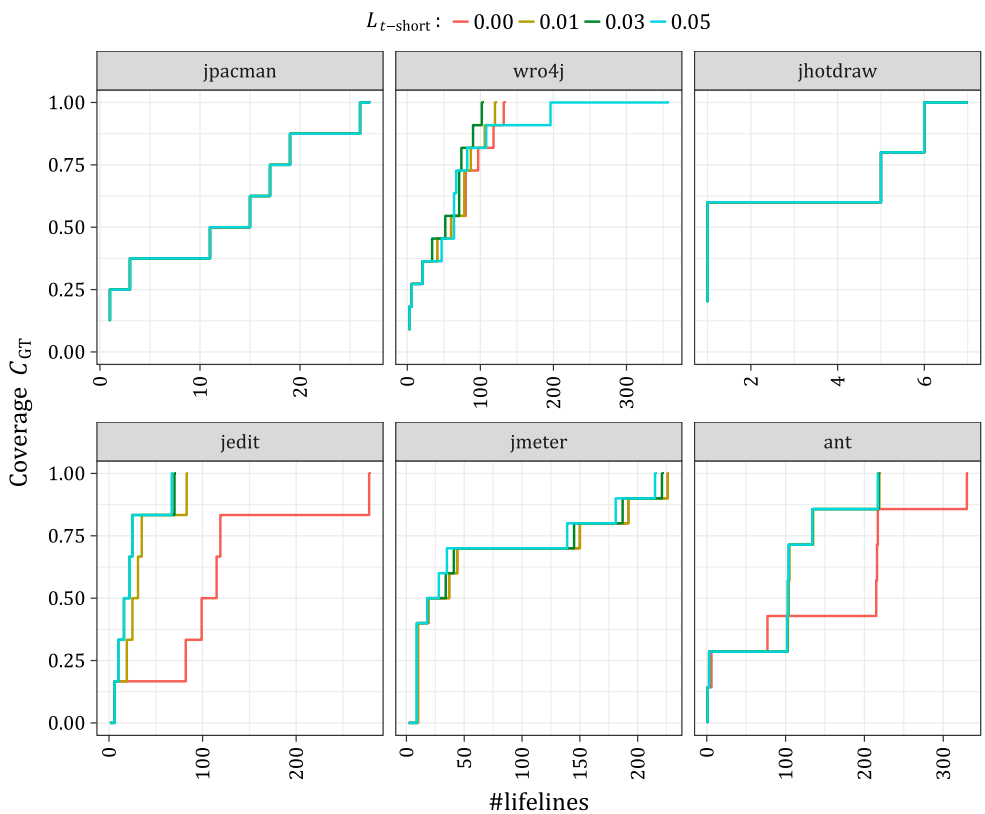

Fig. A·4 The effect of varying the value of $L_{t\text{-short}}$

Kunihiro Noda received the B.Sc. degree in engineering and M.Sc. degree in information science from the Nagoya University in 2009 and 2012, respectively. He had worked as a software engineer in an automotive company for about five years since the graduation from the Nagoya University. He is currently working toward the Ph.D. degree under the supervision of Professor Takashi KOBAYASHI at the Tokyo Institute of Technology. His research interests include program analysis, program comprehension, fault localization, and specification mining.

Takashi Kobayashi is an associate professor in Department of Computer Science, School of Computing, Tokyo Institute of Technology. His research interests include software reuse, software design, software maintenance, program analysis, software configuration management, Web-services compositions, workflow, multimedia information retrieval, and data mining. He received a B.Eng., M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 1997, 1999 and 2004, respectively. He is a member of the JSSST, IPSJ, DBSJ, IEEE-CS, and ACM.

Noritoshi Atsumi is an assistant professor of Academic Center for Computing and Media Studies, Kyoto University. He received his Doctor of Engineering degree from Nagoya University in 2007. His research interests include software reuse, software development support, software maintenance, and program analysis. He is a member of JSSST, IPSJ, IEEE-CS, and ACM.