論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Recovering a Summarized Sequence Diagram through Program Analysis
著者(和文)	野田訓広
Author(English)	Kunihiro Noda
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第11161号, 授与年月日:2019年3月26日, 学位の種別:課程博士, 審査員:小林 隆志,佐伯 元司,権藤 克彦,渡部 卓雄,林 晋平
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第11161号, Conferred date:2019/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Recovering a Summarized Sequence Diagram through Program Analysis

A dissertation presented

by

Kunihiro NODA

to

The Department of Computer Science
in partial fulfillment of the requirements
for the degree of

Doctor of Engineering

School of Computing
Tokyo Institute of Technology
Tokyo, Japan

February 2019

Contents

List of Figures

List of Tables

List of Algorithms

Summary

Program comprehension is a quite important activity in software maintenance. Software documentation is a vital source of information for program comprehension; however, it tends to be outdated or non-existent in many cases. Moreover, comprehending the behavior of an object-oriented system solely from its source code is cumbersome and error-prone due to its dynamism (e.g., dynamic bindings and reflections).

For aiding behavioral comprehension, recovering a sequence diagram from an execution trace, which reflects the actual state of a system, is a promising approach. However, owing to the massiveness of execution traces, reverse-engineered sequence diagrams are often afflicted by scalability issues.

There are many existing techniques for coping with the massiveness of reverse-engineered sequence diagrams. Most existing work focuses on compacting the vertical size of recovered diagrams or effectively exploring massive-scale diagrams (e.g., compaction of repetitive behavior in execution traces and interactive exploration). To address the scalability issues, decreasing the horizontal size of recovered diagrams is likewise essential. Nonetheless, few studies have addressed this point. Thus, further development and improvement of reduction techniques that focus on the horizontal direction are needed.

In this dissertation, we propose a fully automated technique for recovering a summarized sequence diagram of a reasonable size, with a central focus on reducing the horizontal size of recovered diagrams. Our technique consists of three parts: (1) modeling an execution trace with our behavior model; (2) identifying core objects that play important roles in an execution scenario; (3) constructing object groups corresponding to concepts for recovering a summarized sequence diagram.

(1) Modeling an execution trace with our behavior model

Sequence diagram recovery techniques take execution traces as their input. Most existing studies on sequence diagram recovery develop their own tracers, and the formats of traces vary across those tracers.

We present a behavior model (B-model) that models the behavior of an object-oriented system for behavioral design recovery. With the B-model, we can formalize an execution trace in a tracer-independent format; this enables us to develop several trace analysis techniques independent of tracers. Besides,

we define mappings from B-model elements to sequence diagram elements; a B-model event sequence can be easily converted into a sequence diagram, which fills the gap between the representation of an execution trace and that of a sequence diagram.

To demonstrate the feasibility and sufficiency of the B-model, we show an application, which calculates a slice of a recovered sequence diagram, built on the B-model. Through examples of slice calculation, we show that our slicing application successfully recovers sliced sequence diagrams of a reasonable size; it confirms that the B-model holds finer-grained information required for traditional program analyses, and can be a useful tool for behavioral design recovery.

(2) Identifying core objects

The most important concepts of a system are implemented by very few key classes. Comprehending the behavior of objects of such key classes is highly important in an early stage of program comprehension.

We present a technique for identifying core objects of such key classes, which plays important roles in an execution scenario, from a behavioral point of view. Our core identification technique consists of two steps. First, we identify and eliminate temporary objects, which are trivial to comprehend a behavioral overview of a system, by analyzing dynamic scopes of objects based on reference relations and lifetimes thereof. Then, we estimate the importance of non-temporary objects based on access frequencies, and thereby identify core objects.

The results of our experiment using traces generated from various types of open source software (OSS) show that our technique outperforms the state-of-the-art technique. In the experiment, our technique identified a total of 5–11 core objects within top 7–222 (108 on average) important objects, whereas that of the state-of-the-art technique ranged 148–4,378 (998 on average).

(3) Constructing object groups for recovering a summarized sequence diagram

To improve maintainability, in object-oriented programming, a concept is often divided into several classes by using design patterns. From the perspective of program comprehension, it increases the number of design elements to comprehend, and enlarges the amount of effort required for program comprehension.

We present techniques for constructing object groups and recovering a summarized sequence diagram. Our technique constructs object groups at a concept-level based on Pree's meta patterns that are the most primitive design patterns. We detect all the occurrences of Pree's meta patterns in program code, and

then groups template and hook objects involved in the same meta pattern; this reunifies divided concepts as object groups.

Given the results of core object identification, we identify important object groups. By visualizing intergroup interactions only among important groups, we obtain a summarized sequence diagram of a reasonable size that depicts object interactions at a concept-level.

In an early stage of program comprehension where developers do not attain a detailed knowledge of a subject system, they prefer highly abstracted views to comprehend behavioral overviews rather than finer-grained views depicting detailed interactions. Because our summarized sequence diagram provides a behavioral overview (an abstracted view) of important object groups (concepts), our solution can be a valuable tool in an early stage of program comprehension.

The results of our experiments using various types of OSS show that our technique outperforms the state-of-the-art trace summarization technique in terms of reducing the horizontal size of a reverse-engineered sequence diagram. Regarding the quality of object grouping, under the condition of #lifelines < 30 (which is acceptable for manual investigation), our technique achieved an *F*-score (resp. a *Recall*) of 0.670 (resp. 0.793); this is 0.249 (resp. 0.123) higher than that of the state-of-the-art technique.

The runtime overhead incurred by our technique was 129.2 % on average. This overhead is relatively small compared with recent scalable dynamic analysis techniques; thus, it is acceptable in a development phase.

Overall, our technique can recover a summarized sequence diagram with a small runtime overhead. The recovered diagram depicts a behavioral overview of important concepts in a subject system, which is expected to be a valuable tool in an early stage of program comprehension.

Acknowledgments

My utmost gratitude goes to Prof. Takashi Kobayashi, my supervisor in the doctoral course at Tokyo Institute of Technology. His enthusiasm, extensive knowledge, generosity, and kind support enabled me to write this dissertation and enjoy my doctoral journey.

Also, I would like to acknowledge Prof. Kiyoshi Agusa and Prof. Shoji Yuen who are my former supervisors in the master's and bachelor's courses, respectively. I learned a lot of fundamentals of doing research and writing academic papers on software engineering and science from them, which was my invaluable experience.

I am grateful to the co-authors of my publications, Prof. Shinichiro Yamamoto, Prof. Noritoshi Atsumi, Prof. Motoshi Saeki, Prof. Shinpei Hayashi, Tatsuya Toda, Kaixie Lyu, Maaki Nakano, Yamato Haraguchi, Go Takagi, Keita Hiranouchi, Yoshiya Ishida, and Yuu Arimatsu, for the stimulating discussions and their great contributions to my research.

My appreciation also goes to Prof. Takashi Ishio and Kaixie Lyu, the developers of the splendid tools (SELogger and SDExplorer) that greatly facilitated my development of the research tools. They also gave me lots of valuable feedback and suggestions that advanced my research.

Regarding my school life, I thank all the colleagues at Kobayashi laboratory and Agusa/Yuen laboratory for lots of enjoyable moments. I have enjoyed chatting, traveling, drinking, and writing code together with you over the years, which has played a major role as the source of my vitality.

I must also acknowledge the financial support from Japan Student Services Organization (JASSO), Japan Society for the Promotion of Science (JSPS), and Tokyo Institute of Technology. Without the support, I could not have completed this research.

Last, but certainly not least, I would like to express my sincere gratitude to my family. You have always given me a lot of warm support. Thank you.

Kunihiro NODA
February 2019

1

Introduction

Program comprehension is a quite important activity in software maintenance. For aiding comprehension of behavioral aspects of a system, recovering behavioral diagrams from execution traces is a promising approach; however, owing to the massiveness of execution traces, recovered diagrams are often afflicted by scalability issues.

The objective of this research is to develop a technique that recovers a summarized sequence diagram of a reasonable size from an execution trace, with the goal of helping developers comprehend a behavioral overview of a system.

This chapter describes the motivation and outline of this research. We explain the importance of program comprehension and behavioral visualization. Then, we provide a brief description of the techniques for recovering and summarizing sequence diagrams. After the problem description of existing trace summarization techniques, we present challenges to address in this research. Finally, we show the dissertation outline and our contributions.

1.1 Program Comprehension

Software evolves through numerous maintenance activities, such as feature implementation, bug fixes, and refactoring, according to continually changing user/business requirements. For the success of those maintenance activities,

developers need to fully comprehend the structure and behavior of a software system, which is termed as "Program Comprehension" [1].

Program comprehension is a quite important activity in software maintenance. Owing to the complexity of (legacy) software systems, structural and behavioral comprehension of the existing systems (where developers study the existing code, documents, and related artifacts) are time-consuming and error-prone. A recent study conducted a large-scale field study, and reported that the time spent in program comprehension activities accounted for up to 58 % of developers' time [2]. Thus, aiding program comprehension is crucial for the success of software maintenance activities.

Approaches to program comprehension are roughly categorized into three types: top-down, bottom-up, and a combination of those [1, 3, 4]. The top-down approach is taken by developers that are familiar with application domains and program code. Starting from the construction of hypotheses or expectations on a target program, developers try to map those to program code [5]. On the other hand, if developers are unfamiliar with application domains, solution space, and program code, the bottom-up approach is preferred. After comprehension of each code statement, developers mentally chunk or group their understandings into higher-level abstractions, which proceeds iteratively (recursively) until sufficient knowledge about their maintenance tasks is obtained [5].

In program comprehension tasks, developers need to create mental mappings between program code and software behavior; however, because a large amount of source code imposes developers large cognitive overloads, it is difficult and challenging [6]. This emphasizes the need for specialized tools or techniques to facilitate the cognitive process [7]. A great deal of existing work has proposed many types of tools and techniques for aiding program comprehension, such as design or architecture recovery, auto/re-documentation, visualization [8, 9].

1.2 Behavioral Visualization through Program Analysis

Software documentation is a vital source of information for program comprehension; however, in many cases, those documents tend to be outdated or non-existent [10–12]. Especially, behavioral documents/diagrams, such as UML (Unified Modeling Language [13]) sequence diagrams and collaboration diagrams, are created much less often than structural documents/diagrams such as UML class diagrams and component diagrams. Moreover, comprehending the behavior of an object-oriented system solely from source code is cumbersome and

error-prone due to its dynamism (e.g., dynamic binding and reflection). Therefore, in program comprehension activities, developers tend to spend much of their time on behavioral comprehension.

To facilitate behavioral comprehension, many preceding studies have proposed numerous techniques for visualizing the behavior of a system from source code or execution traces [8]. For object-oriented systems, visualizing its behavior as UML sequence diagrams, which are commonly used notations and easy to be accepted in developer communities, is a promising approach [7, 14]. The recovered sequence diagrams (commonly termed as "reverse-engineered sequence diagrams") reflect system's actual behavior; thus, they do not confuse developers by incorrect information stemming from outdated documents, and provide valuable cognitive support for behavioral comprehension.

Techniques for recovering sequence diagrams are categorized into three types from the perspective of analysis types: static, dynamic, and hybrid (a combination of static and dynamic) techniques.

Static approaches mainly take source code as their input, and recover sequence diagrams without any executions. This type of approaches can recover sequence diagrams that depict an overall picture of a subject system (i.e., high coverage). While the analysis cost of static techniques is, in general, relatively lower than that of dynamic ones, static approaches cannot well handle the dynamism mentioned above, and have a risk of recovering infeasible paths.

Dynamic approaches take execution traces as their input for generating reverse-engineered sequence diagrams. With instrumentation of logging code, those approaches record runtime events such as method-entries/exits and object creations, and thereby they recover sequence diagrams that depict actual behavior of specific execution scenarios (Figure 1.1). Because dynamic techniques are based on execution logs, they can well resolve the dynamism, and no infeasible paths are contained in recovered diagrams. There are some drawbacks of dynamic approaches: they impose a certain level of runtime overhead; the analysis costs tend to be expensive; recovered diagrams reflects only single (or a limited number of) execution scenario(s) (i.e., low coverage). Hybrid approaches exploit static analyses to alleviate the disadvantages of dynamic approaches.

A recent survey reported that the majority of state-of-the-art techniques for recovering sequence diagrams relied on dynamic analysis; of a total of 37 existing studies investigated, 8 approaches were static ones, 18 approaches were dynamic ones, and 11 approaches were hybrid ones [14]. As mentioned above, static analysis cannot well resolve dynamism. In addition, most of the information obtained by static analysis can also be captured by dynamic analysis. Thus, dynamic and hybrid approaches are preferred for recovering sequence diagrams.

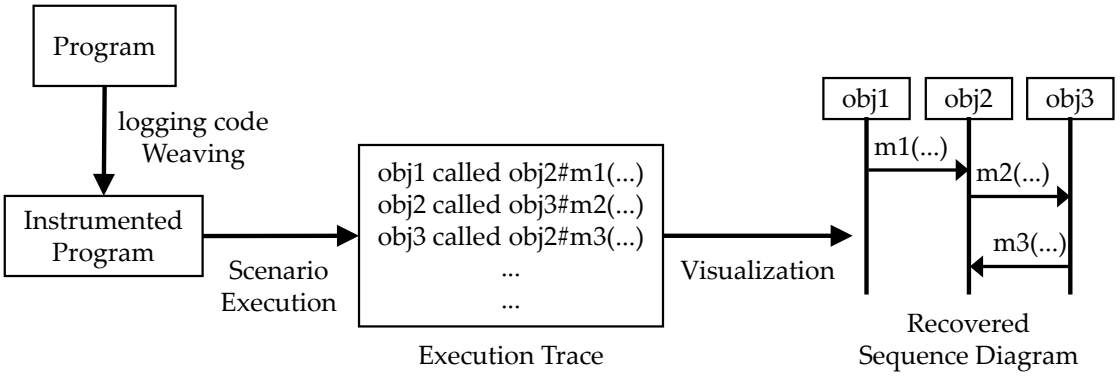

Figure 1.1: Sequence diagram recovery through dynamic analysis.

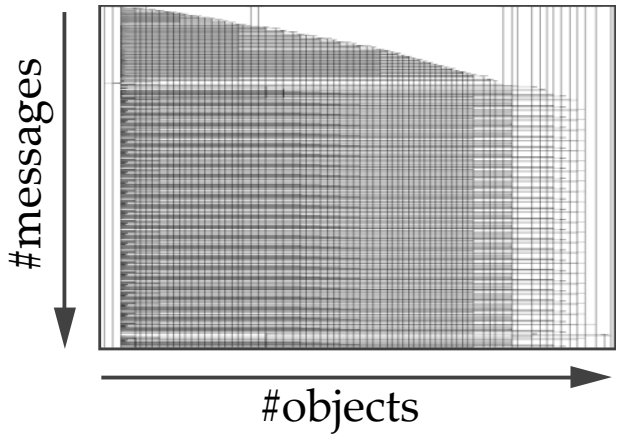

Figure 1.2: Massive-scale reverse-engineered sequence diagram.

1.3 Problem Description and Challenges

Execution traces, which become the input of dynamic (or hybrid) techniques for recovering sequence diagrams, contain an enormous amount of information in general. Even an execution traces generated from a simple and short execution scenario contains tens of thousand events. Executing a subject system for a few minutes produces a large execution trace that contains hundreds of million events [15, 16].

Such a great deal of information causes techniques for recovering sequence diagrams to suffer from scalability issues. If we visualize all the information contained in an execution trace as a reverse-engineered sequence diagram, an unreadable diagram that contains tens of million messages will be produced (Figure 1.2).

Those scalability issues put an emphasis on the importance of developing techniques for summarizing or effectively exploring large-scale reverse-

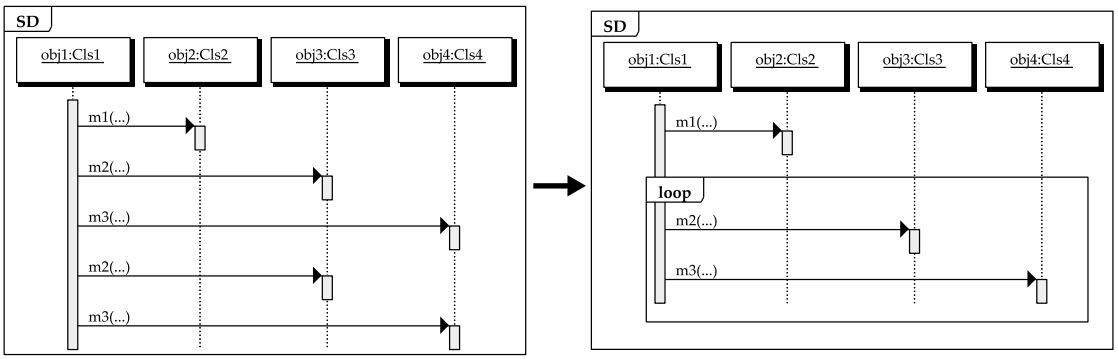

Figure 1.3: Compaction of repetitive behavior (i.e., vertical summarization).

engineered sequence diagrams. Of the existing studies on reverse-engineered sequence diagrams investigated in a recent survey [14], more than half of those have proposed compaction/summarization techniques to address the scalability issues.

The vertical size of a reverse-engineered sequence diagram grows in proportion to execution time, whereas the horizontal size of the diagram grows in proportion to the number of generated objects (Figure 1.2). Most existing techniques focus on vertical summarization of reverse-engineered diagrams [17–29] or effective exploration of massive-scale diagrams [7,30–35]. For example, some work compresses the vertical size of the diagram by detecting and compacting repetitive behavior stemming from iterative statements or recursive method calls [17,18,21,22,24] (Figure 1.3). Some other work provides functionalities of filtering, searching, or interactive exploration for helping developers efficiently examine large-scale diagrams [7,30,35].

To cope with the scalability issues, decreasing the horizontal size of the diagram is likewise quite important. Nonetheless, few studies have addressed this point [24,36,37]; thus, further development and improvement of reduction techniques that focus on the horizontal direction are needed. This motivates us to develop an effective technique for horizontally summarizing reverse-engineered sequence diagrams.

The primary objective and challenge of this research are to address the scalability issues of reverse-engineered sequence diagrams. Especially, we try to develop a fully automated effective technique for summarizing recovered sequence diagrams with a central focus on horizontal reduction, which few existing studies have addressed. We expect that our resulting summarized sequence diagram can be a valuable tool for developers to comprehend a behavioral overview of a system in an early stage of program comprehension.

In an early stage of program comprehension, developers first identify and

comprehend classes and modules that play important roles in a system; from thence, they expand their comprehension into peripheral classes and modules. Previous work states that *usually, the most important concepts of a system are implemented by very few key classes* [38, 39]. Such key classes are crucial for the early stage of program comprehension. Some work tried to automatically identify those key classes for aiding program comprehension [40–42].

We identify core (important) objects of such key classes that play important roles in an execution scenario via program analysis from a behavioral point of view. By building high-level (concept-level) abstractions of identified core and related objects, we significantly reduce the horizontal size of a reverse-engineered diagram; thereby, we produce a summarized sequence diagram of a reasonable size. The resulting diagram depicts a behavioral overview of important concepts in a system. Thus, our solution is expected to be helpful for obtaining a high-level overview in an early stage of program comprehension.

Scalability issues of dynamic (execution trace) analyses have three major components: computational, visual, and cognitive components [43, 44]. The computational component indicates the importance that the results of dynamic analyses can be computed in a reasonable amount of time. The visual component requires that the representation of the results is easy to be interpreted by users. The cognitive component means that the size of the results should be an acceptable one for uses in order to prevent large cognitive overloads.

Our solution, a technique for recovering a summarized sequence diagram, represents its result as a UML sequence diagram that is a commonly used notation; this addresses the visual aspects of the scalability issues. We generate a summarized sequence diagram of a reasonable size via the core object identification and concept-level abstraction mentioned above, which addresses the cognitive aspect. In addition, we try to ensure that the runtime overhead imposed by our technique would be an acceptable one by proper weaving extent settings; this addresses the computational aspect.

1.4 Contributions

The overall contribution of this research is to propose a fully automated technique for recovering a summarized sequence diagram with a central focus of horizontal reduction, which few existing studies have addressed. More specifically, the major contributions of this research are summarized as follows.

- A behavior model for program analysis and behavioral design recovery is presented.

  The formats of execution traces vary across tracers (tools for recording

execution traces). There exists a gap between the representation of an execution trace and that of a recovered sequence diagram.

The behavior model (B-model) that we propose in this dissertation models the behavior of an object-oriented system. By using our B-model, an execution trace (runtime information) is formalized in a tracer-independent form; this enables us to develop several dynamic analysis algorithms independent of any tracers. Moreover, elements in the B-model can be easily converted into sequence diagram elements; this addresses the gap between the representation of an execution trace and that of a recovered sequence diagram. The details of the B-model are described in Chapter 4.

- A technique for identifying core objects that play important roles in an execution scenario is proposed.

  As mentioned above, the most important concepts of a system are implemented by very few key classes. Our technique identifies core objects of those key classes based on dynamic properties (such as reference relations and access frequencies) obtained from B-model data. The identified core objects are valuable for developers, who are not familiar with a subject system, to comprehend a high-level behavioral overview of the system. The details of the core identification technique are described in Chapter 5.

- Techniques for constructing object groups at a concept-level and generating a summarized sequence diagram are developed.

  In an early stage of program comprehension, developers prefer highly abstracted views to comprehend behavioral overviews rather than finer-grained views that depict numerous design elements (such as objects and variables) and detailed interactions. We construct object groups based on design patterns to make a concept-level abstraction of objects. In conjunction with the core identification technique, we identify important object groups; then, we generate a summarized sequence diagram of a reasonable size by visualizing intergroup interactions only among important object groups, which provides a behavioral overview (an abstracted view) of important concepts in a system. The resulting summarized diagram is expected to be a valuable tool in an early stage of program comprehension. The details of the object grouping and visualization techniques are described in Chapter 6.

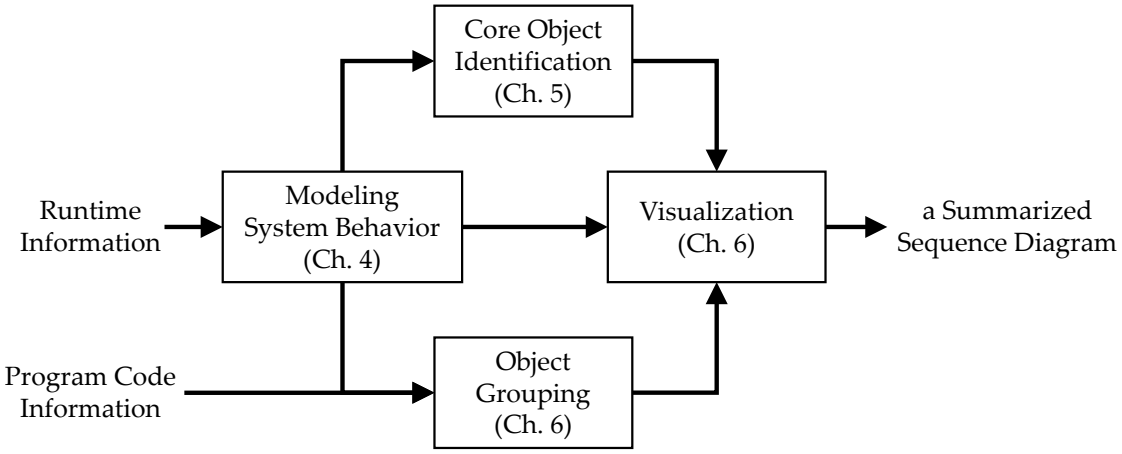

Figure 1.4: Overview of summarized sequence diagram recovery.

1.5 Dissertation Overview

The remainder of this dissertation is organized as follows.

Chapter 2, "Background," briefly describes techniques underlying our approach to recovering a summarized sequence diagram, such as program analysis and design recovery. Besides, we describe key related tools and techniques: summarization of recovered behavioral diagrams, effective exploration of massive-scale sequence diagrams, key class identification, etc.

Chapter 3, "Summarized Sequence Diagram Recovery," describes the overview of our technique for recovering a summarized sequence diagram. The overall process flow of our recovery technique is shown in Figure 1.4. Our technique, first, models an execution trace with the B-model proposed in Chapter 4. Then, taking a B-model event sequence as an input, we apply our core identification technique (Chapter 5) and object grouping technique (Chapter 6) to visualize a summarized sequence diagram. In Chapter 3, we explain details of the overall process of our technique and the relationship among B-model data and the results of core-identification/object-grouping.

Chapter 4, "Modeling the Behavior of an Object-Oriented System for Behavioral Design Recovery," proposes a behavior model (B-model) that models the behavior of an object-oriented system. An execution trace (runtime information) can be formalized in a tracer-independent form via the B-model. B-model data hold fine-grained information necessary for traditional program analyses such as data and control flow analyses. Besides, each element in the B-model can be easily converted into an element of a sequence diagram, which facilitates the development of behavioral design recovery techniques. In Chapter 4, we describe what elements/attributes exist in the B-model and how to convert B-model data

into a sequence diagram. To demonstrate the feasibility and sufficiency of the B-model, we show a sequence diagram slicing application built on the B-model.

Chapter 5, "Identifying Core Objects through Dynamic Analysis," elaborates our technique for identifying core objects that play important roles in a system. The core identification technique consists of 2 steps: pruning temporary objects and estimating object importance. First, we describe how temporary objects can be identified by analyzing the reference relations and lifetimes of them. Then, we explain how to estimate the importance of each object based on access frequencies. We evaluate our technique by using traces generated from various types of open source software. By comparing our technique with a state-of-the-art trace summarization technique and measuring runtime overhead, we show the feasibility and effectiveness of our technique.

Chapter 6, "Abstracting Object Interactions at a Concept Level for Recovery of a Summarized Sequence Diagram," elaborates techniques for constructing object groups based on design patterns and generating a summarized sequence diagram. We describe our algorithm for detecting design patterns in program code, and explain how objects are grouped at a concept-level by using detected patterns. Besides, we show a visualization algorithm for generating a summarized sequence diagram by utilizing the results of core identification and object grouping. We conduct an experiment using various types of open source software, for an overall evaluation of our technique. We show the feasibility and effectiveness of our technique through comparison with a state-of-the-art trace summarization technique.

Chapter 7, "Conclusion," concludes this research. We briefly revisit/review our techniques proposed in Chapters 3 to 6, and describe the summaries of our contributions. Finally, we show some opportunities for future research.

1.6 Origin of Chapters

The core parts of this dissertation are based on peer-reviewed publications. The following list provides the relationship between each chapter and our publications.

Chapter 4

This chapter is based on our journal paper "Reticella: An execution trace slicing and visualization tool based on a behavior model" [45] published in *IEICE Transactions on Information and Systems*, Copyright © 2012 IEICE. The journal paper is an extended version of our earlier work "Sequence diagram slicing" [46] appearing in the *Asia-Pacific Software Engineering Conference (APSEC)*, Copyright © 2009 IEEE.

Chapter 5

This chapter is based on our journal paper "Identifying core objects for trace summarization by analyzing reference relations and dynamic properties" [47] published in *IEICE Transactions on Information and Systems*, An earlier version of the journal paper, entitled "Identifying core objects for trace summarization using reference relations and access analysis," [48] has been published at the *Annual Computer Software and Applications Conference (COMPSAC)*,

Chapter 6

This chapter contains our work "Execution trace abstraction based on meta patterns usage" [49] appearing in the *Working Conference on Reverse Engineering (WCRE)*,

2

Background

This chapter briefly describes techniques underlying our approach to recovering a summarized sequence diagram, such as program analysis and design recovery. In addition, we describe key related tools and techniques: structural and behavioral design recoveries, effective exploration of massive-scale recovered diagrams, key class identification, etc.

2.1 Program Analysis

There are numerous studies utilizing program analysis techniques for various kinds of software development support: aiding program comprehension [8,50], uncovering faulty behavior [51], debugging support [52], automatic documentation [53], etc. Program analysis techniques extract structural and behavioral properties (e.g., software metrics, data/control flows, etc.) of a subject program, and sometimes construct abstracted models (higher-level abstractions) for further analyses and investigations.

2.1.1 Analysis Types

Program analyses can be categorized into three types: static, dynamic, and a combination of those (hybrid).

Static techniques analyze program code without any program executions. The primary source of information of static program analyses is program code (occasionally object code) of a subject system. Static analysis techniques first parse program code, and (optionally) construct intermediate representations (e.g., abstract syntax trees (ASTs)). Then, they obtain or infer structural and behavioral properties/models of a subject system.

While static techniques incur lower analysis cost compared with dynamic ones, the correctness of static analysis results are often impaired by the dynamism of a subject program (e.g., dynamic bindings and reflections). For example, constructing a call graph of an object-oriented system by static analyses needs some approximations for resolving the class hierarchy [54] (e.g., class hierarchy analysis (CHA [55]) and variable-type analysis (VTA [56])); this produces a certain number of incorrect results (false positives).

Dynamic techniques analyze runtime structures and behavior of a subject system with program executions. Many dynamic techniques work on execution traces (execution logs where runtime information are stored). In contrast to static approaches, because dynamic analyses are based on runtime information where dynamism is resolved, dynamic approaches produce no (or fewer) false positives.

While dynamic analyses can grasp actual states of a system, the results of them are valid only for specific execution scenarios exercised. In addition, dynamic analyses are often afflicted by scalability issues because their input, execution traces, are quite large in general [8]. Note that the incompleteness (low coverage) of dynamic analyses is not necessarily a drawback in the context of facilitating program comprehension because developers are not always interested in entire structures and behavior of a subject system, and such a goal-oriented strategy well suits developers' needs [6, 57].

Hybrid techniques alleviate the drawbacks of dynamic approaches by utilizing static analyses [58]; e.g., restricting the extent of dynamic analyses or complementing information dynamic analyses cannot discover.

2.1.2 Collecting Runtime Information for Dynamic Analysis

To perform dynamic analyses, special efforts are required for obtaining runtime information. Major approaches to collecting execution traces are using debugger architectures, instrumentation, and code-rewriting.

Some runtime environments provide debugger architectures; for example, JVM (Java virtual machine) offers JPDA [59] (Java platform debugger architecture) that allows developers to create debugger applications. By using JPDA, developers can create debugger applications that communicate the JVM executing a subject program examined, and can receive runtime events from the JVM. The major advantage of using debugger architectures is to obtain fine-grained controls of the target program being executed. In general, tools built on debugger architectures can stop/resume target program, inspect values of variables, examining internal data of a runtime environment, etc. On the other hand, recording runtime information via debugger architectures incurs a large amount of runtime overhead, which considerably slows down analysis programs and might affect (change) behavior of a subject program.

Another approach is instrumentation, where special logging codes are woven (inserted) into binary files of object code. There are several libraries for manipulating object code; e.g., ASM [60] and Javasisst [61]. The runtime overhead imposed by instrumentation is much lower compared with using debugger architectures. Because reducing the runtime overhead of dynamic analyses is quite important [62], the instrumentation approach (if possible) is preferred than utilizing debugger architectures.

Code-rewriting is a straightforward way to record which parts of a subject program are actually executed. As is the case with instrumentation, this approach embeds special logging code into program source code. Developing tools for rewriting source code tends to be laborious and error-prone; thus, to the best of our knowledge, the code-rewriting approach is taken less often than the instrumentation approach.

2.1.3 Limitations

Because the input of program analysis techniques is source code and execution traces of a subject system, there are some limitations; they cannot capture information stemming from (related to) artifacts except for program code and traces. For example, a dependency between a module and its documents (written in a natural language) cannot be discovered by program analyses. Moreover, even dependencies between code fragments are sometimes difficult to be captured by traditional program analyses because, for example, libraries or frameworks, which are not analyzed, could hide those dependencies [63].

To overcome these limitations, various types of techniques have been proposed in the literature. For example, some studies mined historical data to obtain possible relationships (dependencies) among software artifacts [64–66], which enables us to capture information that cannot be obtained by program

analyses. Utilizing statistical/machine-learning approaches [67, 68] or semantic information (embedded in comments, documents, and identifiers) [69, 70] is likewise a possible solution to capture hidden dependencies.

2.2 Design Recovery

Software documents tend to be outdated or non-existent due to numerous iterative modifications since the first version has shipped [10–12]. Maintaining (legacy) software systems that have such poor documents requires a large amount of effort of developers.

To alleviate the issue, utilizing design recovery techniques is a promising approach. Design recovery is an (automated) approach for recreating design abstractions/models from source code, existing documents (if available), other artifacts, experiences, general knowledge, etc. [71, 72]. Recovered views and models reflect the actual states of a subject system, which plays crucial roles in facilitating program comprehension.

In this section, we describe key related tools and techniques for recovering design models.

2.2.1 Structural Design Recovery

UML class diagrams are widely used notations for describing structural aspects of systems. Recovering class diagrams from source code is straightforward. Some (non-)commercial tools support class diagram recovery from a set of existing source code [73, 74].

Automatically recovered class diagrams often become very large because source code contains numerous implementation details (trivial classes); reverse-engineered diagrams tend to be much larger than forward-designed ones. Developers prefer forward-designed diagrams because the size of reverse-engineered diagrams are overwhelming and identifying important (relevant) elements in such large diagrams is quite laborious [75].

To address the scalability issue, some studies have proposed automated techniques for identifying important classes and generating condensed reverse-engineered class diagrams.

Osman et al. presented a machine-learning-based technique for condensing reverse-engineered class diagrams [76]. Their technique utilizes some design metrics as features to create (train) a model predicting whether each recovered class is important. By using the model, it generates condensed versions of reverse-engineered class diagrams.

Thung et al. extended the technique by Osman et al. by adding network

metrics among classes (e.g., closeness centrality, page rank, etc.) to train a model [77]; that is, their technique considers relationships among classes that include aggregation, generalization, etc. They found that network metrics (features) could discriminate important classes better than design metrics (features), and improved the quality of the classification results.

In such supervised machine learning approaches, it is required that a certain portion of data (classes) must be manually labeled as important or not. The labeling cost is not negligible because reverse-engineered class diagrams contain numerous design elements. To alleviate the manual labeling cost, Yang et al. presented a technique that combined unsupervised learning, which selected the most representative samples as training data, and ensemble learning, which calculated the final results [78].

2.2.2 Behavioral Design Recovery

Behavioral aspects of a system tend to be less often documented than structural aspects; however, fully understanding the behavior of a system is an essential prerequisite for the success of software maintenance tasks. This puts an emphasis on the need for effective behavioral design recovery techniques.

For object-oriented systems, UML sequence diagrams are commonly used for describing system behavior; many existing studies have presented techniques for recovering system behavior as sequence diagrams [7, 14]. As described in Section 2.1.1, static analyses cannot resolve the dynamism of object-oriented systems; thus, dynamic approaches are preferred for behavioral design recovery. A recent survey reported that the majority of state-of-the-art techniques for recovering sequence diagrams relied on dynamic analyses [14].

Owing to the massiveness of execution traces, reverse-engineered sequence diagrams are often afflicted by scalability issues. Thus, most existing behavioral recovery techniques have some functionalities for reducing the total amount of runtime information to visualize or effectively exploring/visualizing a large amount of information.

The means of coping with the massiveness of execution traces are roughly categorized into three types: (1) vertical or (2) horizontal summarization of reverse-engineered sequence diagrams; (3) effective visualization or exploration of a large amount of information. Most existing techniques focus on the (1) or (3) approach. In the following sections, we describe key related work for each category.

2.2.2.1 Vertical Summarization

There are a number of techniques for vertically summarizing recovered sequence diagrams. Major vertical summarization approaches are repetitive behavior compaction and phase detection.

Repetitive behavior compaction approaches.

A great number of similar (repetitive) behavioral patterns appear in an execution trace, owing to iterative statements or recursive method calls in program code. Thus, identifying and compacting such repetitive behavior results in a significant reduction of the vertical size of reverse-engineered sequence diagrams.

Taniguchi et al. presented four compaction rules for identifying similar (not exactly the same) structures in an execution trace [17]. Their technique visualizes identified similar structures with compact notations (their original annotations); thereby, the vertical size of recovered diagrams are greatly reduced.

Jayaraman et al. proposed a regular-expression-based vertical compaction algorithm [24]. By computing the minimal regular expression where a character corresponds to a method call, their technique identifies repetitive method calls. Then, the identified calls are visualized with UML combined fragments in order to obtain a compact visualization.

Identifying similar structures solely from an execution trace is not straightforward and thus produces a certain number of false positives. Myers et al. utilized debug information and program code information to identify behavior arising from iterative statements in an execution trace [18]. Srinivasan et al. likewise used source code information to compact repetitive behavior in their optimized sequence diagrams [22, 22]. By combining static information, their techniques have a high accuracy of loop identification.

Phase detection approaches.

If multiple features are exercised for generating an execution trace, the resulting trace contains runtime information on those multiple features. Phase detection approaches try to divide an entire trace into several segments, called phases, in which the starting point of each phase corresponds to that of each feature in an execution scenario.

Watanabe et al. presented an automated technique for identifying phases in an execution trace [19]. Their technique examines the creation timings of objects, and calculated a working set of active objects. If the contents of the working set are significantly changed at a certain execution point, their technique identifies the point as the beginning of a phase. AMIDA [23] is a sequence diagram recovery toolkit that implements the phase detection algorithms by Watanabe et

al. and a loop compaction algorithm. In AMIDA, each of the identified phases is visualized as a separate sequence diagram.

Pirzadeh et al. presented a phase detection technique and an event sampling technique [25, 27, 28]. Their technique identifies phases in an execution trace based on gestalt laws of perception [79]. It creates dense groups of trace events based on the similarity and continuation scheme: two method call events are made closer if they call the same methods (the similarity scheme); consecutive events are made closer (the continuation scheme). By clustering those dense groups, it identifies phases in an execution trace. After phase detection, it treats identified phases as strata, and samples events from each stratum (i.e., stratified sampling) to generate a compact trace that is representative of the original trace.

Similar phases could appear several times in an execution trace because some computations might happen multiple times in an execution scenario. Pirzadeh et al. presented a technique for identifying those similar phases by exploiting text mining techniques (TF-IDF and cosine similarity) in order to achieve a better (non-redundant) representation [29].

Other approaches.

Hamou-Lhadj and Lethbridge presented a technique for identifying unimportant methods to obtain concise sequence diagrams [20]. Because there are a large number of implementation details in program code, recovered sequence diagram contains numerous messages that are trivial for program comprehension. Their technique calculates the utilityhood score of each method on the basis of its fan-in and fan-out; thereby, it identifies unimportant methods (implementation details). By removing identified trivial method calls, it obtains a vertically (and horizontally) summarized sequence diagram that depicts only important behavior.

2.2.2.2 Horizontal Summarization

Along with the vertical summarization of reverse-engineered sequence diagrams, the horizontal reduction of the diagrams is quite important in order to make the diagrams more practical. Nonetheless, few studies have presented techniques for vertically reducing the size of reverse-engineered sequence diagrams.

A straightforward way to horizontally summarize a reverse-engineered sequence diagram is class/package-name-based grouping or filtering: grouping objects having the same class or package names; filtering out objects of specific classes or packages. JIVE, an interactive visualization environment, provides grouping and filtering functionalities for horizontal summarization [24]:

grouping objects having the same class names; filtering out objects based on type/method names and visibility. Jinsight, which visualizes the runtime behavior of a system, likewise provides functionalities for filtering out information of no interest; e.g., hiding specified classes or method invocations except for the call history leading to a specified method invocation [80].

OGAN is a tool for visualizing representative interaction scenarios related the two classes specified by a user [81]. OGAN computes, for each object, a set of classes using/used-by the object, named a dynamic interaction context. Then, OGAN identifies representative objects based on an equivalence relation regarding dynamic interaction contexts. OGAN would be a valuable tool for program comprehension when developers know the names of classes of interest and would like to investigate behavior thereof (for example in a debugging phase).

Dugerdil and Repond presented a class clustering technique in which each cluster corresponds to a functional entity whose behavior could be understood independent of other entities [37, 82]. Their technique segments an execution trace, and computes binary occurrence vectors that represent whether each class is used in each segment. Then, it clusters classes based on the occurrence vectors. Each cluster is depicted as a single lifeline in a reverse-engineered sequence diagram; thereby, their technique obtains a horizontally summarized sequence diagram. The technique by Dugerdil and Repond is suitable for comprehending the behavior of a complex execution scenario exercising multiple features because it identifies functional components (clusters of objects), which could be understood independent of others, and visualizes relationships among those components.

Toda et al. proposed a technique for constructing object groups based on GoF (Gang of Four) design patterns used in source code [36]. Objects involved in the same design pattern are expected to be strongly related to one another. They defined three grouping rules from the viewpoint of pattern types (i.e., creational, behavioral, and structural patterns). By visualizing interactions only among object groups, their technique achieves a horizontal summarization of a recovered sequence diagram.

Our technique for recovering a summarized sequence diagram is likewise categorized into this type (horizontal summarization approaches). We will compare our technique with other horizontal summarization techniques in Section 6.6.1.2.

2.2.2.3 Effective Visualization and Exploration

Apart from summarization, effective visualization (with specialized views) and exploration are possible approaches to addressing the scalability issues of execution traces.

Bennett et al. discussed the visual limitations of (reverse-engineered) sequence diagrams and stated a set of interaction-features that diagram viewers should provide [7]: selection, navigation, focusing, zooming, searching, annotating, etc. Providing those interaction-features for interactive exploration (together with summarization functionalities) helps developers identify/examine specific behavior of interest in a massive-scale sequence diagram. Many existing visualization tools support (a part of) those interaction-features [7,30,83,84].

Rendering performance is likewise an important factor because visualizing millions of runtime events easily crashes or freezes visualization tools. Several frameworks and tools support UI and data virtualizations (i.e., rendering and retrieving data on demand) to smoothly render massive-scale information [35].

Extravis is a visualization tool that has two mutually-linked specialized views (circular bundle and massive sequence views) employing several visual attributes [31,32]. The circular bundle view depicts structural entities and relationships thereof in a circle form, which can be displayed within one screen. The massive sequence view shows interactions among system entities in chronological order, which is similar to a sequence diagram. To visualize a large-scale interaction history within one screen, they utilized the information mural approach [85]. In the massive sequence view, the entire (nested) structure of a system is shown on the horizontal axis, while an interaction history is plotted on the vertical axis, where interactions are colored and anti-aliased based on the call directions and their importance (the importance is calculated based on the frequency of each interaction). These features help developers grasp a big picture of a system and locate features of interest.

2.3 Other Related Topics

2.3.1 Key Class Identification

The most important concepts of a system are implemented by very few key classes [38,39]. Such key (important) classes are crucial in an early stage of program comprehension. Developers, who are not familiar with a subject system, first identify and comprehend those key classes; from thence, they proceed to investigate peripheral classes.

Several studies have presented automated techniques for identifying key

classes. Most of the techniques exploit network analysis and machine learning [41].

Zaidman and Demeyer presented a technique for identifying key classes by utilizing a link analysis algorithm HITS (hyperlink-induced topic search) and coupling metrics [40, 86]. They consider that key classes are characterized by being tightly coupled. Their technique calculates direct and indirect coupling metrics by using the HITS algorithm, and thereby identifies important classes.

Perin et al. presented a technique for ranking software artifacts by using a network analysis algorithm PageRank [87]. Their technique decides the importance of each class by using the PageRank algorithm on a graph, where each node corresponds to a class and each edge is derived from inheritance and reference dependencies among classes.

Şora likewise proposed a key class identification technique that utilizes the PageRank algorithm [41, 88]. Her technique models a subject system as a directed graph, where each node represents a class and each edge is derived from a number of program dependencies (inheritance, member access, method call, instantiation, etc.). It calculates an important class ranking by using the PageRank algorithm on the graph, and identifies key classes. Şora extended her work by using fuzzy rules regarding class attributes (methods, dependencies, and PageRank values), and thereby improved the quality of key class identification [42]. In addition, she further examined the effect/contribution of various types of attributes in key class identification [89].

There are still several other approaches that utilize network analysis [90–92]. For example, Steidl et al. estimated the importance of each class based on centrality indices (e.g., betweenness, PageRank, Markov, etc.) on (un)directed graphs [90]. Meyer et al. identified important classes by applying K-core decomposition to a static dependency graph [91].

Hammad et al. presented a technique for estimating the importance of each class by examining the change history of a system [93]. They consider that important classes are frequently affected by design changes. Their technique counts the number of design changes that affect each class in a version control history; thereby, it estimates the importance of each class.

Our proposed technique likewise identifies important objects of key classes, which is described in Chapter 5. We will discuss the differences between existing key class identification approaches and our technique in Section 5.6.

2.3.2 Design Pattern Recovery

Design patterns have been widely utilized in various types of software to reuse expert design experiences and improve the quality of software products (e.g.,

GoF design patterns [94], PofEAA [95], POSA [96], etc.). Design patterns provide possible solutions/approaches to address common/recurring problems frequently encountered in specific contexts. Mined (recovered) design pattern instances from existing source code can be valuable tools for helping developers comprehend the structures and behavior of a system.

Numerous tools and techniques for recovering design patterns have been presented in the literature. Typically, there exist three aspects of pattern characteristics that distinguish a pattern from others: structural, behavioral, and semantic aspects. A survey paper by Dong et al. reported that the majority of the existing studies focus on the structural aspect because it can be more easily detected in program code compared with the other aspects [97].

Existing techniques that rely on the structural aspect characterize attributes and relationship of design elements (e.g., generalizations and associations between classes, the numbers of attributes and methods, etc.), and thereby identify design patterns. For example, Tsantalis et al. presented a technique for identifying design patterns (and modified versions thereof) based on similarity scoring [98]. Their technique models a subject system and design patterns as graphs (encoded in matrix forms) and calculates similarity scores between them to identify pattern instances.

Because some design patterns have similar (or exactly the same) structures, identifying those patterns solely from the structural aspect often produces a certain number of false positives [99,100]. Several studies exploit the behavioral aspect (e.g., method invocation histories) and the semantic aspect (e.g., naming conventions) to reduce the number of incorrect detection results. For example, Huang et al. characterized the behavioral aspects (e.g., entries/exits of operations and allocations/destructions of objects) of design patterns with Allen's temporal logics [101]. By utilizing combinations of the structural and behavioral aspects as matching criteria, they improved the scope and precision of design pattern recovery.

2.3.3 Architecture Recovery

Software architecture describes high-level abstractions of a system, which is crucial for many software development activities. However, conceptual architecture (existing in human minds or documentation) is often non-existent or outdated (architecture erosion [102]) because the structures and behavior of a system are continually changed along with system evolution. Manually recovering architecture documents requires a great deal of effort; e.g., for even medium-sized systems of 70–280 KLOC, the recoverers spent around 100 person-hours per system, on average, to obtain ground-truth architecture [103].

A large number of techniques for supporting software architecture reconstruction (SAR), which aims at reconstructing viable architectural views of a system, have presented in the literature [104]. Since the main topic in this dissertation is to recover behavioral design views of a system through program analysis, this section especially focuses on architecture recovery rather than architecture discovery among numerous SAR techniques.

While architecture discovery is a top-down process, architecture recovery is a bottom-up or hybrid (a combination of top-down and bottom-up) process. Architecture recovery techniques analyze software artifacts (e.g., source code and execution trace) and knowledge in order to (re)create architectural views of a system. Compared with design recovery techniques described in Section 2.2, architecture recovery techniques output much more highly abstracted views, such as component diagrams, architectural layer views, etc.

In what follows, we describe key related studies on architecture recovery. We will compare our technique for recovering summarized sequence diagrams with existing architecture recovery techniques in Section 6.6.

Recovering behavioral aspects.

Since the primary topic of this research is to recover behavioral design views, among numerous SAR techniques, SAR techniques for recovering or analyzing behavioral aspects are particularly related to our research.

Liu et al. presented a technique for recovering component behavioral models from execution traces [105]. Given a component configuration that defines mappings between classes and components and is obtained from documents, their technique first identifies component instances, in which a component instance corresponds to one independent run of a component, from entire execution traces. Then, identified component instances are transformed into hierarchical event logs based on calling relations among methods. Finally, their technique recovers behavioral models, which are represented as Petri nets, for each component via a process mining technique. More recently, Liu et al. proposed another technique that focuses on component interfaces [106]; it first identifies the interface of each component, a set of top-level methods exposed to other components, and then recover a behavioral model of each component via process mining.

DiscoTect is a tool for discovering architecture from running systems [107]. DiscoTect examine runtime events via state machines that are defined by users and represent architecture styles; this reveals discrepancies between conceptual architecture and concrete architecture.

Some studies proposed techniques for visualizing interactions among architectural elements in highly abstracted views, which is close to behavioral design

recovery techniques including our approach.

AVID is a tool for visualizing interactions at an architectural level [108]. Users first specify the architectural structure of a system, and then AVID visualize dynamic interactions among architectural elements. In addition, AVID offers sampling functionalities for reducing the total size of dynamic information to visualize [109]. AVID requires human expertise: it assumes that users are familiar with the architectural structure of a subject system; however, this is not true in practice, and thus requiring such human expertise can be a drawback.

Ducasse et al. presented a technique for visualizing condensed dynamic information in specialized views, named runtime polymetric views [110]. Runtime polymetric views visualize dynamic metrics of classes (e.g., #[invoked methods] and #[instances of each class]) and relationships thereof (e.g., inheritance and invocations). Visualizing condensed information (i.e., dynamic metrics) rather than entire traces improves the scalability of their technique.

Collaboration browser is a tool for visualizing collaboration patterns of interest [111]. The tool identifies collaboration patterns, which are equivalence classes of collaboration instances (sequences of messages among objects), via pattern matching regarding sender, receiver, and methods of messages. Users are allowed to specify collaboration patterns to visualize; selected patterns are visualized as interaction diagrams.

Hamou-Lhadj et al. presented a technique for recovering high-level behavioral models from execution traces [112]. To cope with the massiveness of execution traces, their technique filters out utilities that are identified on the basis of the #fan-in of each class. Dynamic interactions (instructions) among trace elements (e.g., packages and classes) are visualized in a UCM (use case map [113]) format.

Salah and Mancoridis presented a technique for constructing a hierarchy of dynamic views from execution traces [114]. To identify features, their technique uses marked traces that are established by manually marking the start and end points of each feature. By identifying a set of objects created or used by each feature, the technique obtains interaction relations among features. Dynamic interactions are visualized as a hierarchy of dynamic views: object-interaction, class-interaction, feature-interaction, and feature-implementation views.

Clustering approaches.

Whereas many architecture recovery techniques require human expertise as their input (this will be described in the later paragraph categorized as "other approaches"), some studies presented automated approaches to identifying architectural entities (components) by utilizing clustering techniques. As our proposed technique likewise clusters system entities (i.e., objects) for summa-

rizing interaction histories (described in Chapter 6), those SAR approaches are related to our approach in terms of clustering software entities.

Static dependencies (e.g., calling relations) are often analyzed for clustering, and sometimes similarities among identifiers or comments are utilized [115]. Many clustering approaches try to identify an architectural component as a highly cohesive and loosely coupled set of classes [116].

Choi and Cho presented a technique for identifying components by utilizing static dependencies among classes [117]. Their technique groups classes based on the degree of change impact among classes: if class A contains another class B, modifying A influences B; thus, the two classes A and B, which have a tightly connected relation (e.g., aggregation or generalization), are grouped into the same component. Mishra et al. clustered classes in a similar fashion in order to create reusable components [118].

Washizaki and Fukazawa presented a refactoring technique, named "extract component" [119]. The "extract component" identifies reusable components (sets of classes) by analyzing reachabilities on a CRG (class relation graph), a directed multigraph in which each node corresponds to a class and each edge represents a static dependency (e.g., inheritance and instantiation); then, it extracts identified components from program code.

Shatnawi et al. proposed a technique for identifying reusable components from APIs [120]. API classes frequently used together in many client applications are expected to form some components: their technique analyzes such a kind of frequency and utilizes static dependencies among classes in order to identify reusable components.

Allier et al. presented a meta-heuristic search algorithm for identifying components, which combines a genetic algorithm (GA) and simulated annealing (SA) [121]. The algorithm takes a trace set, which covers all the functions of a subject system, as their input; it identifies classes that are frequently used together in many traces (i.e., the distance of those classes is small) as a component.

Other approaches.

The survey by Ducasse and Pollet [104] reports that many architecture recovery techniques require human expertise (knowledge) as their input to guide SAR processes and validate the results [57, 107, 108, 122]. For example, human expertise is utilized for creating mappings between low-level entities (e.g., classes) and high-level architectural entities (e.g., components) or identifying architectural patterns in program code.

Richner and Ducasse presented a technique for recovering high-level views of systems based on static and dynamic information [57]. Their technique recovers architectural views in an interactive manner. Developers are allowed to

define views of interest (i.e., queries) with a logic programming language, which specifies elements (and granularity thereof) and relations to visualize.

Riva and Rodriguez presented a technique that provides linked (integrated) views depicting static and dynamic architectural information [122]. Users examine architectural documents in order to identify architecturally significant concepts, and define abstraction rules defining how to abstract the structures of architectural elements. According to the definitions, their technique extracts the static structures of architectural elements from source code, and visualize those in a graph format (a static view). Dynamic information (i.e., interactions among architectural elements) is recorded by instrumentation, and visualized in an MSC (message sequence chart) format (a dynamic view). The static view and dynamic one are synchronized at the same abstraction level; if a user expands an abstracted node (i.e., decreasing the abstraction level) in a static view, the corresponding node is likewise expanded in a dynamic view. The dynamic view offers simple functionalities for abstraction (i.e., grouping participants or contiguous messages) to support effective exploration.

2.3.4 Specification Mining

Specification mining techniques recover (extract) formal specifications (specs) from source code or execution traces. A large number of existing studies have presented automated techniques for mining API usage protocols that client applications ought to respect [51, 123]. Violating such protocols will lead to faulty behavior or program crashes. A well-known API usage protocol is that File#open(...) must be followed by File#close(...); violating this protocol causes a resource leak.

API usage protocols are sometimes (or often) implicit due to poor, outdated, and non-existent documentation. Specification miners analyze source code and/or execution traces, and try to discover such implicit API protocols. Mined protocols are represented in various formats; finite state machines [124, 125], labeled transition systems and formal contexts [126, 127], topic distributions [128], life sequence charts [129], etc. Those protocols can further be used for runtime verification (uncovering faulty behavior), document augmentation, etc.

There exist some studies that focus on other types of specifications rather than API usage protocols. For example, Daikon is a tool for inferring likely invariants, which represents logical properties that are always true for some program execution, at several program points (e.g., method entry/exit points) [130]. Brünink and Rosenblum presented a technique for mining performance specifications in order to detect performance problems in deployed systems [131]. APEx is an algorithm for inferring error specifications for APIs, which is used for uncovering

error-handling bugs [132]. These types of specifications (e.g., performance specifications) are difficult to be documented (in natural languages); thus, it is highly valuable to automatically extract those specifications (as models or properties) from code/traces and check conformance to mined specs.

3

Summarized Sequence Diagram Recovery

In this dissertation, we propose a fully automated technique for recovering a summarized sequence diagram with a central focus on horizontal reduction, which aims to help developers comprehend a behavioral overview of a system in an early stage of program comprehension.

This chapter describes the overview of our recovery technique. We explain each of the major components in our technique and relationship among those.

3.1 Overview

In this dissertation, we propose a summarized sequence diagram recovery technique for helping developers comprehend a high-level behavioral overview of a system. In this section, we explain the overall process flow of our sequence diagram recovery. To facilitate reader understanding, we use a Pac-Man game application as a running example.[1]

The overview of our recovery technique is shown in Figure 3.1. First, we model an execution trace (runtime information) with our behavior model (B-model) (the "Modeling System Behavior" part in Figure 3.1). Then, by analyzing dynamic properties obtained from B-model data, we identify core objects that play important roles in a system (the "Core Object Identification" part). In

[1]The running example is based on a simplified version of jpacman [133].

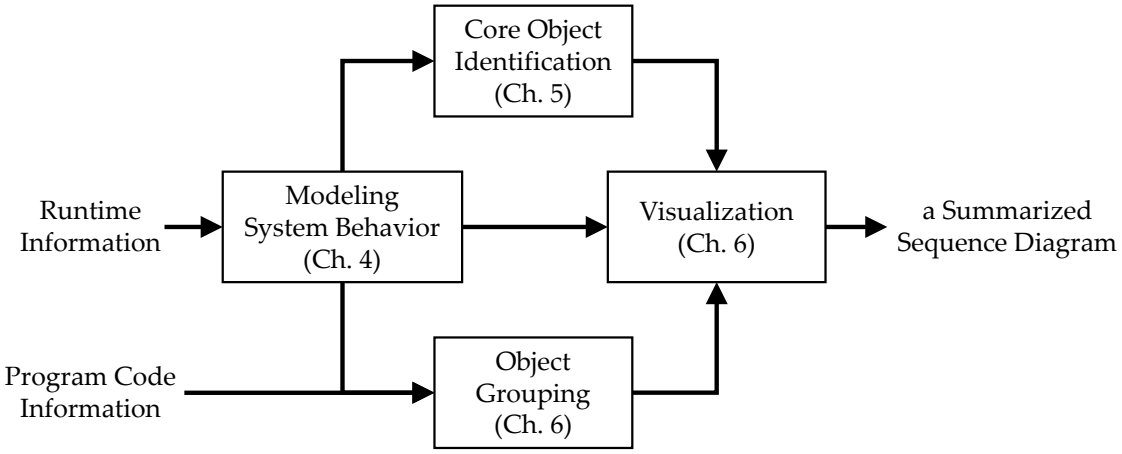

Figure 3.1: Overview of summarized sequence diagram recovery.

addition, we construct object groups corresponding to concepts based on design patterns identified in program code (the "Object Grouping" part). Finally, given B-model data and the results of core object identification and object grouping, we generate a summarized sequence diagram by visualizing intergroup interactions only among important object groups (the "Visualization" part).

In what follows, we describe the overview of each part shown in Figure 3.1 by referring to our running example of a Pac-Man game application. Note that more details of each part will be elaborated in Chapters 4 to 6.

3.2 Modeling System Behavior

Dynamic approaches to recovering behavioral diagrams take execution traces as their input. In general, those approaches record execution traces with the instrumentation of logging code. Although many existing studies have worked on execution traces, the formats of traces vary across their tracers.

We model an execution traces with our B-model in order to formalize runtime information in a tracer-independent form; this enables us to develop several dynamic analysis (trace analysis) algorithms for recovering a summarized sequence diagram, independent of various tracer specifications.

The B-model we propose consists of elements representing runtime events and attributes: e.g., EntryEvent (resp. ExitEvent) that represents "entry into a method/constructor" (resp. "exit from a method/constructor"); VariableDefinition (resp. VariableReference) that denotes "a value is assigned to a variable" (resp. "a value is read from a variable"). An execution trace can be represented as a B-model event sequence $ET = \langle b_1, b_2, ..., b_n \rangle$, where b_i is a B-model event element such as VariableDefinition. A B-model event element b_i can be converted

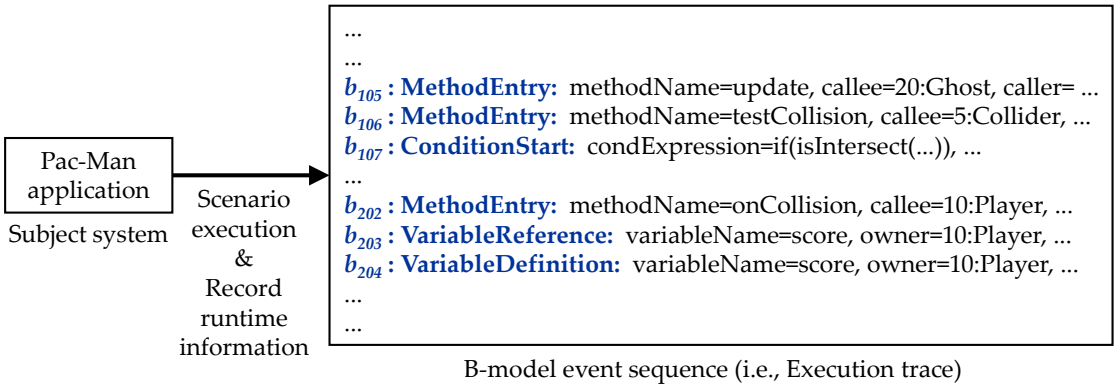

Figure 3.2: Generating an execution trace in the form of a B-model event sequence.

into sequence diagram elements such as a message; thus, we can easily convert an execution trace *ET* into a sequence diagram.

In our running example of a Pac-Man game application, during a scenario execution, many methods are invoked, such as "Ghost#update(...)" (that updates the rendering of ghost entities) and "Collider#testCollision(...)" (that tests whether a collision occurs). Besides, many variables are read and updated, such as "Player#score" that holds the score of a player. As a result, an execution trace corresponds to a scenario execution is recorded as a B-model event sequence shown in Figure 3.2.

More details of this part are elaborated in Chapter 4.

3.3 Core Object Identification

The most important concepts are implemented by very few key classes [38, 39]. Comprehending core objects of such key classes are important in an early stage of program comprehension.

We identify core objects by analyzing dynamic properties, such as reference relations and access frequencies, obtained from a B-model event sequence *ET*. For instance, we obtain reference relations among objects by examining VariableDefinition events in *ET*. We acquire the information on access frequencies and lifetimes of objects by counting VariableDefinition/Reference events and measuring the reference period of each object in *ET*. Our core object identification technique consists of 2 steps: pruning temporary objects and estimating the importance of non-temporary objects.

A large quantity of temporary objects, which do not play any important roles in a system, are generated during a program execution. Based on the information on reference relations and lifetimes of objects, if an object is referenced from no

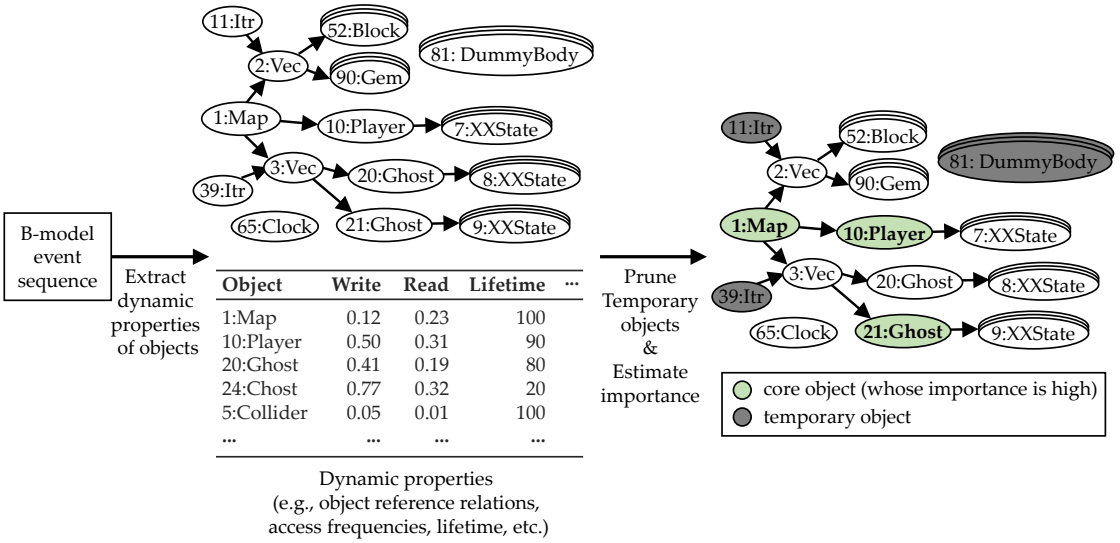

Figure 3.3: Core object identification. Each object is shown as "⟨object id⟩ : ⟨type name⟩". Each directed graph, called Object Reference Graph (ORG), depicts object reference relations; each circle (resp. each edge) corresponds to an object (resp. a reference via a field).

other objects and short-lived, we identify the object as a temporary and prune it.

After that, we estimate the importance of each non-temporary object. We consider that core objects that play important roles in a system are heavily accessed from other objects. Based on the access frequencies of objects, we assign an importance value to each non-temporary object; then, we identify objects whose importance values are especially high as core objects.

In our running example of a Pac-Man game application, core identification is performed as shown in Figure 3.3. By analyzing dynamic properties, we obtain reference relations among objects in the form of an object reference graph (ORG); e.g., a map object ("1:Map") has (indirect) references to block and ghost objects through vector objects. Objects that are referenced from no other objects and short-lived, such as iterator objects ("11:Itr"), are pruned as temporary objects. Then, frequently accessed non-temporary objects, such as a map object and a player object, are assigned high importance values, and thereby identified as core objects.

More details of this part are elaborated in Chapter 5.

3.4 Object Grouping and Visualization

To improve maintainability, in object-oriented programming, a concept is often divided into several classes by using design patterns. However, from the per-

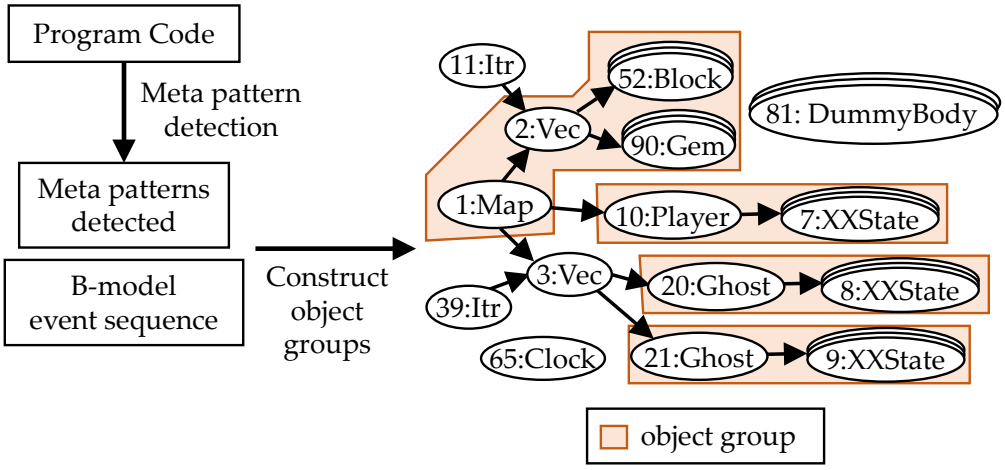

Figure 3.4: Constructing object groups based on Pree's meta patterns.

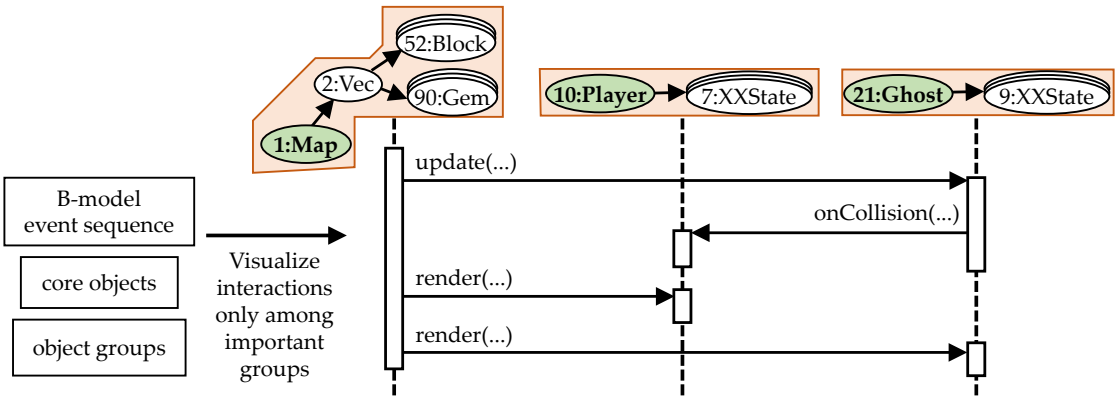

Figure 3.5: Visualizing interactions only among important object groups.

spective of behavioral comprehension, it produces numerous design elements to understand, which increases the amount of effort required for program comprehension.

In this part, we construct object groups at a concept-level based on Pree's meta patterns that are the most primitive design patterns. We detect all the occurrences of meta patterns in program code, and groups objects involved in the same pattern, which reunifies divided concepts as object groups.

Given the results of core identification, we identify important object groups. Then, we recover a summarized sequence diagram by visualizing intergroup interactions only among important groups, which provides a behavioral overview of a system at a concept-level.

In an early stage of program comprehension, where develops do not attain the detailed knowledge of a subject system, they prefer highly abstracted views to comprehend behavioral overviews rather than finer-grained views depicting nu-

merous detailed interactions. Our technique provides a summarized sequence diagram (an abstracted view) that depicts a behavioral overview of important concepts in a system; thus, it can be a valuable tool in an early stage of program comprehension.

Figure 3.4 shows the results of object grouping for our running example of a Pac-Man game application. For instance, a concept "player" is divided into a player object and several state objects by using the GoF (Gang of Four) state pattern. We identify and reunify (group) such divided concepts by grouping objects involved in the same pattern. The resulting group containing a player object ("10:Player") corresponds to the concept "player".

After object group construction, we identify object groups that contain core objects (shown in Figure 3.3) as important ones. By visualizing intergroup interactions only among important groups, we generate a summarized sequence diagram that depicts a behavioral overview of the Pac-Man game application (Figure 3.5).

More details of this part are elaborated in Chapter 6.

4

Modeling the Behavior of an Object-Oriented System for Behavioral Design Recovery

In this chapter, we propose a behavior model (B-model) that models the behavior of an object-oriented system. With the B-model, an execution trace, which is an input of dynamic analysis, can be formalized in a tracer-independent form, and can be easily converted into a sequence diagram.

We first explain the structure and attributes of the B-model. Then, we describe how B-model elements are converted into sequence diagram elements. To demonstrate the feasibility and sufficiency of the B-model, we show an application, which slices a recovered sequence diagram, built on the B-model.

This chapter is based on our journal paper "Reticella: An execution trace slicing and visualization tool based on a behavior model" [45] published in *IEICE Transactions on Information and Systems*, Copyright © 2012 IEICE. The paper is an extended version of our earlier work "Sequence diagram slicing" [46] appearing in the *Asia-Pacific Software Engineering Conference (APSEC)*, Copyright © 2009 IEEE.

4.1 Introduction

Dynamic techniques for recovering sequence diagrams must analyze execution traces to obtain runtime information on how objects/classes interact with one another, and convert them into sequence diagrams. Execution traces, in general, are obtained by the instrumentation of logging code. Most existing studies have developed their own tracers that record runtime events/information in specialized formats to suit for their purposes; the formats of execution traces vary across tracers (existing studies) [134].

In this chapter, we present a behavior model (B-model) for modeling an execution trace. An execution trace can be formalized in a tracer-independent format via the B-model; this enables us to develop execution trace analysis algorithms independent of tracers.

The primary objective of this research is to recover a summarized sequence diagram via program analysis. To this end, the B-model is required to satisfy the following properties.

- The B-model holds runtime information sufficient for traditional program analyses (e.g., data/control flow analyses).
- B-model data can be converted into UML sequence diagrams.

Because a sequence diagram depicts messages among objects (lifelines), recording only method-entry/exit events and caller/callee object identifiers is adequate for recovering a sequence diagram; however, recording more various types of dynamic properties is desirable in order to perform dynamic analyses (e.g., core object identification) more precisely. Thus, we have the B-model hold finer-grained runtime information sufficient for traditional program analyses.

There is a gap between the representation of an execution trace and that of a sequence diagram. To fill the gap, we define the structure and attributes of the B-model such that each element of the B-model can be easily/uniquely converted into an element of a sequence diagram.

To demonstrate the feasibility and sufficiency of the B-model, we show an application that performs program slicing [135,136] (one of traditional program analysis techniques) on a recovered sequence diagram. We show some examples of slice calculation by using some simple subject programs.

The remainder of this chapter is organized as follows. In Section 4.2, we define the structure and attributes of the B-model. Section 4.2.2 shows how B-model elements are converted into sequence diagram elements. To demonstrate the feasibility and sufficiency of the B-model, Section 4.2.2 shows an application of sequence diagram slicing. We describe key related work on formalizing runtime information in Section 4.4. Finally, Section 4.5 summarizes this chapter.

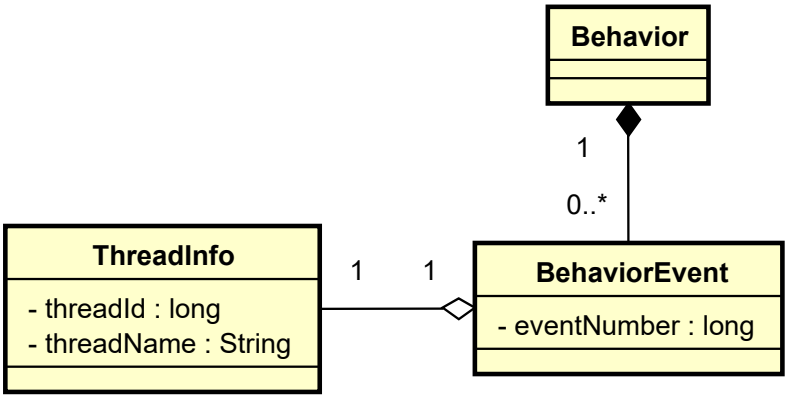

Figure 4.1: Structure of the B-model: core elements.

4.2 Behavior Model (B-Model)

4.2.1 Structural Definition

This section provides the definitions of the structure and attributes fo the B-model. We show the entire structure of the B-model in Figures 4.1 to 4.5.

The behavior of an object-oriented system can be represented as an instance of Behavior class shown in Figure 4.1. A Behavior instance (i.e., an execution trace ET) consists of instances of BehaviorEvent class: $ET = \langle b_1, b_2, ..., b_n \rangle$ where b_i is a BehaviorEvent instance. Instances of BehaviorEvent are categorized into four types of events as follows.

- Events related to method-entries/exits (Figure 4.2).
- Events related to controlling program flow (Figure 4.3).
- Events related to exceptions (Figure 4.4).
- Events related to variables (Figure 4.5).

Events related to method-entries (resp. method-exits) shown in Figure 4.2 corresponds to messages being sent (resp. return messages), which are primary elements of a sequence diagram. Examples of those events are MethodEntry and MethodExit classes (subclasses of MethodEvent class). Such method-related events have some attributes for drawing messages in a sequence diagram, such as method signatures and caller/callee object identifiers.

Events related to control flow, exceptions, and variables hold information required for program analyses such as data/control dependence analyses. Examples of those events are ConditionStart, ExceptionOccurrence, and VariableDefinition (subclasses of ConditionEvent, ExceptionEvent, and VariableEvent). We

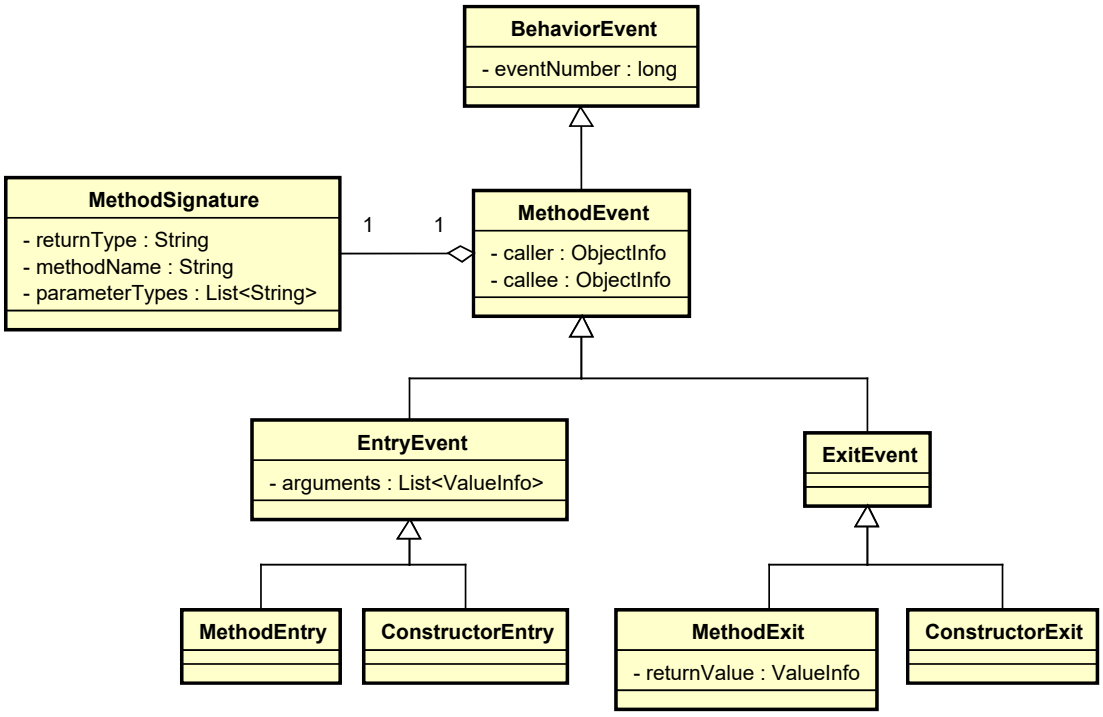

Figure 4.2: Structure of the B-model: method events.

can obtain control (resp. data) flow information from ConditionEvent and ExceptionEvent (resp. VariableEvent) instances.

As an example, we show a program code and the corresponding B-model event sequence generated by a program execution in Figure 4.6 and Figure 4.7.

4.2.2 Conversion into Sequence Diagram Elements

This section describes conversion rules from B-model elements into sequence diagram elements.

Given an execution trace $ET = \langle b_1, b_2, ..., b_n \rangle$, both each b_i and each ObjectInfo instance can be converted into elements of a sequence diagram. The conversion rules are shown in Table 4.1.

Primary elements of a sequence diagram are generated from ObjectInfo and MethodEvent instances. We convert an ObjectInfo instance into a lifeline in a sequence diagram. The identifier and type name of an ObjectInfo instance are displayed in the box on the top of a lifeline. A MethodEntry (resp. ConstructorEntry) instance is converted into a message (resp. a creation message) being sent. An ExitEvent instance corresponds to a return message. We identify sender and receiver lifelines of each message by MethodEvent#caller and

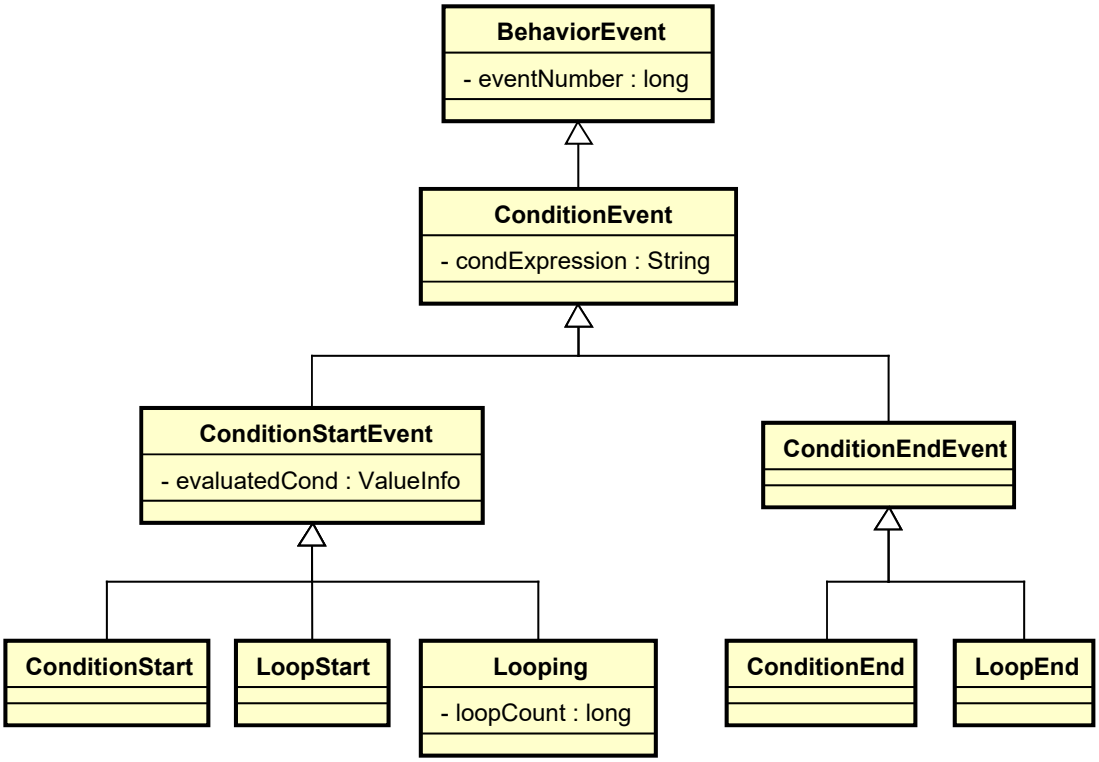

Figure 4.3: Structure of the B-model: condition events.

MethodEvent#callee attributes.

We depict events related to control flow and exceptions as note symbols (comments) in a sequence diagram, in order to facilitate program comprehension. Variable events are not depicted in a sequence diagram, in order to prevent the diagram being complicated.

Converting the B-model event sequence shown in Figure 4.7 results in a sequence diagram shown in Figure 4.8.

4.3 Application: Slicing a Recovered Sequence Diagram

In this section, we show an application that calculates a slice of a recovered sequence diagram in order to demonstrate the feasibility and sufficiency of the B-model. To calculate slices, the application is required to analyze data and control dependencies; we consider that this is an appropriate example to show the feasibility and sufficiency of the B-model.

Given variables of interest (called as a slicing criterion), the slicing appli-

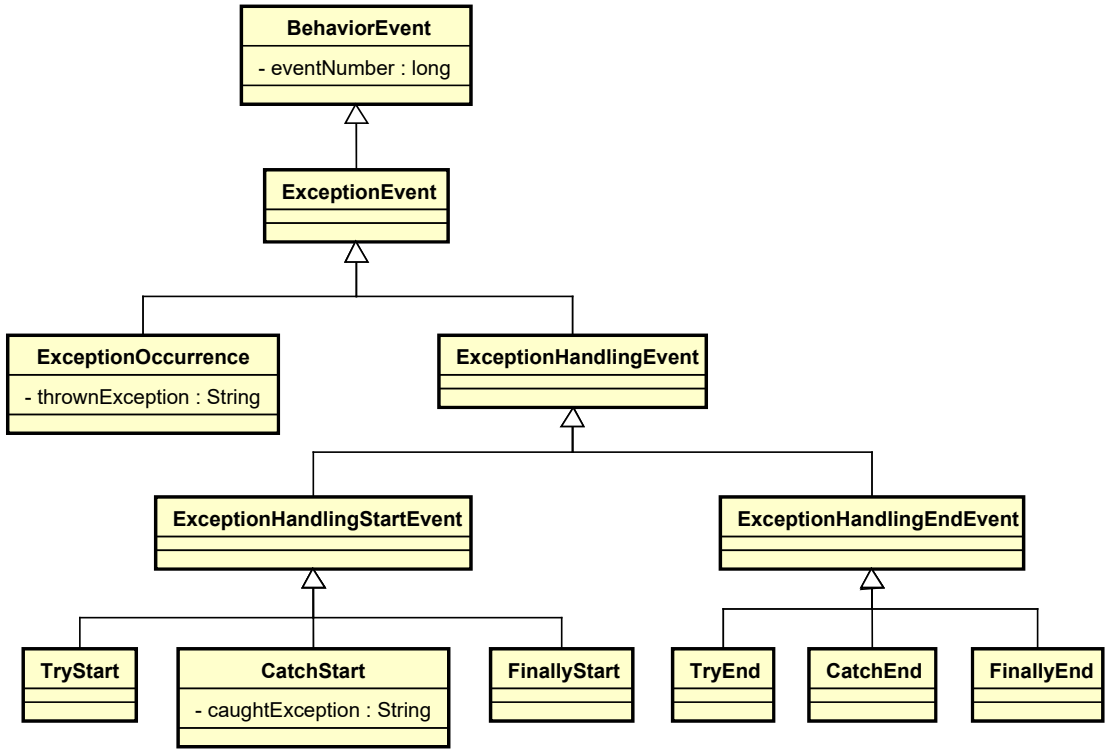

Figure 4.4: Structure of the B-model: exception events.

cation extracts execution trace fragments as a slice, and converts it into a sequence diagram. The application calculates a slice of a B-model event sequence $ET_s = \langle b_{i1}, b_{i2}, ..., b_{im} \rangle$ from an entire sequence $ET = \langle b_1, b_2, ..., b_n \rangle$. Then, the application obtains a sliced recovered sequence diagram by converting ET_s into a sequence diagram according to the rules shown in Section 4.2.2.

Our technique for recovering a summarized sequence diagram described in Chapters 5 to 6 is helpful for developers, who are unfamiliar with a subject system, to grasp a big picture of system behavior in an early stage of program comprehension. Meanwhile, the slicing application, shown in this section, generates a compact sequence diagram that depicts only a small portion of entire behavior developers are interested in (the portion affects the value of a given slicing criterion); thus, the slicing application is a possible solution to scalability issues of sequence diagram recovery when developers are (partly) familiar with a subject program and know which variables in the program should be inspected (e.g., in a debugging phase).

In the following sections, we describe the definitions of dependencies, algorithms for slice calculation, and examples of slice calculation with simple subject programs.

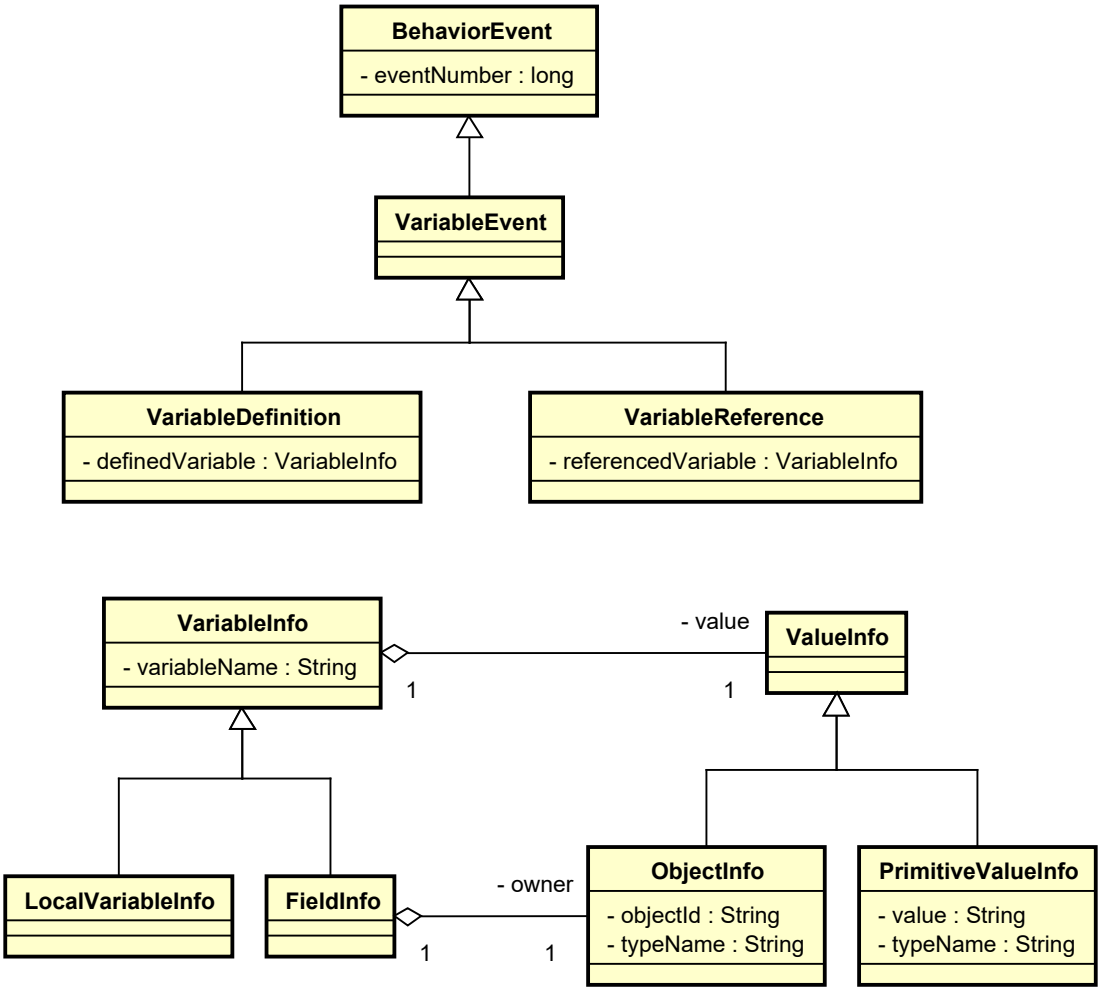

Figure 4.5: Structure of the B-model: variable events and related elements.

4.3.1 Dependency Definitions

We define the following three types of dependencies on the B-model for slice calculation.

- Data dependency
- Control dependency
- Start-End dependency

In what follows, let $ET = \langle b_1, b_2, ..., b_n \rangle$ be an execution trace and $1 \leq i, j \leq n \wedge i < j$ holds.

```
public class Sample {
   private int i;

   public static void main(String[] args) {
      Sample obj1 = new Sample();
      obj1.i = 1;
      obj1.method();
   }

   private void method() {
      if (this.i < 0) {
         System.out.println("this#i is negative.");
      } else {
         System.out.println("this#i is positive.");
      }
   }
}
```

Figure 4.6: Example code.

Table 4.1: Conversion rules from B-model elements to sequence diagram elements.

B-model element	Sequence diagram element
ObjectInfo	Lifeline
MethodEntry	Message (being sent)
ConstructorEntry	Creation message
MethodExit and ConstructorExit	Return message
ConditionEvent and ExceptionEvent	Note symbol

4.3.1.1 Data Dependencies

We define a total of five types of data dependencies. First, we define a data dependency regarding so-called def-use pairs.

Definition 4.1 (Data Dependency: Def-Use)
b_j is data dependent on b_i if all the following conditions hold.

- b_i is a VariableDefinition instance where a variable v is defined.
- b_j is a VariableReference instance where v is referenced.

Secondly, we define a data dependency regarding variable reference events.

b_{01} : **MethodEntry:** methodName=main, caller=(*system*), callee=static:Sample, ...
b_{02} : **ConstructorEntry:** methodName=Sample, caller=static:Sample, callee=1:Sample, ...
b_{03} : **ConstructorExit:** methodName=Sample, caller=static:Sample, callee=1:Sample, ...
b_{04} : **VariableDefinition:** variableName=obj1, value=1:Sample, ...
b_{05} : **VariableDefinition:** variableName=i, value=1, owner=1:Sample, ...
b_{06} : **MethodEntry:** methodName=method, caller=static:Sample, callee=1:Sample, ...
b_{07} : **VariableReference:** variableName=i, value=1, owner=1:Sample, ...
b_{08} : **ConditionStart:** condExpression=this.i < 0, evaluatedCond=false, ...
b_{09} : **VariableReference:** variableName=out, owner=static:System, ...
b_{10} : **MethodEntry:** methodName=println, caller=1:Sample, callee=2:java.io.PrintStream, ...
b_{11} : **MethodExit:** methodName=println, caller=1:Sample, callee=2:java.io.PrintStream, ...
b_{12} : **ConditionEnd:** condExpression=this.i < 0, ...
b_{13} : **MethodExit:** methodName=method, caller=static:Sample, callee=1:Sample, ...
b_{14} : **MethodExit:** methodName=main, caller=(*system*), callee=static:Sample, ...

Figure 4.7: B-model event sequence generated by the execution of the code shown in Figure 4.6.

Definition 4.2 (Data Dependency: Reference)
b_j is data dependent on b_i if the following conditions hold.

- b_i is a VariableReference instance where a variable v is referenced.
- Either of the following conditions holds.
  - b_j is a MethodEntry instance regarding a method m where m is invoked as "$v.m(...)$".
  - b_j is a ConditionStartEvent instance where v is used in the predicate.
  - b_j is a VariableReference instance regarding a variable t where v is referenced as "$v.t$".
  - b_j is a VariableReference instance regarding an array variable a where a is referenced with the index v.
  - b_j is an ExceptionOccurrence instance regarding a throw statement s_t where v is used in the s_t.

Finally, we define data dependencies regarding method/constructor invocations.

Definition 4.3 (Data Dependency: Method/Constructor Parameters)
When a method (or constructor) m is invoked, the value of each formal parameter of m is defined as the value of each actual parameter of the method invocation. Let b_i^s (resp. b_j^s) be the VariableReference (resp. VariableDefinition) instance of the sth formal (resp. actual) parameter of m. There exists a data dependency from b_i^s to b_j^s (i.e., b_j^s is data dependent on b_i^s).

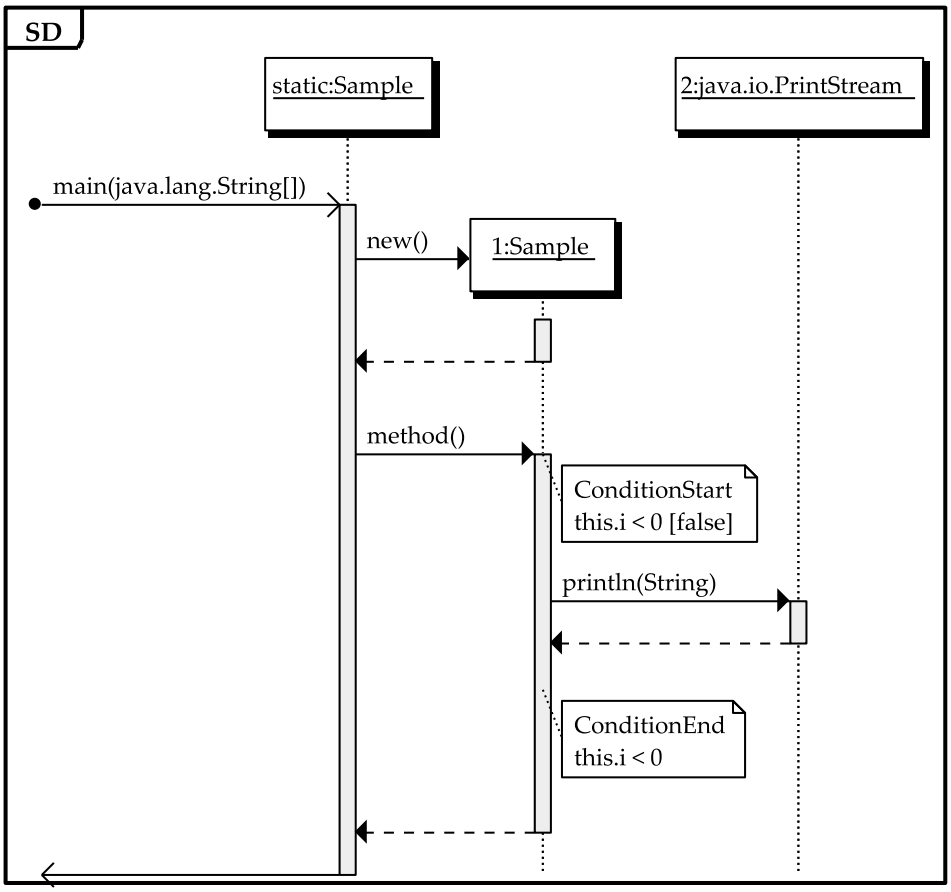

Figure 4.8: Sequence diagram generated from the B-model event sequence shown in Figure 4.7.

Definition 4.4 (Data Dependency: Method Return Value)
If the return value of a method invocation is referenced at b_j for the first time, and let b_i be an VariableReference instance regarding a variable referenced in the return statement, there exists a data dependency from b_i to b_j (i.e., b_j is data dependent on b_i).

Definition 4.5 (Data Dependency: Constructor Return Value)
If the newly created instance of a constructor invocation is referenced at b_j for the first time, and let b_i be the ConstructorExit instance of the constructor invocation, there exists a data dependency from b_i to b_j (i.e., b_j is data dependent on b_i).

4.3.1.2 Other Dependencies

In this section, we define control and start-end dependencies. First, control dependencies are defined as follows.

Definition 4.6 (Control Dependency: Conditional Statement)
b_j is control dependent on b_i if whether b_j occurs depends on the evaluation result of the predicate of a ConditionStartEvent instance b_i.

Definition 4.7 (Control Dependency: Try Statement)
b_j is control dependent on b_i if b_i is an ExceptionHandlingStartEvent instance and b_j occurs in the try/catch/finally body corresponding to b_i.

Definition 4.8 (Control Dependency: Exception Handling)
b_j is control dependent on b_i if b_i is an ExceptionOccurrence regarding an exception instance e, and b_j is an CatchStart instance where e is caught.

We define a start-end dependency so that a sequence diagram converted from a slice ET_s is *sound*; that is, we ensure that if there exists a start/entry event (i.e., an EntryEvent, ConditionStartEvent, or ExceptionHandlingStartEvent instance) in ET_s, the corresponding end/exit event (i.e., an ExitEvent, ConditionEndEvent, or ExceptionHandlingEndEvent instance) must exist in ET_s.

Definition 4.9 (Start-End Dependency)
b_i is start-end dependent on b_j if b_i is a start/entry event and b_j is the corresponding end/exit event.

Note that the dependency edge of a start-end dependency starts from the end/exit event b_j to the start/entry event b_i.

4.3.2 Slice Calculation

We calculate a slice of $ET = \langle b_1, b_2, ..., b_n \rangle$ by traversing dependence edges backward (i.e., calculate a backward slice). Let (b_i, V) be a slicing criterion where V is a set of variables of interest that can be referenced at the point of b_i occurring. The slice of ET regarding the slicing criterion (b_i, V) is calculated as Algorithm 4.1.

Given a slicing criterion (b_i, V), we first identify the latest VariableDefinition instance b_d^v that defines the value of $v \in V$ such that $d < i$ (ll.1–4); there is no re-definition of v between b_d^v and b_i. Then, starting from the identified

Algorithm 4.1 Calculating a slice of a B-model event sequence.

Input: an execution trace $ET = \langle b_1, b_2, ..., b_n \rangle$; a slicing criterion (b_i, V).
Output: a slice $ET_s = \langle b_{i1}, b_{i2}, ..., b_{im} \rangle$.

1: $D \leftarrow \emptyset$
2: **for each** $v \in V$ **do**
3: $\quad b_d^v \leftarrow$ the latest VariableDefinition instance regarding v before b_i s.t. both b_d^v and b_i occur on the same thread if v is a local variable; otherwise, v is a field variable.
4: $\quad D \leftarrow D \cup \{b_d^v\}$
5: $S \leftarrow \emptyset$
6: **for each** $b_s \in D$ **do**
7: $\quad S_s \leftarrow \{b_t \mid b_t$ is reachable by traversing dependence edges backwards from b_s. $\}$
8: $\quad S \leftarrow S \cup S_{b_s}$
9: $ET_s \leftarrow \langle b_{i1}, b_{i2}, ..., b_m \rangle$ s.t. $b_{ik}(1 \leq k \leq m) \in S$ and all the events in the sequence are sorted in chronological order.
10: **return** ET_s

VariableDefinition instance b_d^v, we calculate a slice by backwardly traversing dependence edges (ll.5–10).

4.3.3 Examples of Slice Calculation

In this section, we show examples of slice calculation with simple subject programs.

4.3.3.1 Producer-Consumer Program

The first subject is a traditional producer-consumer program that works on multi-threads.

The program of the producer-consumer problem, which we use in this example, has three key classes: Consumer, Producer, and SharedBuffer classes. The SharedBuffer class has a private field SharedBuffer#value that plays a role of a common buffer used between objects of the Consumer and Producer classes. A Consumer object retrieves a value from the common buffer via a method SharedBuffer#getValue(). A Producer object stores a value to the common buffer via a method SharedBuffer#putValue(...). Objects of the Consumer and Producer classes work on different threads, and try to get/put a value from/to the common buffer.

A recovered sequence diagram that represents the whole behavior of the Consumer/Producer program is shown in Figure 4.9. In Figure 4.9, each number between '[' and ']' indicates that the event occurred at the *i*th position in the execution trace.

There are two sets of accesses to the common buffer in Figure 4.9. A Consumer object first tries to read the value from the common buffer; however, because the common buffer is empty, the Consumer object waits for a value being stored. After a Producer object stores a value to the common buffer, the Consumer object is notified and reads the stored value from the common buffer.

We chose $(b_{158}, \{num\})$ as a slicing criterion to examine what behavior affected the value of the common buffer, where *num* is a local variable used in a method defined in the Consumer class, and holds the return value of SharedBuffer#getValue() method. The event b_{158} is a MethodEntry event for printing the value of *num* after the second set of accesses to the common buffer by Producer/Consumer objects.

The slicing result, a sliced sequence diagram, regarding the given slicing criterion $(b_{158}, \{num\})$ is shown in Figure 4.10. The resulting diagram depicts behavior related to the second set of accesses to the common buffer, and does not contain behavior of the first set of accesses. The size of the sliced sequence diagram is almost half of the original diagram size; our sequence diagram slicing application greatly reduces the size of a recovered sequence diagram.

4.3.3.2 Simple Text Editor

The second subject is a simple text editor program.

We apply our slicing algorithm to a larger program: a simple text editor program that is implemented by using Java Swing classes and has basic editing functionalities (e.g., changing font appearances as shown in Figure 4.11).

The execution scenario we analyze is as follows:

1. Type some words.
2. Change font families and font styles.
3. Type some words again.

A recovered sequence diagram that represents the whole behavior of the execution scenario is shown in Figure 4.12.

The sequence diagram shown in Figure 4.12 contains a total of 1,490 messages (4,217 B-model events). Although the execution scenario and the subject program are both small, a large number of messages is contained in the recovered diagram because several methods for obtaining and updating the current

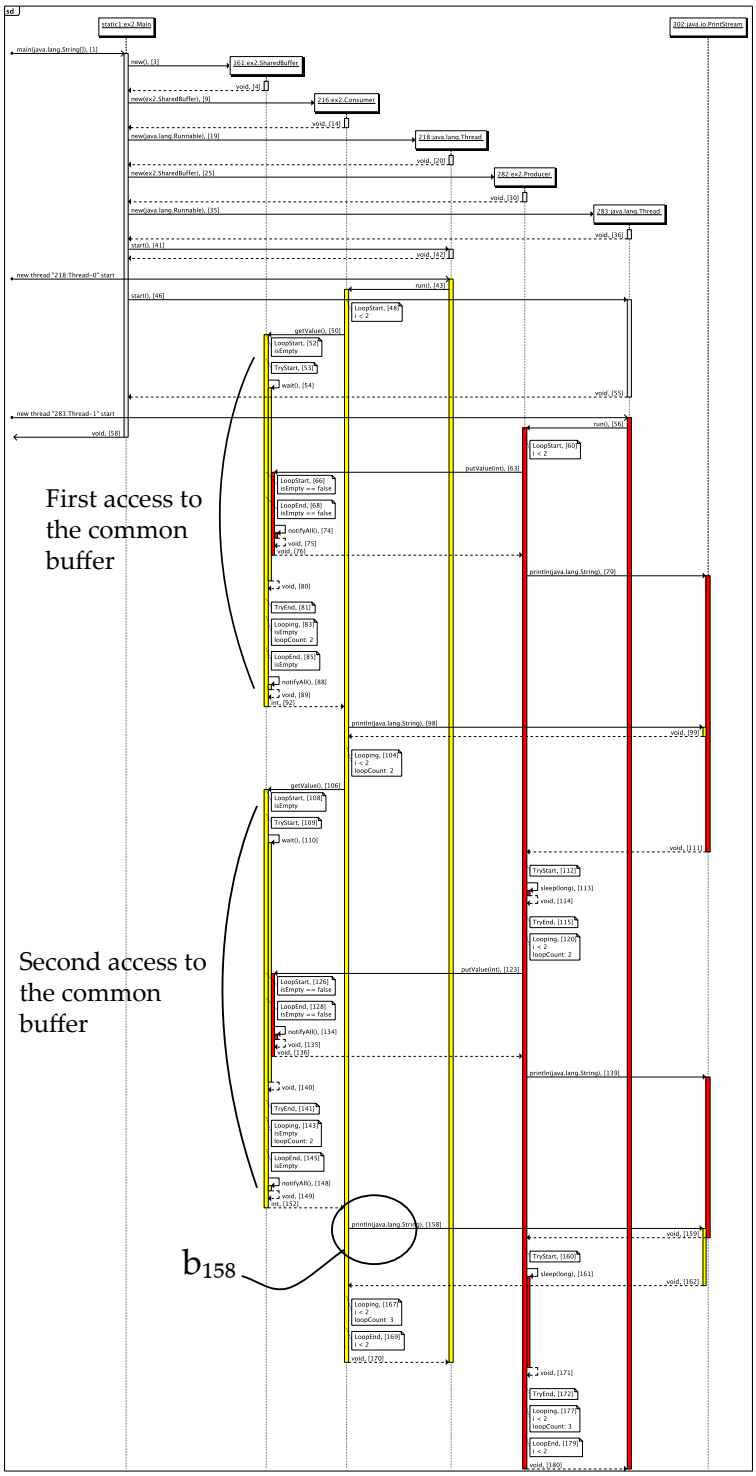

Figure 4.9: Sequence diagram that depicts the whole behavior of the consumer-producer program.

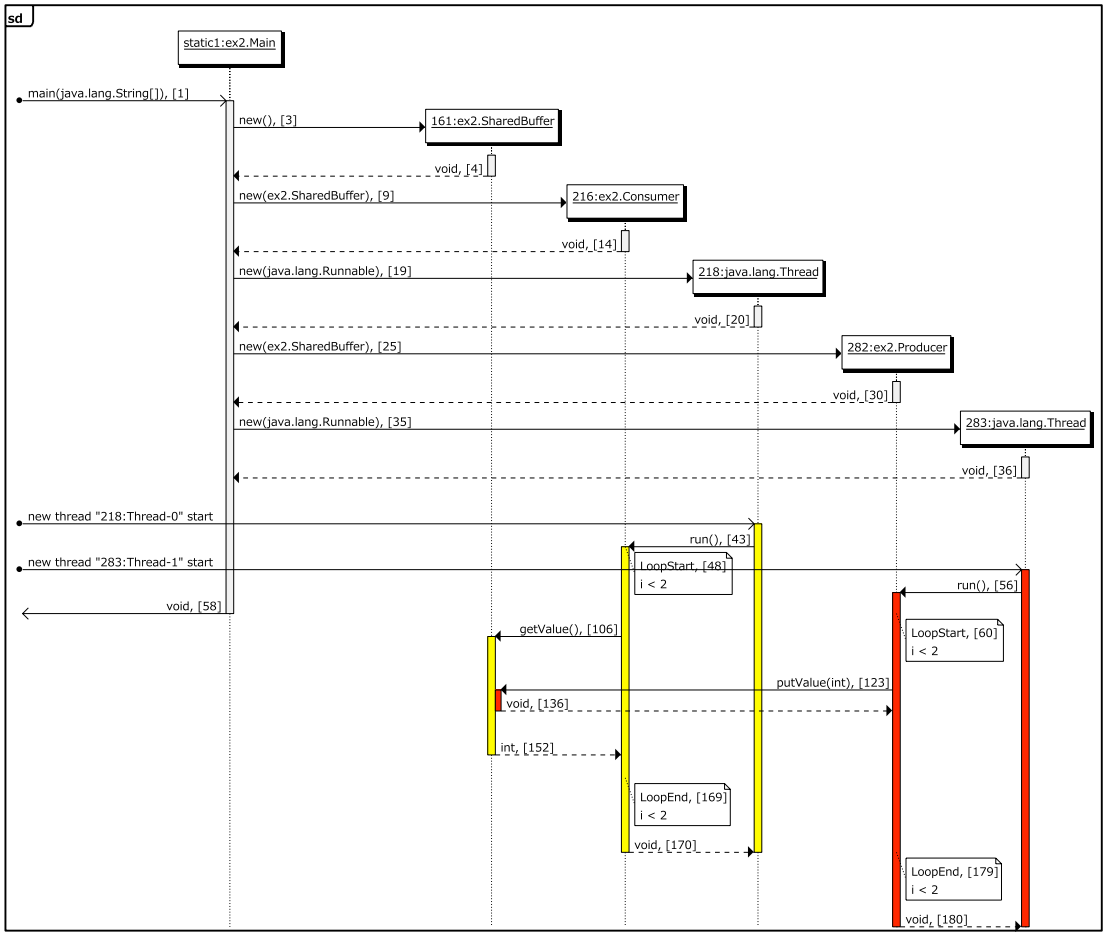

Figure 4.10: Slice of the behavior shown in Figure 4.9 regarding a slicing criterion $(b_{158}, \{num\})$.

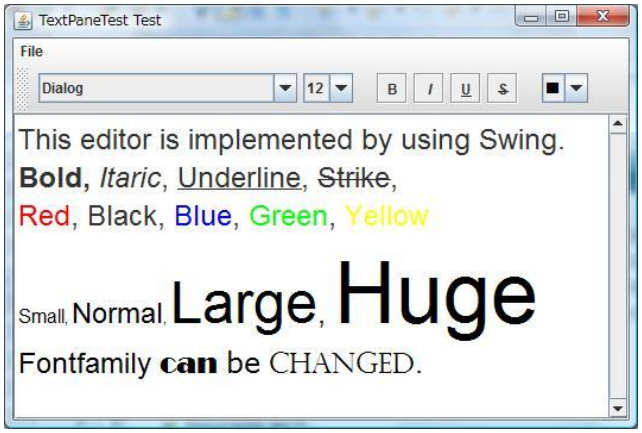

Figure 4.11: Screenshot of the simple text editor analyzed.

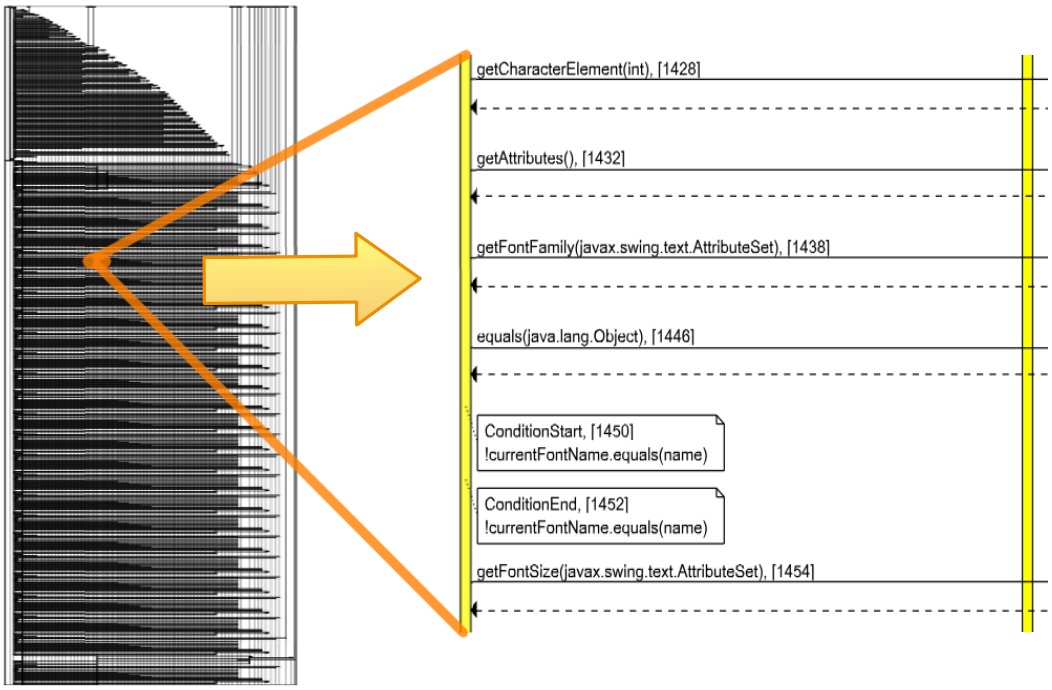

Figure 4.12: Sequence diagram that depicts the whole behavior of the simple editor program.

attributes of font styles were frequently invoked, every time we updated the caret position and typed characters. The large resulting diagram is not acceptable for manual investigation in a practical time.

We chose $(b_{4087}, \{currentFontFamily\})$ as a slicing criterion to examine behavior related to changing font families, where b_{4087} is an event that occurred after the second step in the execution scenario, and the variable *currentFontFamily* held information on the current font family setting. The resulting sliced sequence diagram is shown in Figure 4.13. The sliced diagram contains only a total of 24 messages (58 B-model events), and depicts behavior related to changing the current font family, such as getAttributes() and getFontFamily(...) method invocations.

The reduction rate of #messages is 1.61 % (calculated by [#messages in the sliced diagram] / [#messages in the original diagram] × 100). The reduction rate of the number of B-model events is 1.37 % (calculated [#events in the sliced trace ET_s] / [#events in the original trace ET] × 100). Our slicing application successfully extracted partial behavior of interest and recovered a sliced sequence diagram of a reasonable size. The size of the sliced sequence diagram is small sufficient for manual investigation in practical time.

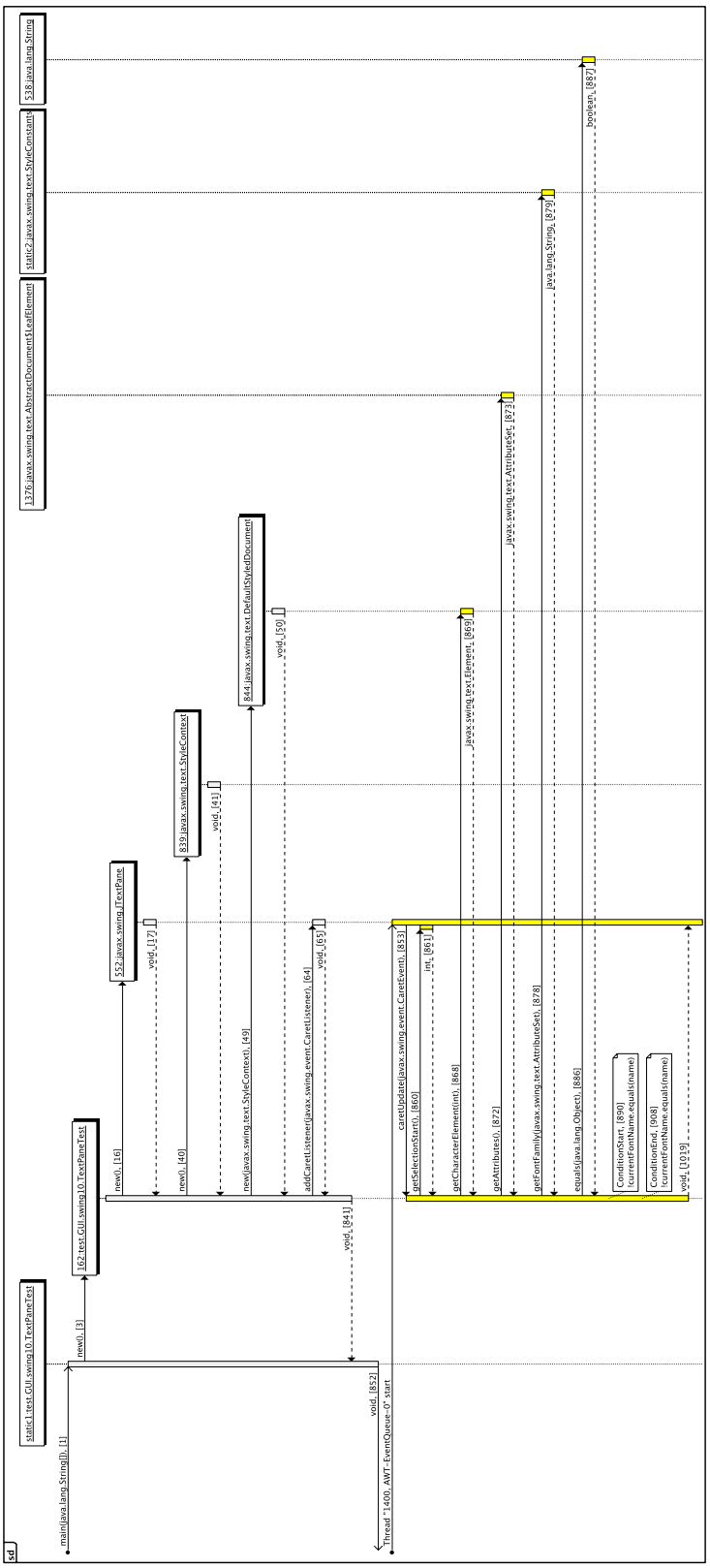

Figure 4.13: Slice of the behavior shown in Figure 4.12 regarding the slicing criterion $(b_{4087}, \{currentFontFamily\})$.

Through two slicing examples shown above, we confirm the feasibility and sufficiency of the B-model; it holds finer-grained information required for traditional program analyses, and can be easily converted into a sequence diagram for behavioral design recovery.

4.4 Related Work

Most existing studies on dynamic analyses have developed their own tracers, and built many applications on specialized formats of traces [134]. This hinders interoperability and makes it difficult to combine or reuse existing trace analysis algorithms in order to develop further effective techniques.

Some studies have presented trace formats for several purposes.

Hamou-Lhadj and Lethbridge proposed a metamodel for exchanging trace data in a compact but lossless format [134]. They model traces of routine (method) calls with their graph-based format, named compact trace format (CTF). A primary advantage of the CTF is scalability. They transform dynamic call trees into more compact ordered directed acyclic graphs (DAGs); that is, they represent similar (not necessarily identical) subtrees only once in a DAG. Alawneh et al. took a similar approach in order to define a metamodel, named MPI (message passing interface) trace format2 (MTF2), for representing traces of high-performance computing (HPC) in a compact format [137]. Because the size of an execution trace is quite large, the compactness of trace data is quite important for many applications working on traces [138, 139].

While our B-model holds finer-grained information regarding data and control flows, these metamodels do not have such kinds of information. Thus, these metamodels are not suitable for applications that need some program analyses.

Briand et al. defined a trace metamodel that models method-entries/exits, conditions, loops, and distribution information [140]. In addition, they defined a scenario diagram metamodel, and mapping between the trace metamodel and the scenario diagram metamodel with OCL (object constraint language [141]). With these metamodels and mapping, they achieved reverse-engineering of UML sequence diagrams for distributed Java software.

While our B-model holds runtime information at an event-level (e.g., variable definitions and references; method entries and exits), the trace metamodel by Briand et al. records statement-level information. Thus, developing trace analysis algorithms on B-model data is more intuitive and easier than using the metamodel by Briand et al. because code statement often consists of multiple expressions (i.e., algorithm developers are required to decompose statement-level information into more primitive ones). Moreover, the metamodel by Briand

et al. does not have information related to variable definitions and references; that is, we cannot obtain data-related information (e.g., data dependencies; access frequencies of each object (that are used in our core object identification technique described in Chapter 5)) from the metamodel by Briand et al.

The major difference between the metamodel by Briand et al. and other trace formats (including our B-model) is that the metamodel by Briand et al. has distribution information whereas other trace formats do not. By utilizing distribution information, Briand et al. achieved recovery of sequence diagrams for distributed systems.

Reiss and Renieris proposed various types of techniques for selecting, compacting, and encoding dynamic call trees [142]. For example, their technique represents a dynamic call tree in a string format; if a function A calls another function B and then another function C, the trace data (dynamic call tree) is represented as 'ABvCv', where 'v' is a marker for return. Then, they encode those string data by run-length encoding. Their techniques focus on the compactness of trace data as the CTF does. Compared with our B-model, the trace formats by Reiss and Renieris do not hold finer-grained information such as variable definitions and condition-block entries.

Brown et al. presented Step, a framework for the efficient encoding of general trace data [143]. The framework offers a domain specific language (DSL), named Step definition language (Step-DL), which enables us to flexibly define a trace format. As an experiment, with Step-DL, they defined a complete set of events generated by Java virtual machine (JVM) and a hierarchy for JVMPI (JVM profiling interface [144]) event data. Because information included in the B-model is a subset of runtime information obtained from JVM, we can define the structure and attributes of the B-model with Step-DL.

There are other types of trace formats that do not include routine (method) calls [134], such as PDATS [145] and HATF [146]. Because runtime information on method calls is indispensable for generating reverse-engineered sequence diagrams, these formats are not suitable for developing sequence diagram recovery techniques.

4.5 Summary

Techniques for recovering sequence diagrams through dynamic analysis takes execution traces as their input. There are diverse formats of execution traces; most existing studies have implemented own tracers, and built many applications on their specialized formats of traces.

We presented a behavior model (B-model) that models the behavior of an object-oriented system. The B-model formalizes an execution traces (runtime

information) in a tracer-independent format; this enables us to develop several dynamic analysis algorithms independent of tracers.

The B-model holds fine-grained runtime information required for traditional program analyses (e.g., runtime events related to control flows and variable definitions/references); we can build several types of dynamic analyses on the B-model. In addition, elements of the B-model can be easily converted into elements of a sequence diagram, which is suitable for developing behavioral design recovery techniques.

To demonstrate the feasibility and sufficiency of the B-model, we showed a sequence diagram slicing application built on the B-model. With the definitions of three types of dependencies, our slicing application can calculate a slice of a B-model event sequence. The application can obtain a slice of a recovered sequence diagram by converting the sliced event sequence into a sequence diagram. Through two slicing examples (a multi-threaded program and a GUI text editor program), we confirmed the feasibility and sufficiency of the B-model.

In the following chapters, we assume that an execution traces is represented in a form of a B-model event sequence. Several algorithms for recovering a summarized sequence diagram (e.g., core object identification) will be defined on the B-model.

5

Identifying Core Objects through Dynamic Analysis

The most important concepts of a system are implemented by very few key classes. To identify important objects of such key classes and comprehend their behavior are crucial in an early stage of program comprehension.

In this chapter, we present a method for identifying core objects by analyzing reference relations and dynamic properties. 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, by estimating the importance of each non-trivial object based on their access frequencies, 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 identifying core objects, compared with the state-of-the-art trace summarization technique.

This chapter is based on our journal paper "Identifying core objects for trace summarization by analyzing reference relations and dynamic properties" [47] published in *IEICE Transactions on Information and Systems*, Copyright © 2018 IEICE. The paper is an extended version of our earlier work "Identifying core objects for trace summarization using reference relations and access analysis," [48] appearing in the *Annual Computer Software and Applications Conference (COMPSAC)*, Copyright © 2017 IEEE.

5.1 Introduction

The most important concepts of a system are implemented by very few key classes [38, 39]. In an early stage of program comprehension, developers try to identify core (important) objects of such key classes and comprehend a behavioral overview of a system.

In this chapter, we present a technique of identifying core objects for trace summarization by analyzing the reference relations and dynamic properties. 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 [147]. 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. Identified core objects will be used in Chapter 6 to obtain a horizontally compactified version of a reverse-engineered sequence diagram that contains system's key behavior comprised of messages from among the core objects.

We applied our core identification technique to traces of various open-source software systems to evaluate its feasibility and effectiveness. The results showed that our technique achieved superior performance compared with the state-of-the-art trace summarization technique in terms of identifying core objects.

Our technique outputs an importance-based object ranking. In the experiment, our technique identified all the core objects within top 7–222 (108 on average) important objects, whereas that of the state-of-the-art technique ranged 148–4,378 (998 on average). Moreover, 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 effectively identify core objects with a small overhead and is expected to be a valuable tool in an early state of program comprehension.

The main contributions of this chapter 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.
- We demonstrate the feasibility and the effectiveness of the proposed technique through experiments with various open-source software systems.

The remainder of this chapter is organized as follows. Section 5.2 details the proposed technique. Section 5.3 briefly describes how to visualize a summarized sequence diagram by using the results of our core identification technique. In Section 5.4, we evaluate our technique through experiments, and Section 5.5 discusses threats to validity. Section 5.6 describes key related works. Section 5.7 summarizes this chapter.

5.2 Identifying Core Objects by analyzing Reference Relations and Dynamic Properties

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 Section 5.4.3.2.)

We elaborate each of the steps in the following sections with a running example shown in Figure 5.1.

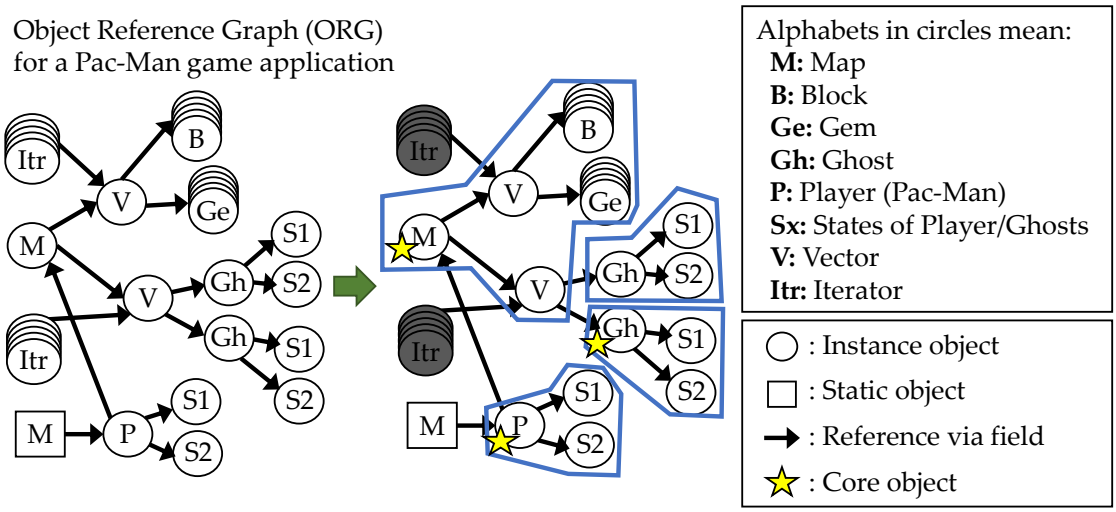

Figure 5.1: Running example (Pac-Man game example).

The left graph in Figure 5.1 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.

5.2.1 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.
- ***Captured*:** A state other than those listed above.

Algorithm 5.1 Reference escape analysis.

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

```
1:  for all b_i ∈ ET do
2:    O_referenced ← all the static objects referenced at b_i
3:    for all o ∈ O_referenced do
4:      EscapeState(o) ← "GlobalEscape"
5:  for all b_i ∈ ET do
6:    if b_i is ConstructorEntry then
7:      o_created ← an object created at b_i
8:      EscapeState(o_created) ← "Captured"
9:    else if b_i is VariableDefinition then
10:     f ← the field assigned at b_i
11:     o_owner ← the owner object of f
12:     o_assigned ← the value assigned to f
13:     if EscapeState(o_owner) == "GlobalEscape" then
14:       EscapeState(o_assigned) ← "GlobalEscape"
15:     else
16:       EscapeState(o_assigned) ← "ReferenceEscape"
```

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) described in Chapter 4. An execution trace can be represented in the form $\langle b_1, b_2, ..., b_n \rangle$, where b_i is an event element in B-model.

Algorithm 5.1 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. Secondly, if temporary objects have mutual or

Algorithm 5.2 Lifetime analysis.

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

```
1: for all b_i ∈ ET do
2:   if b_i is ConstructorEntry then
3:     o_created ← the object created at b_i
4:     CreatedAt(o_created) ← i
5:   else
6:     for all o_k referenced at b_i do
7:       LastAccessedAt(o_k) ← i
8: for all object o_s do
9:   Lifetime(o_s) ← LastAccessedAt(o_s) − CreatedAt(o_s)
```

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 5.2 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.

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

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 Section 5.4.3.3.

Note that both Algorithm 5.1 and Algorithm 5.2 analyze the events of *VariableReference/Definition* regarding 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 Figure 5.1, 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 Figure 5.1.

5.2.2 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 Section 5.4.3.3.

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

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

$WF(o_i)$, $RF(o_i)$, 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 from one another, the unity-based normalization is necessary. Note that if either $WF(o_i)$, $RF(o_i)$, or $MIF(o_i)$ is zero, $Importance(o_i)$ is likewise zero. If the maximum count is equal to the minimum count, then the frequency value is one (e.g., if $WC_{\max}(\boldsymbol{O}) = WC_{\min}(\boldsymbol{O})$, 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 Figure 5.1, 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 Figure 5.1 indicates that the object is a core object.

5.3 Visualizing a Summarized Sequence Diagram by using Identified Core Objects

In this section, we briefly describe how to visualize a summarized sequence diagram by using the results of our core identification technique.

We will present a technique for grouping objects at a concept-level in Chapter 6. Given the results of core object identification, we identify important object groups in which each group contains a core object. Then, we obtain a summarized sequence diagram that depicts a behavioral overview of a system by visualizing interactions only among important groups.

In the running example, the object reference graph is decomposed into four object groups by object grouping, as shown on the right of Figure 5.1. Groups that contain core objects are identified as important ones. The resulting summarized sequence diagram will depict interactions only among the important object groups.

5.4 Experiment

We have implemented our technique as a tool. Our tool records runtime information based on the B-model. It then performs core object identification, as described in the previous sections. To weave logging codes into the target system, our tool uses *SELogger* [148] which is a part of *REMViewer* [149]. *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 selected the utilityhood-based trace summarization proposed by

Hamou-Lhadj and Lethbridge [20], which is the most promising approach for identifying core-behaviors among existing techniques, as the baseline technique.

5.4.1 Research Questions and Evaluation Approaches

RQ_1: Compared with the baseline technique, how effective is our approach in terms of identifying core objects?

Motivation:

Our primary concern and objective are to identify core objects that implement the core concepts of a system and play important roles in an execution scenario. In RQ_1, we investigate the effectiveness of our proposed technique in terms of the performance of core object identification.

Evaluation Approach:

In our technique, if the importance value of an object is greater than the threshold I_t, the object is identified as a core object. Meanwhile, in the baseline technique, if an object does not receive any methods whose utilityhood value is smaller than the threshold U_t, the object is not depicted in a summarized sequence diagram; the object is regarded as a non-core object. In both of the baseline technique and our technique, all objects are ranked in the order of importance (utilityhood).

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 I_t and U_t, we evaluate the trade-off relationship between the top-k recall and the value of k. As described in Section 5.3, top-k important objects will be included in a resulting summarized sequence diagram; the top-k recall reflects the usefulness of the resulting diagram, and the k value affects the horizontal size of the resulting diagram.

RQ_2: Do both of the temporaries removal and the non-core objects elimination contribute to the performance of core object identification?

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 how many objects are removed as non-core objects for each step since it directly affects the final results of core object identification.

RQ_3: What is the effect of varying the tunable parameters of our technique on the core identification 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 core identification 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; secondly, 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.

5.4.2 Experimental Setup

5.4.2.1 Subject Systems and Execution Scenarios

The subject systems and execution scenarios are as shown in Table 5.1. For each subject, we selected a representative execution scenario.

Table 5.1: Subject systems and execution scenarios. In the "KLOC" column, lines of test code are not counted.

Project	Ver.	KLOC	Execution scenario
jpacman [133]	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 [150]	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 [151]	7.6	138.9	launch a demo application; draw a rectangle; draw a rounded-rectangle on the right side; quit the application
jEdit [152]	5.3.0	186.7	launch the application; type some sentences including line breaks; quit the application
JMeter [153]	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 [154]	1.9.8	224.3	build Ant itself (i.e., execute the 'build' task defined in the build.xml)

5.4.2.2 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 5.2 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 frames per second (FPS)).

As for *JHotDraw*, since the documentation and source code comments written by its developers provide the description of the important classes for its

Table 5.2: 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*

architecture, we define the important classes as our ground truth.

For other subject systems, we use the definition of important classes provided by Şora [41, 88]. Şora extracted the classes that are important to comprehend design overviews from the documentation or developer tutorials of the subject systems. Note that Şora also defined the important classes for JHotDraw; however, we re-extract important classes from the source code comments because the definition by Şora was for an older version of JHotDraw.

Of the sets of important classes extracted from execution scenarios, documents, and the definitions by Şora, 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 top-k recall, for each class c defined in Table 5.2, we consider the class c is covered if at least one object of a sub-type of c (including itself) is contained in top-k important objects. Formally, the top-k recall is

calculated as follows:

$$Recall@k = \frac{\Sigma_{c \in GT} \text{isCovered}(c, O_k)}{\mid GT \mid}$$

If at least one object of a sub-type of c (including itself) is contained in the set of the top-k important objects O_k, then $\text{isCovered}(c, O_k) = 1$; otherwise, $\text{isCovered}(c, O_k) = 0$.

5.4.2.3 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.

5.4.2.4 Settings of Tunable Parameters

Our technique has two types of tunable parameters as described in the description of RQ_3 (see Section 5.4.1). 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 core identification performance.

5.4.2.5 Settings of Baseline Technique

For comparison, we apply the baseline technique [20] 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 and Lethbridge [20]: Polymorphic calls are resolved by Rapid Type Analysis (RTA) [155] 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 and Lethbridge ignore methods of inner classes, we do not ignore those. This is because some of the core classes shown in Table 5.2 are implemented as inner classes.

5.4.3 Results

5.4.3.1 Answer to RQ_1

The recorded runtime information is shown in Table 5.3. The numbers of messages and objects affect the size of a 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 top-k recall and the value of k are as shown in Figure 5.2. The dashed line in Figure 5.2 is a random classifier that randomly classifies whether each object is important.

Table 5.3: Recorded runtime information. The "Objects" column does not include objects of collection libraries.

Project	Total events	Messages	Loaded classes	Objects
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

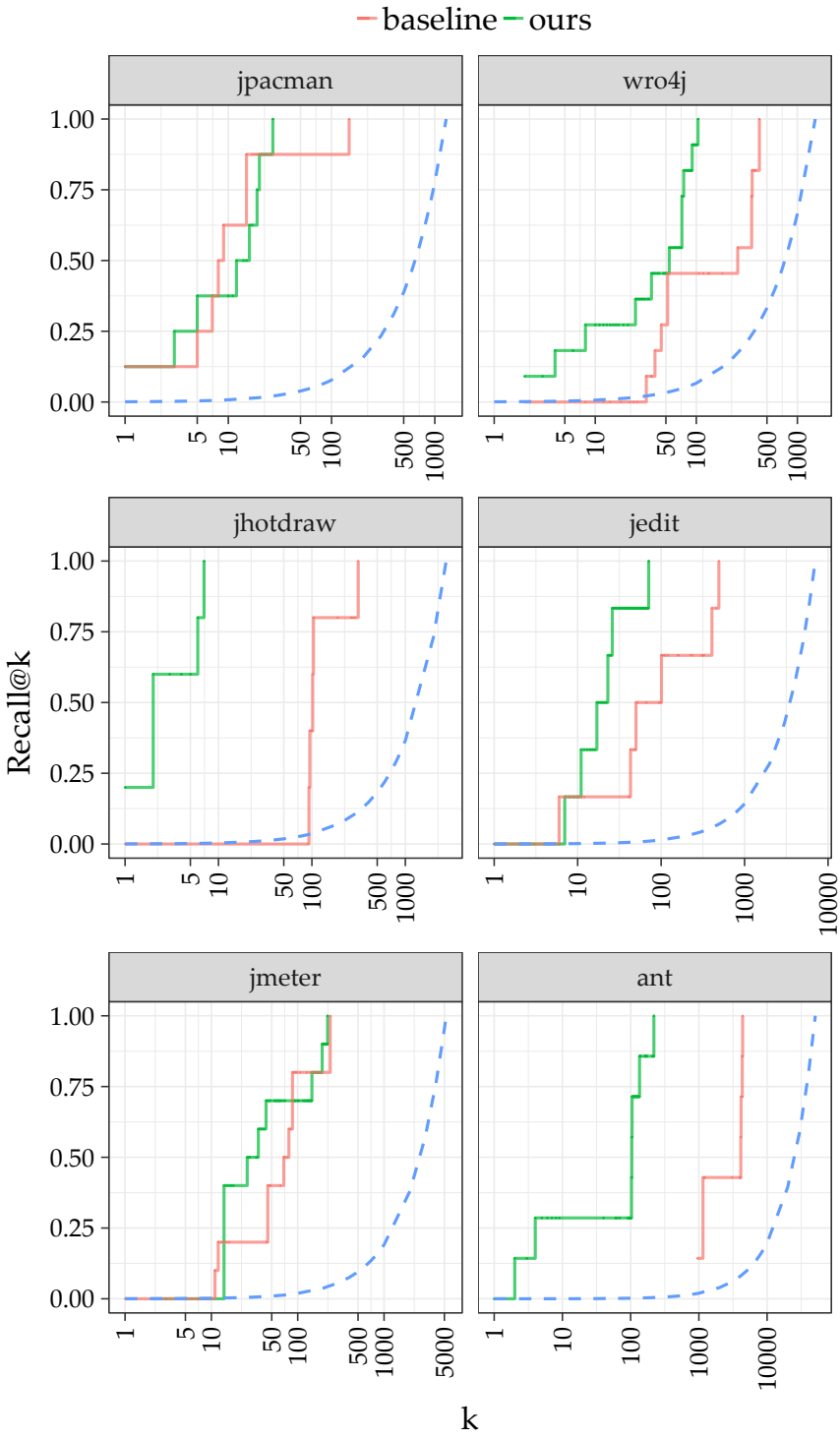

Figure 5.2: Core identification performance.

Table 5.4: Values of k at $Recall = 1$.

Project	Ours	Baseline
jpacman	27	148
wro4j	104	421
JHotDraw	7	313
jEdit	71	492
JMeter	222	238
Ant	219	4,378
Average	108	998

As shown in Figure 5.2, for all the subject systems, our technique reaches $Recall = 1$ with a much smaller value of k, compared with the baseline. The values of k at $Recall = 1$ are shown in Table 5.4. Table 5.4 shows that our technique can identify all the core objects within top 7–222 (108 on average) important objects, whereas that of the baseline ranges 148–4,378 (998 on average). From the results, we confirm that our technique is more effective in terms of identifying core objects, compared with the baseline technique.

Some core objects were difficult to be identified based solely on access frequencies. Those objects are roughly categorized into the following two types.

(1) Some important objects wrap (or serve as proxies for) other objects. Such objects delegate requested tasks to wrapped objects; the access (esp. write) frequency of the wrapper objects (i.e., core objects) is relatively lower, whereas that of the wrapped objects becomes higher. Thus, those important wrapper (proxy) objects are difficult to be identified by our technique.

(2) Some important objects are almost stateless, aggregate many other objects, and complete some complex tasks by a combination of a bunch of method invocations on the aggregated objects. Those important objects play important roles in a system; however, they are less frequently accessed from other objects. Thus, such important objects are difficult to be identified based on access frequencies.

The number of the above-mentioned types of core objects is expected to be small because our technique achieved a high performance as shown in Figure 5.2. We consider that combining other types of metrics (e.g., network metrics [41] on ORG) could alleviate the risk of such false negatives.

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 object ranking is relatively small.

Varying the threshold I_t (U_t) changes the number of objects regarded as core objects. As described in Section 5.3, the number of core objects affects the horizontal size (the abstraction level) of the resulting summarized sequence diagram. Thus, the number of ties affects the degree of the change of #lifelines when varying the threshold to raise/reduce the abstraction level of the resulting summarized diagram. We elaborate this finding in the following paragraphs.

Let $U_{M_1}, U_{M_2}, ..., U_{M_M} (s.t., U_{M_i} < U_{M_{i+1}})$ be all the possible values of utilityhood, and $I_{O_1}, I_{O_2}, ..., I_{O_N} (s.t., I_{O_j} < I_{O_{j+1}})$ 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 $\#\text{core}(U_t = U_{M_i})$ (resp. $\#\text{core}(I_t = I_{O_j})$) be #core (the number of objects regarded as core objects) when the value of U_t (resp. I_t) is set to U_{M_i} (resp. I_{O_j}).

Figure 5.3 shows the distribution of the difference of #core between two adjacent threshold values; i.e., the difference between $\#\text{core}(U_t = U_{M_i})$ (resp. $\#\text{core}(I_t = I_{O_j})$) and $\#\text{core}(U_t = U_{M_{i+1}})$ (resp. $\#\text{core}(I_t = I_{O_{j+1}})$). From the Figure 5.3, 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 #core is much smaller in our technique, compared with the baseline; the #core is changed by 1–34 in our technique and by 1–1,826 in the baseline.

Thus, changing the threshold U_t can causes a drastic change of the abstraction level of the resulting summarized diagram; it is difficult to change the threshold value flexibly in the baseline technique. Meanwhile, the degree of the change of #core 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. It would be appropriate to choose the higher-importance point as the threshold value in an earlier stage of program comprehension. Along with increasing the degree of understanding, the developer can change the threshold value to the lower-importance point.

Our technique is more effective in terms of identifying core objects, compared with the baseline technique. Our technique can identify all the core objects within top 7–222 (108 on average) important objects, whereas that of the baseline ranges 148–4,378 (998 on average).
Moreover, our technique is capable of changing the abstraction level of a resulting summarized sequence diagram more flexibly, compared with the

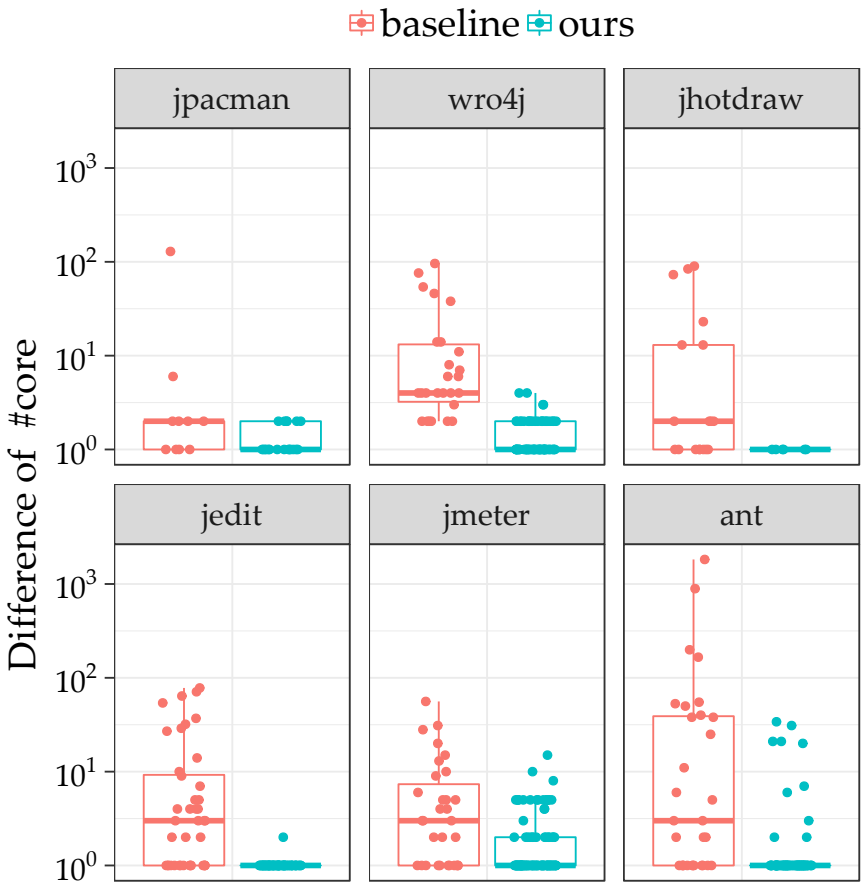

Figure 5.3: Distribution of the difference of #core between two adjacent threshold values (i.e., the difference between #core($U_t = U_{M_i}$) (resp. #core($I_t = I_{O_j}$)) and #core($U_t = U_{M_{i+1}}$) (resp. #core($I_t = I_{O_{j+1}}$))).

baseline technique.

5.4.3.2 Answer to RQ_2

Table 5.5 shows the number of (non-)temporaries and the rate of pruned objects. A significant number of temporary (unimportant) objects are pruned: on average, 9,791 objects (67.4 % of all the objects) are pruned. On the other hand, the performance of eliminating low-importance objects is shown in Table 5.4. Eliminating low-importance objects further reduces the number of core-candidate objects from 1,648 up to 108 on average. Both of the two reduction steps reduce a great number of objects as shown in Table 5.5 and Table 5.4; therefore, both of

Table 5.5: Results of pruning temporaries. The "Rate of pruned" column is calculated by #(temporaries)/#(all objects).

Project	#temporaries	#non-temporaries	Rate of pruned [%]
jpacman	733	556	56.9
wro4j	1,120	384	74.5
JHotDraw	2,157	579	78.8
jEdit	5,765	1,265	82.0
JMeter	914	4,331	17.4
Ant	48,055	2,773	94.5
Average	9,791	1,648	67.4

the steps contribute to the final results (the core identification performance).

To investigate the individual contribution of eliminating low-importance 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 a resulting summarized sequence diagram, numerous 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 low-importance objects elimination.

Both pruning temporaries and eliminating low-importance objects contribute to the core identification performance. Both of the reduction steps are essential to enable developers to flexibly change the abstraction level of a resulting summarized sequence diagram.

5.4.3.3 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 Figure 5.4, Figure 5.5, and Figure 5.6, resp. Enlarging the weight w_w, the core identification performance rises for four projects and decreases for two projects (Figure 5.4). Increasing the weight w_r makes the core identification performance better for two projects and worse for two projects (Figure 5.5). As for w_{mi}, increasing

the weight makes the performance worse for two projects and has few positive effects; there are almost no performance improvements by increasing the w_{mi} (Figure 5.6).

Enlarging the weight for write access w_w leads a good effect on the core identification 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.

Increasing the value of w_{mi} causes a negative effect on the performance. Wrapper, proxy, or utility objects, which are almost stateless, are more likely to be regarded as important ones if we enlarge w_{mi}. Those types of objects are normally non-core objects, although a few core objects serve as wrappers as mentioned in Section 5.4.3.1. Thus, enlarging w_{mi} produces much noise, and leads to performance degradation.

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.

Secondly, Figure 5.7 shows the effect of varying the lifetime threshold for short-lived objects $L_{t\text{-short}}$ with fixing other parameters as default values. From Figure 5.7, 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 core identification performance. We consider that 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 core identification performance did not vary. For only one project, the core identification performance varied; however, the change was very slight and ignorable (the value of k (i.e., #core) at $\mathit{Recall} = 1$ varied by only 1). We consider that this result is attributed to the situation such that, for our subject systems, there were no important objects that were the *root*s of successive procedures (see Section 5.2.1). 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 $L_{t\text{-long}}$.

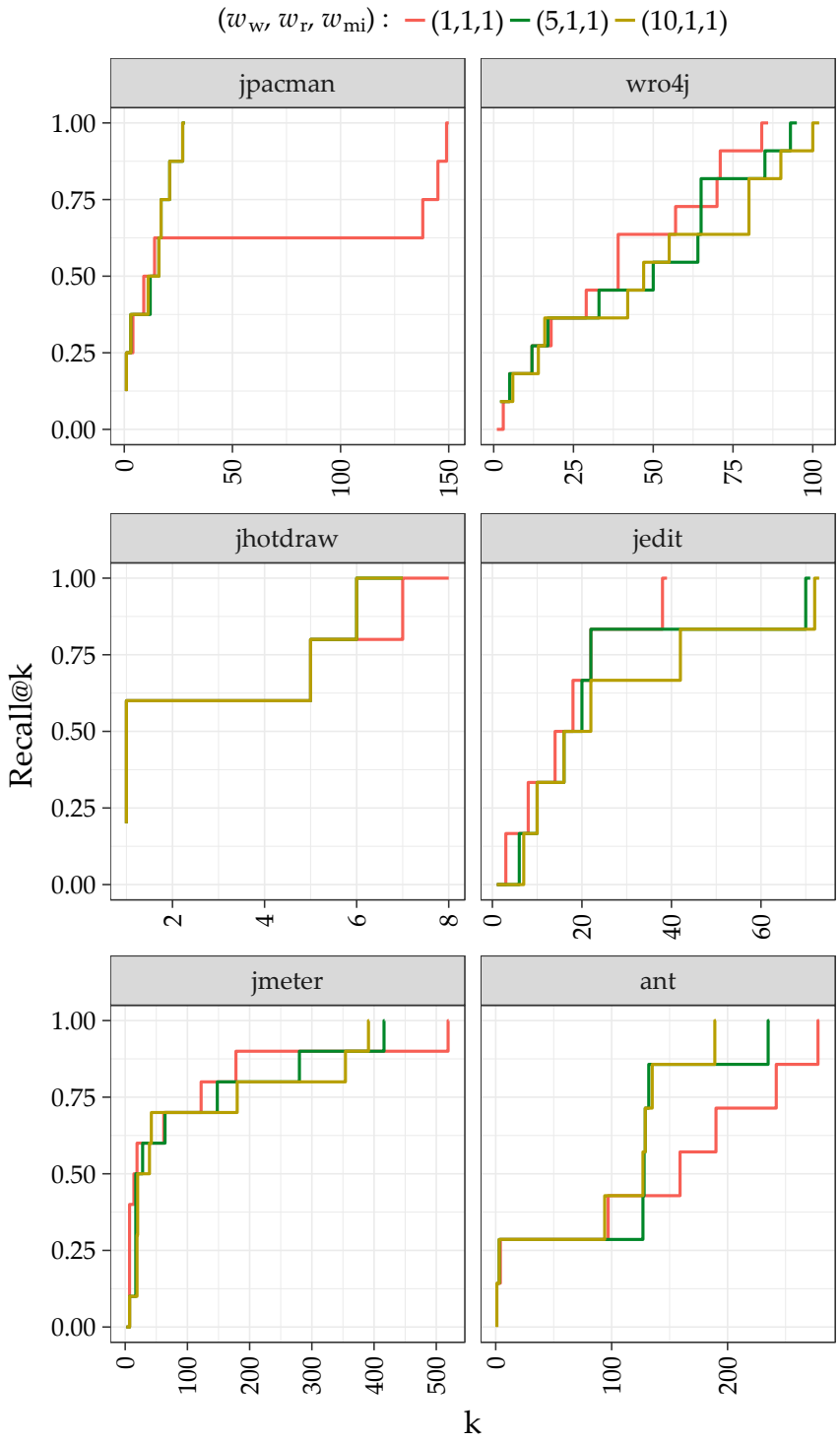

Figure 5.4: Effect of varying the value of w_w.

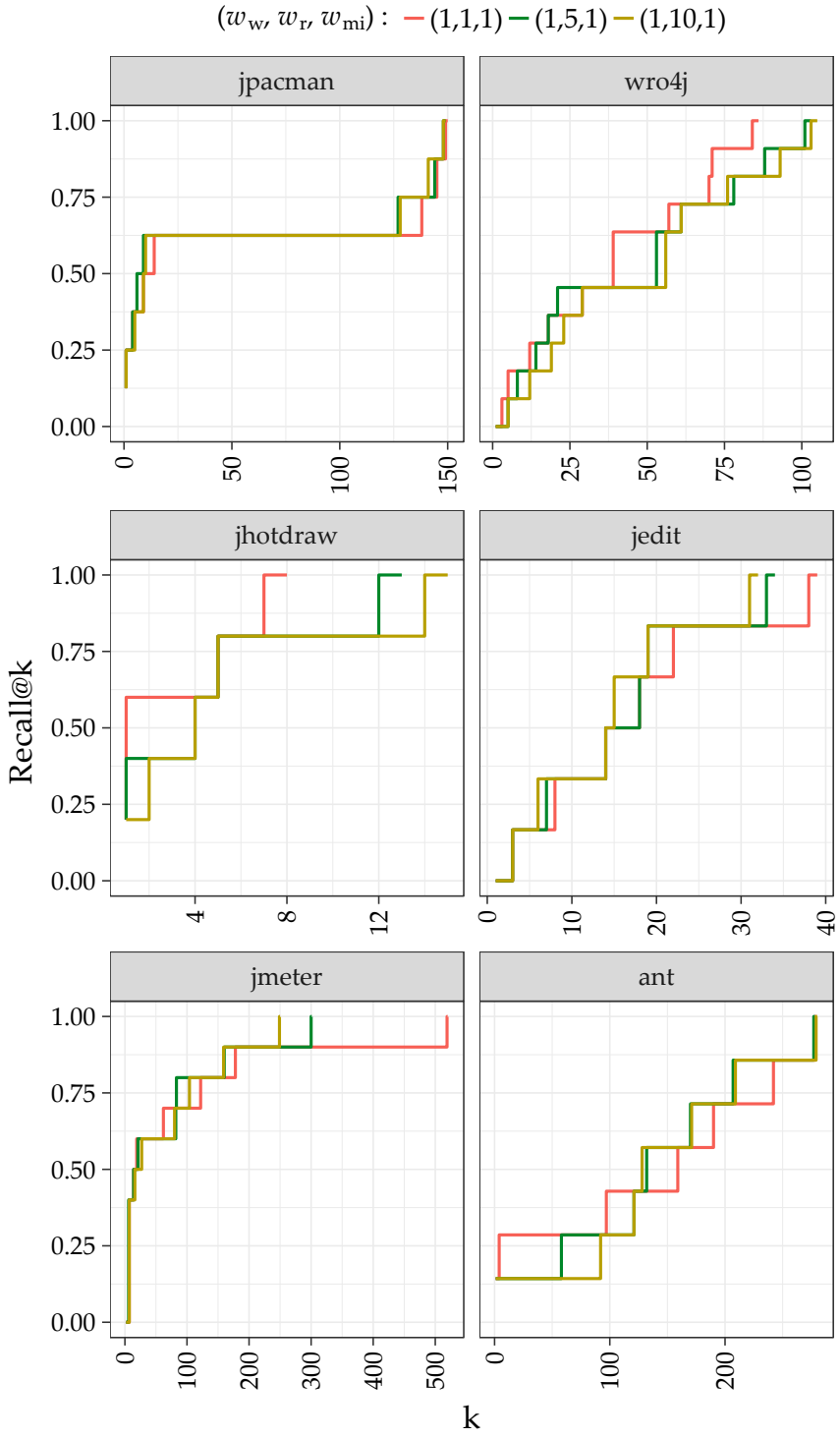

Figure 5.5: Effect of varying the value of w_r.

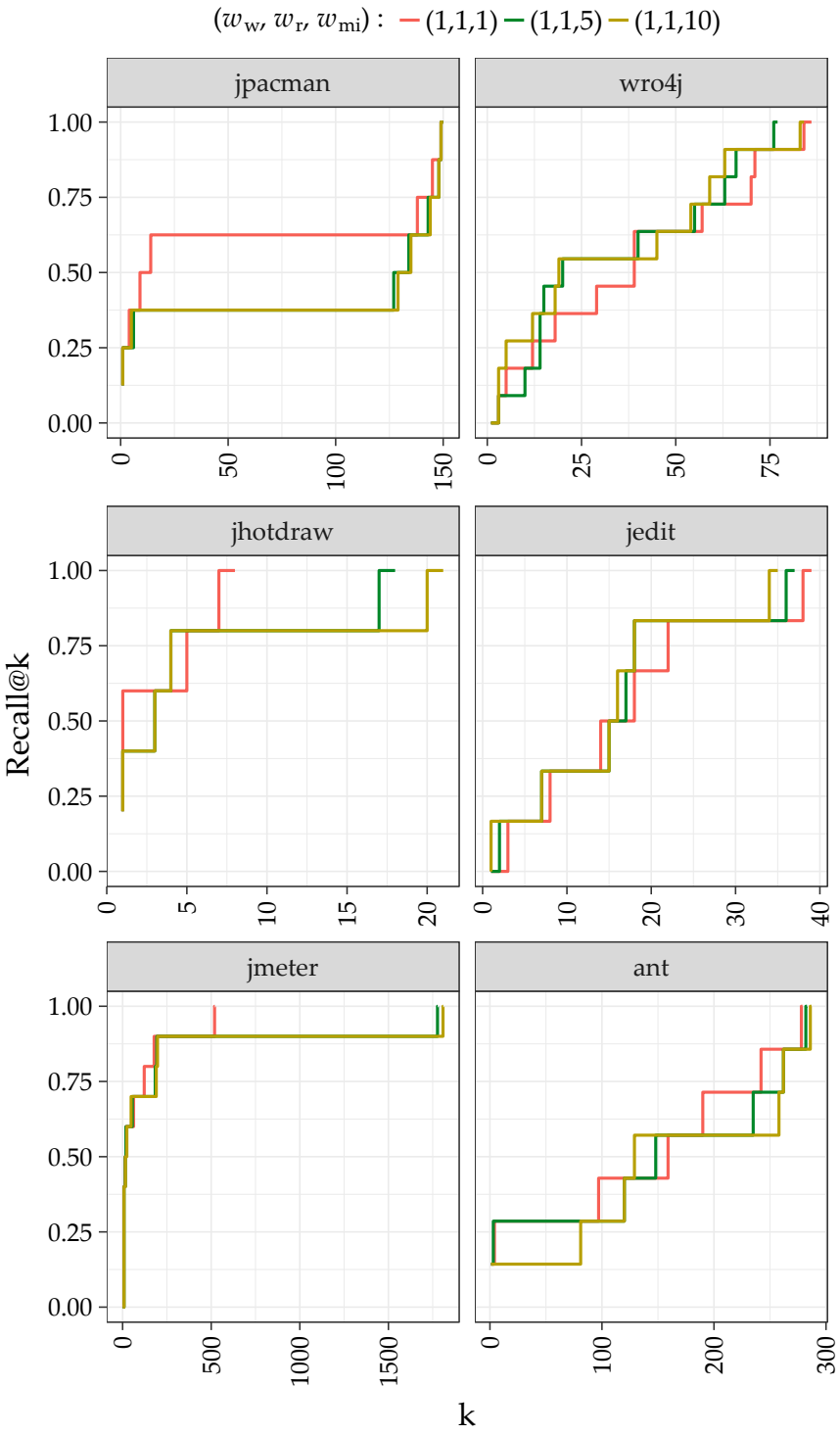

Figure 5.6: Effect of varying the value of w_{mi}.

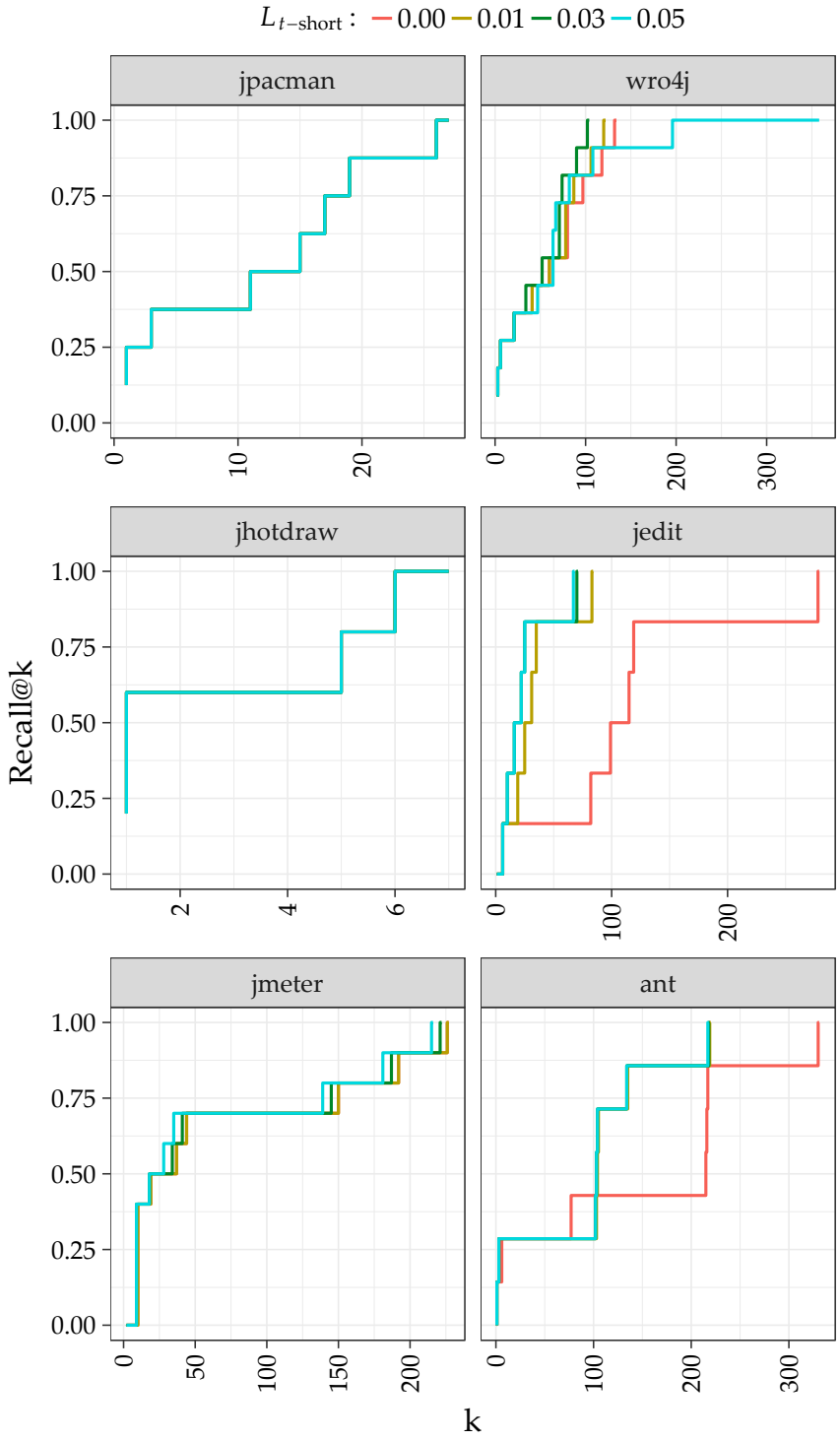

Figure 5.7: Effect of varying the value of $L_{t\text{-short}}$.

Table 5.6: Execution time.

Project	Base [s]	With logging [s]	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

Enlarging the weight for write access leads the improvement of the core identification 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 core identification 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.

5.4.3.4 Answer to RQ_4

Table 5.6 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: four background threads were used for writing a trace data onto a disk; the trace data was recorded 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 [156] imposed 100 %–800 % overhead in most cases, and the record and replay system [157] involved 480 % overhead on average.

Developers need to execute an instrumented application only once for core object identification (and summarized sequence diagram recovery). 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.

5.5 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 described in Section 5.6 to divide the long trace.

The ground truth of the core objects for *jpacman* were defined by the author. Although we have conducted some studies by using *jpacman* over two years and have sound 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 core identification 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 core identification results (and resulting summarized sequence diagrams); because we focus on the performance of core object identification, an evaluation of time savings is out of scope of this research. User studies need conducting in future studies to evaluate and validate the usefulness of identified core objects (and resulting summarized sequence diagrams) in actual maintenance tasks.

5.6 Related Work

5.6.1 Identifying Important Design Elements of a System

There exist several techniques for identifying important classes of a software system. Most of the techniques exploit network analysis and machine learning [41].

Thung 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) [77,78]. 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, some studies proposed techniques for identifying the most important key classes (i.e., core parts) of a system by using network analysis [40–42, 89–92] (we described the details of existing studies in Section 2.3.1). Note that the number of the key classes, which are important to comprehend the design overview, is around 5–10 [40,41]; this number is much lower than the number of classes contained in a forward designed class diagram.

Those techniques that utilize network analysis for identifying key classes 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 those existing techniques 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 identifies key *concrete classes/objects* that are instantiated at 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 existing techniques with our technique. We need to use an appropriate one according to the purpose of tasks developers undertake.

There is another study that utilized historical information to measure importance of classes [93]. Their technique identifies classes, which are frequently affected by design changes in a change history, as more important classes. Combining historical information with static and dynamic dependencies/properties would improve the quality of key class identification. However, there are some drawbacks to such an approach: historical information is not always available; a change history tends to be noisy, which makes analysis more complex and difficult.

5.6.2 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" for characterizing temporary data structures and program regions that create and use them [147].

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 [158]. The object flow graph was used for some purposes, such as architectural risk analysis [159] and impact analysis [160].

5.7 Summary

Identifying core (important) objects of key classes, which implement the most important concepts of a system, and comprehending their behavior are crucial in an early stage of program comprehension.

In this chapter, we presented a technique for identifying 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.

We evaluated our technique on traces of various open-source software systems. The results showed that our technique had superior core identification performance, compared with the state-of-the-art trace summarization technique. Our technique identified all the core objects within top 7–222 (108 on average) important objects, whereas that of the state-of-the-art technique ranged 148–4,378 (998 on average). The runtime overhead imposed by our technique was 167.6 % on average, which is relatively small compared with recent scalable dynamic analysis techniques; it shows the practicality of our technique.

Overall, our technique can effectively identify core objects with a small runtime overhead; thus, it can be a valuable tool in an early stage of program comprehension.

6

Abstracting Object Interactions at a Concept Level for Recovery of a Summarized Sequence Diagram

In this chapter, we propose an object grouping technique for horizontally summarizing a reverse-engineered sequence diagram. Our technique constructs object groups based on Pree's meta patterns, in which each object group corresponds to a concept in the domain of a subject system. Visualizing interactions only among important object groups, we generate a summarized reverse-engineered sequence diagram that depicts a behavioral overview of the subject system.

We evaluated the feasibility and effectiveness of our technique with traces generated from various types of open source software. The results of our experiment showed that our technique outperformed the state-of-the-art trace summarization technique in terms of both the quality of object grouping and the horizontal reduction of reverse-engineered sequence diagrams.

Overall, our technique can significantly reduce the horizontal size of a reverse-engineered sequence diagram while incurring only a small runtime overhead. The resulting summarized sequence diagram depicting a behavioral overview of a system is expected to be a valuable tool for developers in an early stage of program comprehension.

This chapter is based on our paper "Execution trace abstraction based on meta patterns usage" [49] published at the *Working Conference on Reverse Engineering (WCRE)*,

6.1 Introduction

In this chapter, we propose object grouping and visualization techniques for reducing the horizontal size of reverse-engineered sequence diagrams, with the goal of helping developers comprehend a behavioral overview of a system. Our technique constructs object groups based on Pree's meta patterns [161, 162] that are the most primitive design patterns; in our technique, each group corresponds to a concept in the domain of a subject system. In conjunction with the core object identification technique described in Chapter 5, we identify important object groups and visualize intergroup interactions only among those important groups; this generates a summarized version of a reverse-engineered sequence diagram. The summarized diagram depicts a behavioral overview of the subject system and is expected to be a valuable tool for developers in an early stage of program comprehension.

To improve maintainability in object-oriented programming, a concept is often divided into several classes by using design patterns [94, 162]. For instance, in a game application, the concept of *player* is divided into several state classes such as *normal state* and *death state* by using the GoF (Gang of Four) state pattern. From the perspective of maintainability, implementing a concept with several classes is beneficial; however, from the perspective of program comprehension, it produces numerous design elements to understand, which can confuse developers and increase the amount of effort required for program comprehension

tasks. Our technique constructs object groups based on Pree's meta patterns, in which each group corresponds to a concept. By reunifying divided concepts as object groups, we generate a summarized sequence diagram that depicts a behavioral overview of a system at the concept level.

We evaluated the feasibility and effectiveness of our technique with traces generated from various types of open source software. The results showed that our technique outperformed the state-of-the-art trace summarization technique in terms of horizontal summarization of reverse-engineered sequence diagrams. With regard to the quality of object grouping, under the condition of #lifelines (i.e., the horizontal size of a sequence diagram) being less than 30, the F-score and *Recall* of our technique were 0.670 and 0.793 on average, respectively, whereas those of the state-of-the-art technique were 0.421 and 0.670, respectively. Our technique imposed a runtime overhead of 129.2 % on average, which is relatively small compared with recent scalable dynamic analysis techniques.

The main contributions of this chapter are as follows:

- We propose an algorithm for constructing object groups based on Pree's meta patterns, where each object group corresponds to a concept.
- We present an algorithm for drawing a summarized sequence diagram that depicts a behavioral overview as intergroup interactions among important object groups.
- The feasibility and effectiveness of our technique are evaluated using traces generated from various types of open source software.

The remainder of this chapter is organized as follows. In Section 6.2, we briefly provide the background knowledge required for describing our technique. Section 6.3 elaborates our trace summarization technique. We evaluate our technique in Section 6.4 and discuss the threats to validity in Section 6.5. Section 6.6 describes key related work and Section 6.7 summarizes this chapter.

6.2 Background

6.2.1 Pree's Meta Patterns

Pree's meta patterns [161, 162] provide a classification of template-hook structures that are commonly used in object-oriented programming. Figure 6.1 shows the classification consisting of seven types of template-hook structural patterns. Pree's meta patterns are the most primitive design patterns: most concrete design patterns (e.g., the GoF design patterns [94]) are realized by using some of

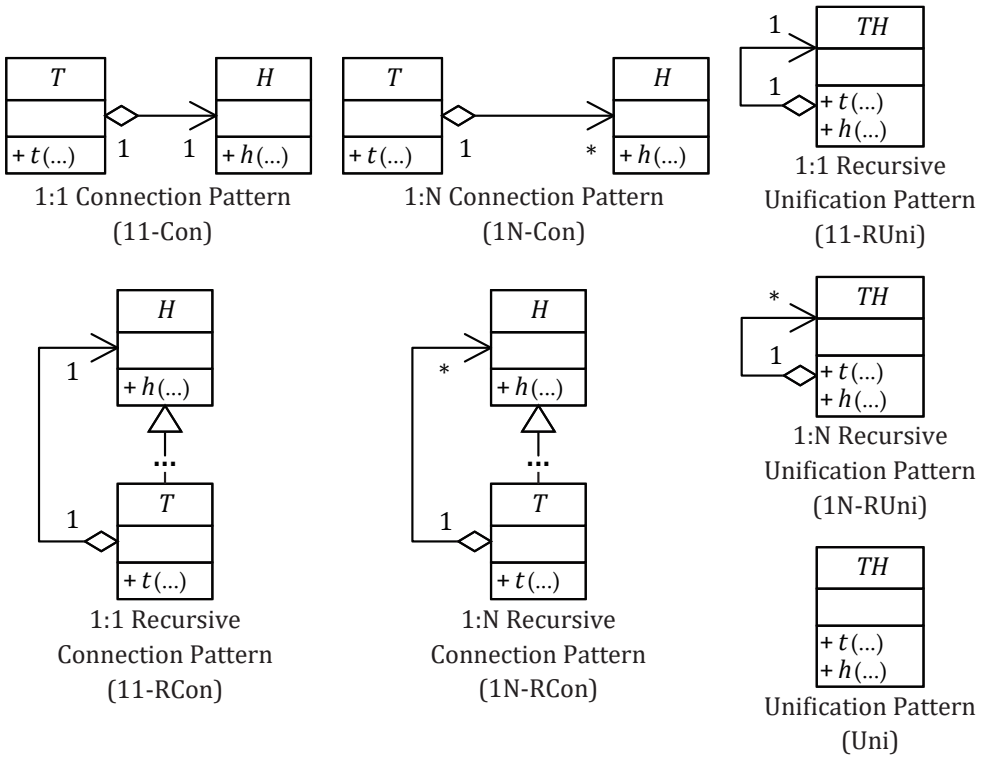

Figure 6.1: Pree's meta patterns. *T*, *H*, *t*(), and *h*() denote a template class, a hook class, a template method, and a hook method, respectively. *TH* denotes that a template class and a hook class are unified into one class. The name in the parentheses below each pattern is the abbreviated name used in this dissertation.

the meta patterns. For example, the GoF decorator pattern is an instance of the 11-RCon pattern.

In our technique, by identifying template-hook objects involved in the same meta pattern, we construct object groups that each correspond to a concept.

6.2.2 Core Object Identification

We presented a core object identification technique (COIT) in Chapter 5. By analyzing reference relations and dynamic properties, the COIT identifies core objects that are important in comprehending the design overview of a system. The core identification process consists of the following two steps: (1) eliminating temporary objects and (2) estimating the importance of each object.

In step (1), the COIT analyzes reference relations and lifetimes for each object to identify temporary objects. By analyzing reference relations among objects in a manner similar to compilers' escape analysis, the COIT identifies the dynamic

scope for each object and assigns an escape state to each object. Escape states the COIT assigns are categorized into three types: GlobalEscape, ReferenceEscape, and Captured. GlobalEscape (resp. ReferenceEscape) denotes an object that is referenced from another static (resp. non-static) object. An object is marked as Captured if the object is not referenced from any other objects.

Based on the escape state and lifetime for each object, the COIT identifies an object o_i as a temporary object if it satisfies the following condition.

$$(\mathit{EscapeState}(o_i) = \text{“Captured”} \wedge \mathit{Lifetime}(o_i) < \mathit{Lifetime}_{\max}(\boldsymbol{O}) \cdot L_{t\text{-long}})$$
$$\vee\ (\mathit{EscapeState}(o_i) = \text{“ReferenceEscape”} \wedge \mathit{Lifetime}(o_i) < \mathit{Lifetime}_{\max}(\boldsymbol{O}) \cdot L_{t\text{-short}})$$

Here, $\boldsymbol{O}$ is a set of all the objects. $\mathit{Lifetime}_{\max}$ returns the maximum lifetime over all the objects. $L_{t\text{-long}}$ and $L_{t\text{-short}}$ are threshold factors for deciding whether an object is long-lived, short-lived, or neither. The above condition means an object o_i is identified as a temporary object if o_i satisfies one of the following conditions: o_i is referenced from no other objects and is not long-lived; o_i is referenced only from other non-static objects and is short-lived.

In step (2), the COIT estimates the importance of each non-temporary object based on the access frequency. Important objects are expected to be heavily accessed from other objects. The COIT calculates the importance value of each object as the harmonic mean of the write, read, and method-invocation frequencies. By building an importance-based object ranking $R = \langle o_1, o_2, ..., o_n \rangle$, the COIT identifies an object o_i as a core object if the importance of o_i is greater than the threshold I_t.

6.3 Constructing Object Groups Corresponding to Concepts

In this section, we describe the details of our technique for recovering a summarized version of a reverse-engineered sequence diagram. Our technique consists of the following steps:

1. Meta pattern detection (Section 6.3.1).

2. Construction of object groups based on meta patterns (Section 6.3.2).

3. Visualization of intergroup interactions among important object groups (Section 6.3.3).

6.3.1 Meta Pattern Detection

Because Pree's meta patterns employ a structural classification, we can identify the meta patterns via static analysis. Algorithm 6.1 shows our meta pattern detection algorithm. First, we detect hook methods (l.1). Then, we identify meta patterns by detecting template methods that invoke the hook methods (l.2).

For the hook method detection, visiting all the method declaration nodes in an abstract syntax tree (AST), we check whether each method overrides/is-overridden-by another method. If a method overrides/is-overridden-by another method, the method is identified as a hook method (ll.3–16). Note that we do not treat all constructors and all methods declared in the topmost type (e.g., Object#equals(...) and Object#toString(...)) as hook methods.

For the meta pattern detection, traversing all the method invocation nodes in an AST, we test whether each method invokes the detected hook methods. If a method invokes the hook methods, the caller method is treated as a template method, and a meta pattern consisting of the template method and the invoked hook methods is detected (ll.17–27).

We determine the type of each meta pattern by analyzing the relationships between the template and hook classes (ll.28–35). Note that the multiplicity of the relationship between the template and hook classes is determined on the basis of the fields of the template class. If a template class refers to a hook class via a field whose type is not a subtype of java.util.(Iterable|Map), we determine the multiplicity as 1 (i.e., 1:1 meta patterns); otherwise, the multiplicity is treated as N (i.e., 1:N meta patterns). Although our algorithm for determining multiplicity generates some false positives, the final result of object grouping is not affected because our grouping algorithm does not require the multiplicity information (described in Section 6.3.2).

6.3.2 Object Grouping based on Meta Patterns

6.3.2.1 Overview

We construct object groups by using the meta pattern information detected in Section 6.3.1.

As mentioned in Section 6.1, in object-oriented programming, a concept is often divided into several classes by using design patterns to improve maintainability. Thus, we consider a set of template and hook objects involved in the same meta pattern corresponds to a concept. Our grouping algorithm identifies template/hook objects for each detected meta pattern and groups those objects.

In a template-hook structure, the template method defines the outline of a

Algorithm 6.1 Meta pattern detection.

Input: a set of source code $S = \{s_1, s_2, ..., s_n\}$.
Output: a set of meta-patterns detected $P = \{p_1, p_2, ..., p_m\}$.

```
1:  H ← DETECTHOOKS(S)
2:  return DETECTMETAPATTERNS(S, H)

3:  function DETECTHOOKS(S)
    ▷ In: a set of source code S.
    ▷ Out: a set of sets HS s.t. H ∈ HS is a set of hook methods;
           m_h ∈ H overrides or is-overridden-by other methods in H.
4:      HS ← ∅
5:      for each s ∈ S do
6:          a ← the AST of s
7:          for each method declaration node n in a do
8:              m ← the method declared in n
9:              if (∃H ∈ HS)[m ∈ H] then
10:                 continue
11:             M_super ← all the methods overridden by m
12:             M_sub ← all the methods that override m
13:             if M_super ≠ ∅ ∨ M_sub ≠ ∅ then
14:                 H ← {m} ∪ M_super ∪ M_sub
15:                 HS ← HS ∪ {H}
16:     return HS
```

(Continued on the next page.)

process and the hook methods correspond to the details of the process. Out technique groups template/hook objects such that the behavior of the template method is retained and the behavior of the hook methods is omitted after grouping; thereby, the resulting summarized sequence diagram depicts a behavioral outline of a system.

We categorize the seven types of Pree's meta patterns into the following three pattern types, in terms of behavioral aspects.

- Recursive patterns (consisting of 11-RUni, 1N-RUni, 11-RCon, and 1N-RCon).
- Connection patterns (consisting of 11-Con and 1N-Con).
- Unification pattern (consisting of Uni).

Algorithm 6.1 Meta pattern detection (continued).

(Continued from the previous page.)

```
17: function detectMetaPatterns(S, HS)
    ▷ In: a set of source code S; a set of sets of hook methods HS.
    ▷ Out: a set of meta patterns P, where p ∈ P is a triple ⟨m_t, H, t⟩,
           m_t is a template method, H is a set of hook methods,
           and t is a meta-pattern type.
18:   P ← ∅
19:   for each s ∈ S do
20:     a ← the AST of s
21:     for each method invocation node n in a do
22:       m_i ← the method invoked in n
23:       m_e ← the method enclosing the expression of n
24:       if (∃H ∈ HS)[m_i ∈ H] then
25:         t ← detectPatternType(m_e, m_i)
26:         P ← P ∪ {⟨m_e, H, t⟩}
27:   return P

28: function detectPatternType(m_t, m_h)
    ▷ In: a template method m_t; a hook method m_h.
    ▷ Out: a meta-pattern type t = ⟨c, m⟩, where c is a category name, and m is
    multiplicity.
29:   t_t ← the type declaring m_t (i.e., the template type)
30:   t_h ← the type declaring m_h (i.e., the hook type)
31:   mul ← the multiplicity of the reference from t_t to t_h
32:   if m_h is invoked with the keyword 'this.' or 'super.' then return
    ⟨"Unification", mul⟩
33:   if t_t is equal to t_h then return ⟨"Recursive Unification", mul⟩
34:   if t_t is one of the subtypes of t_h then return ⟨"Recursive Connection", mul⟩
35:   return ⟨"Connection", mul⟩
```

For each pattern type, we describe our grouping approach in the following sections.

6.3.2.2 Object Grouping for the Recursive Patterns

In a meta pattern of the recursive patterns, a template object recursively refers to template/hook objects; that is, a reference chain consisting of template/hook

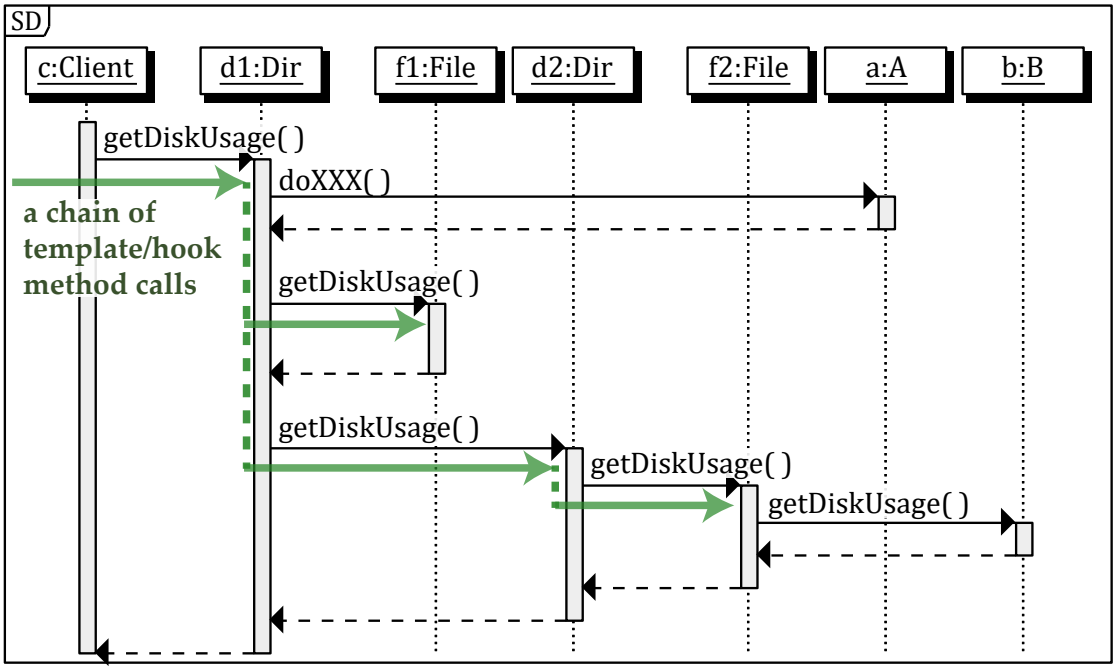

Figure 6.2: A chain of template and hook method calls. The Dir (resp. File) class is a template (resp. hook) class. Classes A and B are neither template classes nor hook classes.

objects is constructed. Typically, the template and hook methods have the same method signature. Once the template method is invoked, template/hook method calls are propagated in the reference chain; as a result, a chain of template/hook method calls is constructed.

For example, suppose a concept of *file system* that consists of two elements: *directory* and *file*. Typically, a file system is realized using the GoF composite pattern (i.e., 1N-RCon pattern). There is a FileBase class that declares some abstract methods for file operations. A Dir (resp. File) class that extends the FileBase class as a template (resp. hook) class corresponds to *directory* (resp. *file*).

Suppose a method FileBase#getDiskUsage() that returns disk usage information. The getDiskUsage() method is overridden in the Directory and File classes. Once the Dir#getDiskUsage() (a template method) is invoked, template/hook methods (i.e., Dir#getDiskUsage and File#getDiskUsage()) are recursively invoked; as a result, a chain of template/hook method calls is constructed as shown in Figure 6.2.

If the template and hook methods have the same method signature, we group all template/hook objects that can be reached by traversing a chain of template/hook method calls. If the signature of the template method is different from that of the hook methods, we group template/hook objects (except for the template object that receives the first message in a chain of template/hook

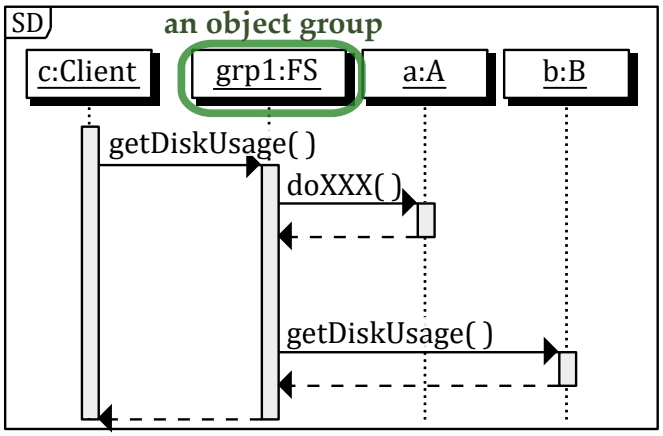

Figure 6.3: Resulting summarized sequence diagram that depicts an outline of the behavior shown in Figure 6.2. The lifeline named grp1:FS corresponds to the concept of *file system*. The lifeline named b:B is not gathered into grp1:FS because class B is neither a template class nor a hook class.

method calls) such that the first template and hook method calls are retained after grouping. (Recall that intra-group interactions are omitted in a summarized sequence diagram.)

In the case of *file system* shown in Figure 6.2, all the template/hook objects (i.e., Dir and File instances) are gathered into one group. Figure 6.3 shows the resulting summarized sequence diagram that depicts a behavioral outline of Figure 6.2. In the summarized diagram, only the first call of getDiskUsage() is retained and successive template/hook method calls are omitted. Note that although B#getDiskUsage() has the same signature as the template/hook methods, the object of class B is not added into the group because the class B is neither a template class nor a hook class.

6.3.2.3 Object Grouping for the Connection Patterns

In a meta pattern of the connection patterns, a template object refers to hook objects. Typically, the method signature of the template method is different from those of the hook methods. Once the template method is invoked, the template object calls the hook method for each hook object.

For example, suppose a GUI application built using Model-View-Controller (MVC) architecture. Once a property of a model object is changed, view objects receive notifications of the property change and update their presentations. This notification and update mechanisms are typically realized using the GoF observer pattern (i.e., 1N-Con pattern). A property change in a model is notified by a template method such as Model#notifyPropertyChanged(...); subsequently,

for each hook object, a hook method such as View#onPropertyChanged(...) is invoked in the template method, which updates the presentations.

The hook objects involved in a connection pattern, whose hook methods are invoked in the template method of the same template object, are gathered into one group. For example, a GUI application using MVC architecture mentioned above, view objects (i.e., hook objects) that are notified from a model object (i.e., template object) are grouped. The resulting summarized sequence diagram depicts interactions between the model object and the group of the view objects.

6.3.2.4 Object Grouping for the Unification Pattern

In the unification pattern, the template and hook methods belong to the same class; that is, a template object also behaves as a hook object. We do not (cannot) perform any grouping for the unification pattern. Note that each non-grouped object will be regarded as a group consisting of a single object during visualization (described in Section 6.3.3).

6.3.2.5 Overall Algorithm of Object Grouping

We show the overall algorithm of our object grouping in Algorithm 6.2. By Algorithm 6.2, object groups are constructed as mentioned in the previous sections.

We assume that an execution trace is represented in the form of an event sequence based on the behavior model (B-model) described in Chapter 4. The B-model represents the behavior of an object-oriented system. The B-model consists of event elements such as *EntryEvent* / *ExitEvent* events, which represent "entry into a constructor/method" / "exit from a constructor/method," 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 $\langle e_1, e_2, ..., e_n \rangle$, where e_i is an event element in the B-model.

In Algorithm 6.2, once a template method is invoked, either detectObjGrpForRecursivePattern or detectObjGrpForConnectionPattern is called according to the pattern type, which constructs an object group (ll.1–9).

For the recursive patterns, if the template and hook methods have the same name, the template object is inserted into the resulting group (ll.21–23). The function calleeObj(e) returns the callee object of the specified entry event e. Then, callee objects in a chain of template/hook method calls are added into the resulting group; that is, if all the invoked methods in the call stack *cs* are the template/hook methods, all the callee objects are added into the resulting group (ll.24–35). Note that some code fragments in the template/hook methods might be extracted as private-methods whose names are different from those

Algorithm 6.2 Object grouping based on meta patterns.

Input: an execution trace $ET = \langle e_1, e_2, ..., e_l \rangle$;
meta patterns detected $P = \{p_1, p_2, ..., p_m\}$.
Output: a set of object groups $OGS = \{OG_1, OG_2, ..., OG_n\}$.

1: $T \leftarrow \emptyset$
2: **for each** *EntryEvent* $e_i \in ET$ **do**
3: **for each** $p \in P$ **do**
4: **if** the template method of p is invoked at e_i **then**
5: **if** p belongs to *recursive patterns* **then**
6: $t \leftarrow$ detectObjGrpForRecursivePattern(e_i, p)
7: **if** p belongs to *connection patterns* **then**
8: $t \leftarrow$ detectObjGrpForConnectionPattern(e_i, p)
9: $T \leftarrow T \cup t$
10: $OGS \leftarrow \emptyset$
11: $O_{\text{all}} \leftarrow$ all the objects generated during the execution
12: **for each** $o_i \in O_{\text{all}}$ **do**
13: **for each** $p \in P$ **do**
14: $OGS \leftarrow OGS \cup \{\{o \mid o \in OG \wedge \langle o_i, p, OG \rangle \in T\}\}$
15: $OGS \leftarrow \{OG \mid OG \in OGS \wedge (\nexists OG_i \in OGS)[OG \subset OG_i \wedge \exists o_j \exists o_k \exists p[\langle o_j, p, OG \rangle \in T \wedge \langle o_k, p, OG_i \rangle \in T]]\}$
16: **return** OGS

17: **function** detectObjGrpForRecursivePattern(e_{entry}, p)
▷ **In:** an *EntryEvent* e_{entry}; a meta pattern $p = \langle m_t, H, t \rangle$.
▷ **Out:** a triple $\langle o_t, p, OG \rangle$, where o_t is a template object, p is a meta pattern, and OG is an object group.
18: **if** e_{entry} is a self-call **then**
19: $e_{\text{entry}} \leftarrow$ the latest entry event e_l s.t. e_l is not a self-call and occurs before e_{entry}
20: $OG \leftarrow \emptyset$
21: $m_{at} \leftarrow$ the method invoked at e_{entry}
22: **if** m_{at} and $m_h \in H$ have the same name **then**
23: $OG \leftarrow OG \cup \{$calleeObj$(e_{\text{entry}})\}$
24: $e_{\text{exit}} \leftarrow$ the *ExitEvent* that corresponds to e_{entry}

(Continued on the next page.)

Algorithm 6.2 Object grouping based on meta patterns (continued).

(Continued from the previous page.)

```
25:   i_entry, i_exit ← the indices of e_entry, e_exit in ET, resp.
26:   cs ← a new call stack
27:   for i ← i_entry + 1 to i_exit do
28:     e_i ← the ith event in ET
29:     if e_i is an EntryEvent then
30:       cs.push(e_i)
31:       M ← {m | m is the method invoked at e_k ∈ cs ∧ e_k is not a self-call}
32:       if (∀m ∈ M) [(∃m_h ∈ H)
            [m has the same name and declaring class as m_h]] then
33:         OG ← OG ∪ {calleeObj(e_i)}
34:     if e_i is an ExitEvent then
35:       cs.pop(e_i)
36:   return ⟨calleeObj(o_entry) , p , OG⟩

37: function detectObjGrpForConnectionPattern(e_entry, p)
    ▷ The types of the inputs/output are the same
      as detectObjGrpForRecursivePattern.
38:   e_exit ← the ExitEvent that corresponds to e_entry
39:   O_callee ← {o | o is an object that receives a message m
                    ∧ m has the same name and declaring class as m_h ∈ H
                    ∧ m comes from calleeObj(e_entry) in trace(e_entry, e_exit)}
40:   return ⟨calleeObj(o_entry) , p , O_callee⟩
```

of the template/hook methods. To handle such a case, by ignoring self-calls and focusing on inter-object interactions, we detect a chain of template/hook method calls (ll.18, 19, and 31).

For the connection patterns, we group hook objects whose hook methods are invoked in the template methods of the same template object (ll.38–39). The function $\textsc{trace}(e_{\text{start}}, e_{\text{end}})$ returns a partial trace from e_{start} to e_{end} in the entire execution trace ET.

Our algorithm ignores the types of the return values and the parameters when testing the equality of methods (ll.22, 32, and 39); that is, we equate overloaded methods.

After constructing object groups for each meta pattern, we unify object groups having the same template object and meta pattern (ll.10–14); e.g., in

the GoF state pattern (i.e., 11-Con pattern), several state objects (i.e., the hook objects) that receive messages from the same template object are gathered into one group. Finally, if an object group is a subset of another group associated with the same meta pattern, we eliminate the subgroup (l.15).

It is worth noting that the template/hook objects involved in a meta pattern p might also be involved in another meta pattern p'. Thus, Algorithm 6.2 constructs soft clusters of objects; an object is allowed to belong to multiple object groups.

Note that because Algorithm 6.2 is pseudocode, the algorithm description is slightly simplified. When implementing the algorithm, thread information should be considered; that is, a call stack *cs* should be created per thread, and object grouping should be performed independently for each thread.

6.3.2.6 Object Grouping for Delegation Patterns

In object-oriented programming, delegation can be used as an alternative to inheritance; as a result, some delegate methods might appear in a chain of template/hook method calls. Grouping delegate objects that are the callee of those delegate methods together with template/hook objects might improve the quality of object grouping.

For example, in the case of the *file system* mentioned in Section 6.3.2.2, the File class might delegate the disk usage calculation to another class under certain conditions. In Figure 6.2, suppose that class B is a delegate of the File class and B#getDiskUsage() is called as a delegate method in File#getDiskUsage() under certain conditions. Because the object of the class B is also an element in the concept of *file system*, the object should be added to the object group named grp1.

Modifying Algorithm 6.2 as follows, we can group delegate objects together with template/hook objects.

- Relaxing the condition of line 32 as follows: $(\forall m \in M)\ [(\exists m_h \in H)[m$ has the same name as $m_h]]$ (i.e., the equality check regarding the declaring class is deleted; if a method is invoked by another method having the same name, we consider the invoked method as a delegate method).

- Grouping objects involved in a connection pattern in the same manner used for grouping objects involved in a recursive pattern. The only exception is that the template object is never added to the resulting group (i.e., ll.22–23 are skipped).

The grouping method that allows (resp. disallows) delegate methods to appear in a chain of template/hook method calls is referred to as "MP+D" (resp.

"MP"). We investigate how grouping delegate objects affects performance in Section 6.4 (RQ_2). Our technique groups delegate objects by default because "MP+D" outperforms "MP" as described in Section 6.4.3.2.

6.3.3 Visualizing intergroup Interactions among Important Object Groups

Combined with the core identification technique (COIT) mentioned in Section 6.2.2, we identify important object groups. By visualizing intergroup interactions only among the important object groups, we obtain a summarized version of a reverse-engineered sequence diagram that depicts a behavioral overview of a system.

Algorithm 6.3 shows our algorithm for drawing a summarized sequence diagram. Given an importance-based object ranking created by the COIT, if an object group contains an important object o_{imp} whose importance is greater than the threshold I_t, we treat the object group as an important one that should appear as a lifeline in the resulting summarized sequence diagram (ll.1–8). If no groups contain an important object o_{imp}, we create a new object group whose only member is the object o_{imp} (ll.6–7).

There are two types of methods for drawing a sequence diagram: class-level and instance-level. In a class-level sequence diagram, lifelines having the same types are unified into one lifeline and thereby the horizontal size of the diagram is reduced. Meanwhile, an instance-level sequence diagram provides a detailed behavioral view for each object, which increases the horizontal size of the diagram. A class-level diagram is useful in an early stage of program comprehension. As the understanding of a subject system deepens, an instance-level sequence diagram would become more suitable.

In Algorithm 6.3, object groups are converted from instance-level into class-level at lines 9–15. If an instance-level diagram is needed, it needs only to skip lines 9–15.

The function drawIntergroupInteractions draws intergroup interactions among the given groups. Each group is visualized as a lifeline in the resulting diagram. For each object group, a *group-id* and a *group-type-name* are displayed in the box on the top of the lifeline where the *group-id* is a unique identifier for the group and the *group-type-name* is the type name of an arbitrary object in the object group.

Because our object grouping constructs soft clusters, an object can belong to multiple groups. Suppose an object o belongs to multiple groups $G_1, G_2, ..., G_n$. If an entry event e whose callee object is o occurs, we must determine which of the groups should receive the message corresponding to e. If the message is

Algorithm 6.3 Drawing a summarized sequence diagram.

Input: an object ranking $R = \langle o_1, o_2, ..., o_m \rangle$,
where the importance of o_i is greater than that of o_{i+1};
a threshold of the importance I_t;
all the object groups $OGS = \{OG_1, OG_2, ..., OG_n\}$.

1: $OGS_{\text{imp}} \leftarrow \emptyset$
2: **for** $i \leftarrow 1$ **to** m **do**
3: **if** IMPORTANCE$(o_i) < I_t$ **then**
4: **break**
5: $S \leftarrow \{OG \mid OG \in OGS \wedge o_i \in OG\}$
6: **if** $S = \emptyset$ **then**
7: $S \leftarrow \{\{o_i\}\}$
8: $OGS_{\text{imp}} \leftarrow OGS_{\text{imp}} \cup S$
9: $TGS \leftarrow \emptyset$
10: **for each** object group $OG \in OGS_{\text{imp}}$ **do**
11: $TGS \leftarrow TGS \cup \{$TYPENAMESET$(OG)\}$
12: $OGS_{\text{class}} \leftarrow \emptyset$
13: **for each** $TG \in TGS$ **do**
14: $OG_{\text{class}} \leftarrow \{o \mid o \in OG \wedge OG \in OGS_{\text{imp}} \wedge$ TYPENAMESET$(OG) = TG\}$
15: $OGS_{\text{class}} \leftarrow OGS_{\text{class}} \cup \{OG_{\text{class}}\}$
16: DRAWINTERGROUPINTERACTIONS(OGS_{class})

17: **function** TYPENAMESET(OG)
▷ **In:** an object group OG.
▷ **Out:** a set of type names.
18: **return** $\{t \mid t$ is the type name of $o \in G\}$

either the template or hook method of a meta pattern p, and G_i is constructed in regard to p, we send the message to G_i; otherwise, we send the message to an arbitrary group that contains the object o.

6.4 Experiment

6.4.1 Research Questions and Evaluation Approaches

We address the following research questions through the experiment.

RQ$_1$: How effective is our technique in terms of reducing the horizontal size of reverse-engineered sequence diagrams?

Motivation:

We aim to investigate the performance of our technique with respect to reducing the horizontal size of reverse-engineered sequence diagrams, which is the primary objective of this research. If a sequence diagram is small and contains object groups that are important in comprehending a design overview, the diagram is useful for program comprehension.

Evaluation Approach:

For each subject system, we extract the ground truth of important concepts from execution scenarios, documents, and tutorials (Section 6.4.2.2). Utilizing the ground truth, we evaluate the quality of the resulting object groups by using the evaluation measures described in Section 6.4.2.3. The better the quality measures are, the more useful the resulting summarized diagram will be for program comprehension. We evaluate the effectiveness of our technique by investigating the trade-off between the values of the quality measures and the horizontal size (i.e., #lifelines) of the resulting diagram.

As a baseline, we selected the recent trace summarization technique that constructs object groups based on lifetimes and reference relations [48]. Given an importance-based object ranking R and a threshold of the importance I_t, the baseline technique constructs hardly-clustered object groups as follows. For each object o_{imp} whose importance is greater than I_t, the baseline technique gathers objects $o \in O$ where each object $o \in O$ is (in)directly referenced from o_{imp} and the lifetime of o is smaller than that of o_{imp}. Thus, for each important object o_{imp}, the baseline technique constructs an object group such that o_{imp} and the gathered objects constitute a composition relation.

The baseline technique generates an instance-level sequence diagram, while our technique generates a class-level sequence diagram. For comparison, we convert the object groups constructed by the baseline technique into class-level groups by applying ll.9–15 in Algorithm 6.3.

RQ$_2$: How much is performance improved by grouping delegate objects together with template/hook objects?

Motivation:

As mentioned in Section 6.3.2.6, delegate methods might appear in a chain of template/hook method calls. In this research question, we investigate how

grouping delegate objects together with template/hook objects affects performance.

Evaluation Approach:

We compare the performance of "MP+D" and "MP" (see Section 6.3.2.6). As RQ_1, we investigate the trade-off between the values of the quality measures and the #lifelines.

RQ_3: Does our object grouping based on meta patterns have a wide range of application?

Motivation:

Our technique constructs object groups based on meta patterns; therefore, the number of opportunities for object grouping depends on the number of template/hook method pairs in a subject system. If meta patterns are rarely used in the subject system, the number of object groups constructed will be small; this might lessen the effectiveness of our technique. For each subject system, we investigate to what extent our grouping technique can be applied.

Evaluation Approach:

We investigate the number of meta patterns in each subject system and what percentage of generated objects are grouped by our technique.

RQ_4: How much is the runtime overhead imposed by our technique?

Motivation:

Our technique weaves logging codes into a subject system for analyzing runtime information, which causes a runtime overhead. In terms of practicality, it is highly important to ensure that the runtime overhead will be small. In this research question, we investigate the overhead imposed by our technique.

Evaluation Approach:

For each subject system, we measure the execution times of the woven and original programs to calculate the runtime overhead. We evaluate the overhead through a comparison against the overhead incurred by recent scalable dynamic analysis techniques.

Table 6.1: Subject systems and execution scenarios. In the "KLOC" column, lines of test code are not counted.

Project	Ver.	KLOC	Execution scenario
jpacman [133]	SVN r53	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 to the power state; quit the application.
JModeller [163]	SVN r1015	45.5	launch the application; add two class figures side by side; add attributes and methods; edit the names of the class, attributes, and methods; add connectors indicating association, dependency, and inheritance between the classes; quit the application.
wro4j [150]	1.7.7	34.0	execute *wro4j-runner* while specifying test resources (*.js and *.css files) as target files, 'cssMin' as a pre-processor, and 'jsMin' as a post-processor.
JMeter [153]	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.

6.4.2 Experimental Setup

6.4.2.1 Subject Systems and Execution Scenarios

The subject systems and execution scenarios are shown in Table 6.1. For each subject, we selected a representative execution scenario.

6.4.2.2 Ground Truths

For each subject system, we define the ground truth for object grouping. We extract the important concepts in the domain of each subject system from scenario descriptions and documents. Then, we examine the source code and identify the types (classes) that implement the extracted concepts. Table 6.2 shows the ground truth for each subject system.

jpacman is a Pac-Man game application written in Java. We define the main entities in the Pac-Man game as the important concepts; these include *player* (Pac-Man), *ghost*, and *map*.

JModeller is a modeling application for drawing class diagrams; it is built on top of JHotDraw, which is known as a well-designed GUI framework. We extract the important concepts from an introductory article [164] that describes the design details of the JModeller application; e.g., *class*, which is a primary element in a class diagram, and *connection*, which defines a relationship between two classes.

wro4j is an application used for improving the loading time of a web application. For example, wro4j provides the functionalities of merging and minifying web resources (e.g., *.js and *.css files). Documentation [165] written by developers provides an overview of the design of wro4j. We consider components appearing in the architecture diagram in the document as the important concepts; e.g., *model*, which represents web resources to process, and *pre/post-processors*, which process web resources.

JMeter is an application that provides functionality for measuring the performance of a web application. The user manual [166] describes the components of JMeter. We extract major concepts relating to our execution scenario from the user manual. In addition, an overview of a test-execution flow is provided in the wiki [167]. We also extract important concepts from the description; e.g., *test configuration*, which specifies the method of measuring the performance, and *sampler*, which sends requests to target servers.

6.4.2.3 Evaluation Measures

One commonly used external quality measure for evaluating cluster quality is the F-measure. We also use the F-measure to evaluate the quality of object grouping. We calculate the F-measure in the same manner as Steinbach et al. [168, 169].

Let $RS = \{R_1, R_2, ..., R_n\}$ be the ground truth (the reference set) of grouping, where $R_i (1 \leq i \leq n)$ is a set of type names defined in Table 6.2. Let $OGS_{\text{class}} = \{OG_1, OG_2, ..., OG_m\}$ be the resulting groups of our technique or the baseline technique, where $OG_j (1 \leq j \leq m)$ is a set of objects (OGS_{class} is obtained at l.15 in Algorithm 6.3). Let $TS = \{T_1, T_2, ..., T_m\}$ be the resulting groups of type names, where $T_i = \{t \mid t \text{ is the type name of } o \in OG_j \wedge OG_j \in OGS_{\text{class}}\}$.

First, for the reference group R_i and the resulting group T_j, the *Recall* and *Precision* are defined as follows:

$$Recall(i, j) = \frac{|R_i \cap T_j|}{|R_i|},$$

$$Precision(i, j) = \frac{|R_i \cap T_j|}{|T_j|}.$$

Table 6.2: Important concepts and types corresponding to the concepts (i.e., ground truths). In the "Types corresponding to the concept" column, the binary names of the types are shown in the column. The package names of the types are omitted for simplicity. If a type is used as an anonymous class in the code, the name of the base class to extend is shown in the following brackets.

Project	Concept	Types corresponding to the concept
jpacman	Player	*Player, StateManager, Player$NormalState, Player$PowerState*
	Ghost	*Ghost, StateManager, Ghost$NormalState, Ghost$EatableState, RunningGhostBrain, RedGhostBrain*
	Gem	*Gem, EatGem*
	Map	*Map*
JModeller	Class	*ClassFigure, SeparatorFigure, GraphicalCompositeFigure, jmodeller.ClassFigure$(1 | 4 | 5) [TextFigure], RectangleFigure, JModellerClass*
	Layout	*StandardLayouter*
	Connection	*AssociationLineConnection, DependencyLineConnection, InheritanceLineConnection, JModellerClass*
	Tool	*DelegationSelectionTool, TextTool, JModellerApplication$1 [CreationTool], ConnectionTool, UndoableTool*
	Tool palette	*ToolButton*

(Continued on the next page.)

Then, the F score for the reference group R_i and the resulting group T_j is defined as follows:

$$F(i,j) = \frac{2 \cdot Recall(i,j) \cdot Precision(i,j)}{Recall(i,j) + Precision(i,j)}.$$

Finally, the F score for the reference set RS and the grouping result TS is calcu-

Table 6.2: Important concepts and types corresponding to the concepts (i.e., ground truths) (continued.).

Project	Concept	Types corresponding to the concept
(Continued from the previous page.)		
wro4j	Model	*WroModel, Group, Resource*
	WroManager	*WroManager*
	PreProcessor	*PreProcessorExecutor$2, [DefaultProcessorDecorator], BenchmarkProcessorDecorator, ExceptionHandlingProcessorDecorator, SupportAwareProcessorDecorator, MinimizeAwareProcessorDecorator, ImportAwareProcessorDecorator, CssMinProcessor, CSSMin*
	PostProcessor	*GroupsProcessor$1, [DefaultProcessorDecorator], BenchmarkProcessorDecorator, ExceptionHandlingProcessorDecorator, SupportAwareProcessorDecorator, MinimizeAwareProcessorDecorator, ImportAwareProcessorDecorator, JSMinProcessor, JSMin*
	Locator	*StandaloneServletContextUriLocator*
JMeter	Test configuration	*TestPlan, ThreadGroup, HTTPSamplerProxy, ResultCollector, Arguments, LoopController*
	Sampler	*HTTPSamplerProxy, HTTPHC4Impl*
	Listener	*ResultCollector, Summariser, Summariser$Totals, SummariserRunningSample*
	Test executer	*StandardJMeterEngine, PreCompiler, JMeterThread*

lated as follows:

$$n = \sum_{R_i \in RS} |R_i| \,,$$

$$F = \sum_{R_i \in RS} \frac{|R_i|}{n} \max_{T_j \in TS} F(i, j).$$

The $\max_{T_j \in TS} F(i, j)$ is the maximum value of $F(i, j)$ over all the elements in TS

and R_i. The F is the weighted average of $\max_{T_j \in TS} F(i, j)$ over all the elements in RS.

We investigate the trade-off between the F score and the horizontal size of the resulting sequence diagram (i.e., #lifelines). Note that, by definition, the F score easily reaches 1 if we create the power set of the set of all the objects; therefore, using only the F score is inappropriate for evaluating performance, and we must investigate the trade-off between the F score and #lifelines.

The more the objects of the important types shown in Table 6.2 are contained in the resulting sequence diagram, the more useful the resulting diagram will be for program comprehension. Thus, we also evaluate the rate of the important types contained in the resulting diagram (i.e., the recall). The *Recall* for the reference set RS and the grouping result TS is calculated in the same manner as F:

$$Recall = \sum_{R_i \in RS} \frac{|R_i|}{n} \max_{T_j \in TS} Recall(i, j) .$$

6.4.2.4 Weaving Extent

Our technique requires the results of meta pattern detection (Section 6.3.1) and an execution trace. The execution trace must contain information regarding method-entries/exits, objects, and field accesses. Note that the field access information is used for obtaining an importance-based object ranking R (see Algorithm 6.3). We use *SELogger*, which is a part of REMViewer [149], for recording execution traces.

We aim to help developers comprehend the behavior specific to the domain of a subject system. For this reason, we weave logging codes only into the classes defined in the subject system; that is, libraries are not instrumented. As the only exception, collection libraries are instrumented in order to analyze reference relations more precisely (details are provided in the paper describing the baseline technique [48]). This weaving condition is realistic in terms of avoiding a heavy logging overhead.

6.4.3 Results

The recorded runtime information is shown in Table 6.3. The numbers of messages and objects affect the size of a reverse-engineered sequence diagram. The number of lifelines (i.e., the horizontal size) of a class-level (resp. instance-level) reverse-engineered sequence diagram is equal to the number of loaded classes (resp. objects).

Table 6.3: Recorded runtime information.

Project	Total events	Messages	Loaded classes	Objects
jpacman	19,024,287	8,387,304	57	1,273
JModeller	1,074,279	568,940	153	6,993
wro4j	311,192	147,858	295	1,504
JMeter	630,969	333,190	372	5,562

6.4.3.1 Answer to RQ_1

Varying the threshold I_t (an input of Algorithm 6.3) causes the resulting object groups to change, which affects the values of the quality measures (i.e., the *F* score and *Recall*) and #lifelines. Figure 6.4 and Figure 6.5 show the trade-off relationship between the quality measures and #lifelines.

In our technique, the entire set of resulting object groups (i.e., the output *OGS* of Algorithm 6.2) is independent of the value of threshold I_t. As the value of I_t decreases, the number of groups displayed in the resulting diagram increases. Thus, as #lifelines is increased, the values of the quality measures monotonically increase.

On the other hand, in the baseline technique, the entire set of resulting object groups depends on the value of I_t. After the value of I_t is determined, the baseline technique constructs a set of object groups from scratch. Thus, the values of the quality measures do not monotonically increase along with the #lifelines. For instance, if an object *o*, which (in)directly refers to numerous objects (i.e., *o* is close to the root object in reference relations), ranks as one of the most important objects, many objects are gathered into the same group, which causes the *Recall* value to increase. Because the baseline technique constructs hard clusters of objects, reducing the value of I_t increases the number of groups (i.e., causes the large object group containing *o* to be decomposed); this might cause the *Recall* value to decrease, which also affects the *F* score (e.g., the JModeller case).

As shown in Figure 6.4 and Figure 6.5, in most of the subject systems, the values of our technique's quality measures increase significantly at the portion of the graph with the fewest #lifelines, and our technique outperforms the baseline technique at #lifelines > 25.

A sequence diagram with many #lifelines (e.g., > 100) is not suitable for practical use because it requires significant effort for developers to comprehend the content of the diagram. Thus, we focus on the area where #lifelines is less than 30 (i.e., small enough for manual investigation) in Figure 6.4 and Figure 6.5. The maximum performance under the condition of #lifelines < 30 is shown as

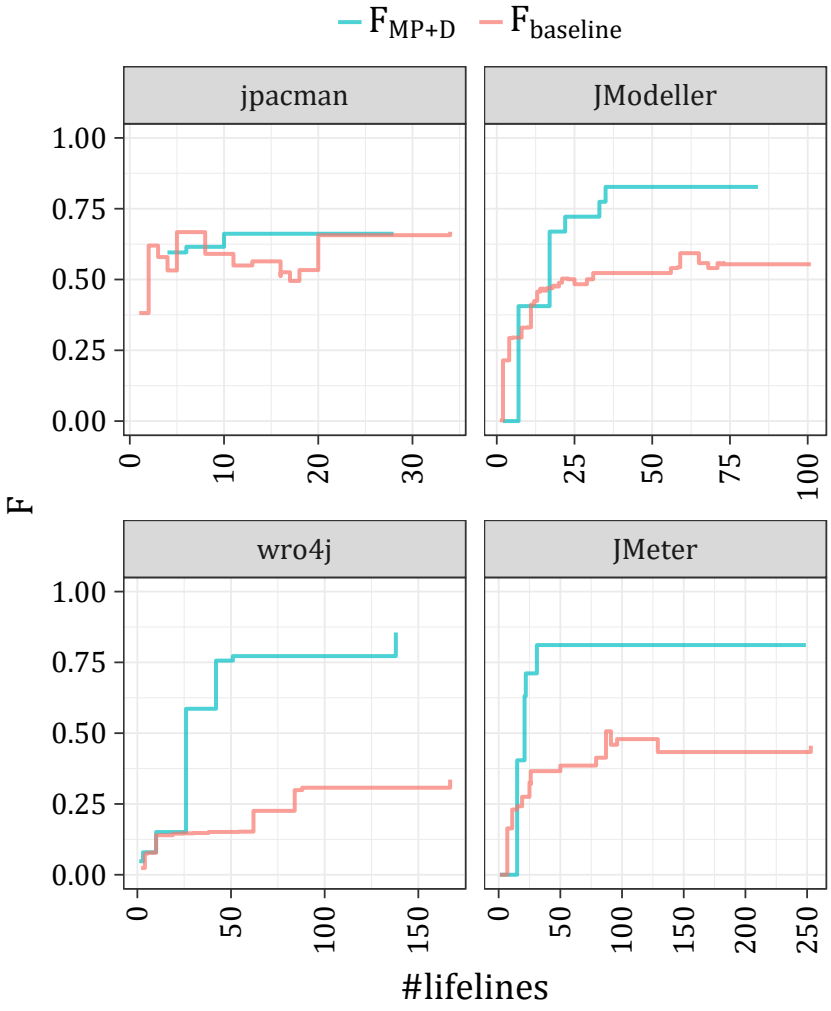

Figure 6.4: Performance of our technique and the baseline technique (F score vs. #lifelines).

Table 6.4. Table 6.4 shows that our technique (the "MP+D" column) outperforms the baseline technique for almost all cases. Our technique achieved an F score (resp. a *Recall*) of 0.670 (resp. 0.793) on average, which is 0.249 (resp. 0.123) higher than that of the baseline technique; thus, our technique is more effective compared with the baseline in terms of reducing the horizontal size of reverse-engineered sequence diagrams.

The cases where our grouping technique did not work well can be categorized into the following three types.

(1) Some important concepts are implemented using a combination of multiple meta patterns. Applying our grouping algorithm to objects corresponding

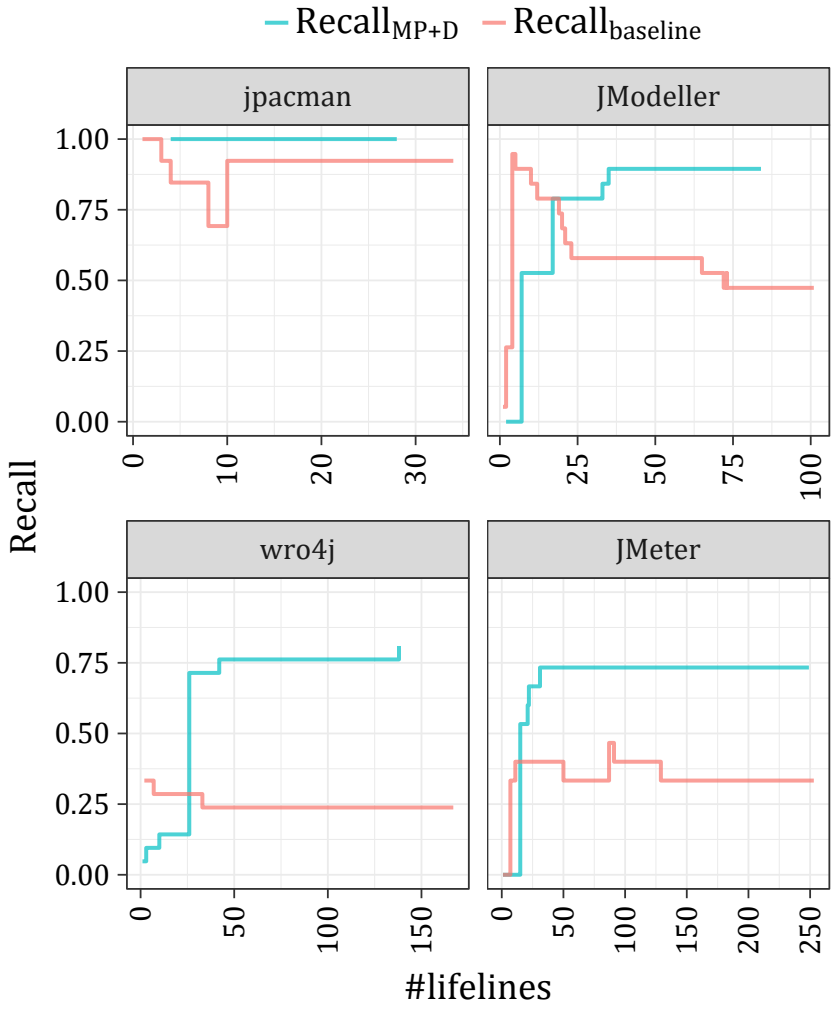

Figure 6.5: Performance of our technique and the baseline technique (*Recall* vs. #lifelines).

to such concepts results in a set of multiple small groups; this decreases the *Recall*.

(2) Some important concepts participate in the same meta pattern. Objects, which implement such concepts, are grouped into the same group. This leads to a negative effect on the *F*-score (i.e., decreases the precision).

(3) Some important concepts are not involved in any meta or delegation patterns; obviously, our grouping technique cannot handle such a case.

These cases are expected to occur infrequently because the results of object grouping are of high quality as shown in Table 6.4; thus, we consider that these cases does not become a large threat to our technique. Regarding the type (3), we further investigate the prevalence of meta and delegation patterns used in

Table 6.4: Maximum performance under the condition of #lifelines < 30. Each shaded cell indicates the maximum performance value in each row.

	MP+D		MP		baseline	
Project	*F*	*Recall*	*F*	*Recall*	*F*	*Recall*
jpacman	0.662	1.000	0.636	0.538	0.667	1.000
JModeller	0.722	0.789	0.655	0.632	0.503	0.947
wro4j	0.586	0.714	0.586	0.714	0.146	0.333
JMeter	0.711	0.667	0.767	0.667	0.366	0.400
Average	0.670	0.793	0.661	0.638	0.421	0.670

each project in Section 6.4.3.3 (RQ_3).

Our technique achieved an *F* score (resp. a *Recall*) of 0.670 (resp. 0.793) on average, under the condition of #lifelines < 30 (which is acceptable for manual investigation); this is 0.249 (resp. 0.123) higher than that of the baseline technique. Thus, our technique is more effective than the baseline technique in terms of reducing horizontal size of reverse-engineered sequence diagrams.

6.4.3.2 Answer to RQ_2

We show the performance of "MP+D" and "MP" (see Section 6.3.2.6) in Figure 6.6 and Figure 6.7. Figure 6.6 (resp. Figure 6.7) shows the trade-off relationship between the *F* score (resp. the *Recall*) and #lifelines. As shown in Figure 6.6 and Figure 6.7, allowing delegate methods in a chain of template/hook method calls (i.e., "MP+D") tends to improve the performance of our technique.

Table 6.4 shows the maximum performance when there are fewer than 30 #lifelines, which is a size acceptable for manual investigation. As shown in Table 6.4, the *F* score (resp. the *Recall*) of "MP+D" is 0.009 (resp. 0.155) higher than that of "MP" on average. For all the subject systems, allowing delegate methods in a chain of template/hook method calls increased the number of opportunities for object grouping. Moreover, grouping delegate objects together with template/hook objects improved performance in all cases (except for the *F* score in the JMeter case); i.e., the grouping of delegate objects rarely impaired the performance.

Grouping delegate objects together with template/hook objects improved performance in almost all cases. Under the condition of #lifelines < 30, the *F* score (resp. the *Recall*) of "MP+D" is 0.009 (resp. 0.155) higher than that of "MP" on

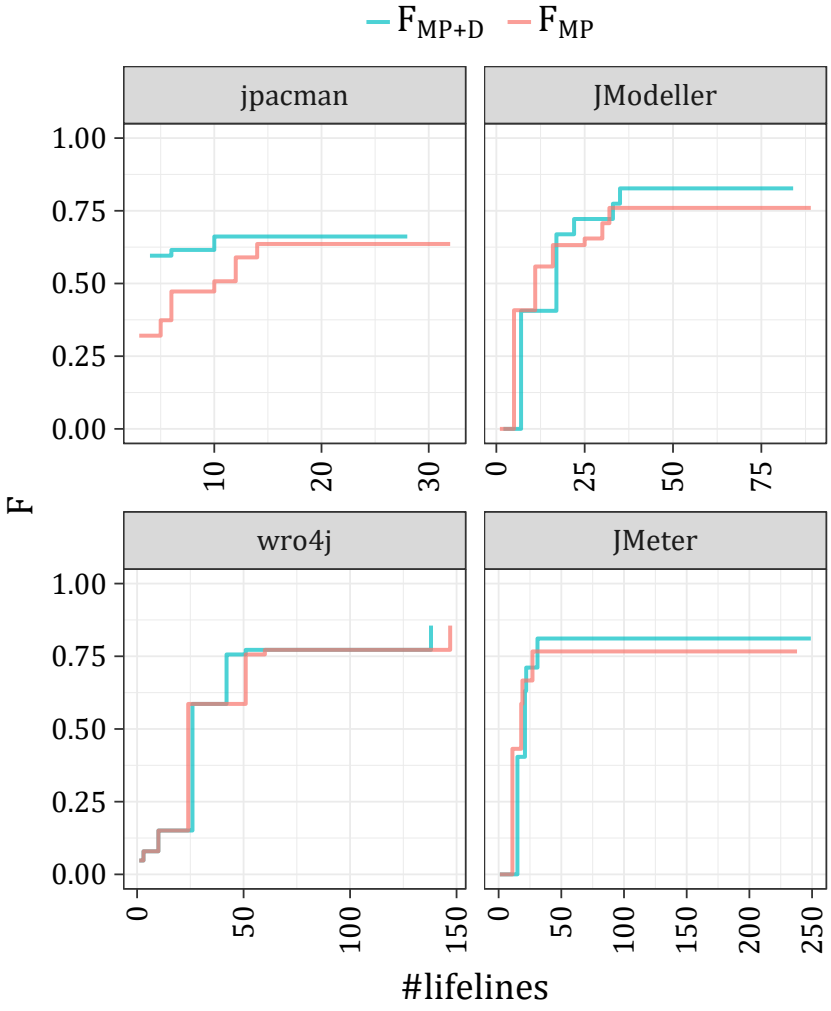

Figure 6.6: Effect of allowing delegate methods in a chain of template/hook method calls (*F* score vs. #lifelines).

average.

6.4.3.3 Answer to RQ_3

Table 6.5 and Table 6.6 show the numbers of meta patterns and delegations for each subject system. Table 6.5 shows the number of method invocations (i.e., #pairs of caller and callee methods) that are statically detected (e.g., in the jpacman case, there are 62 pairs of template and hook methods). Table 6.6 shows the number of pairs of template and hook methods for each pattern type.

As shown in Table 6.5, 33.2 % of all method invocations are hook/delegate

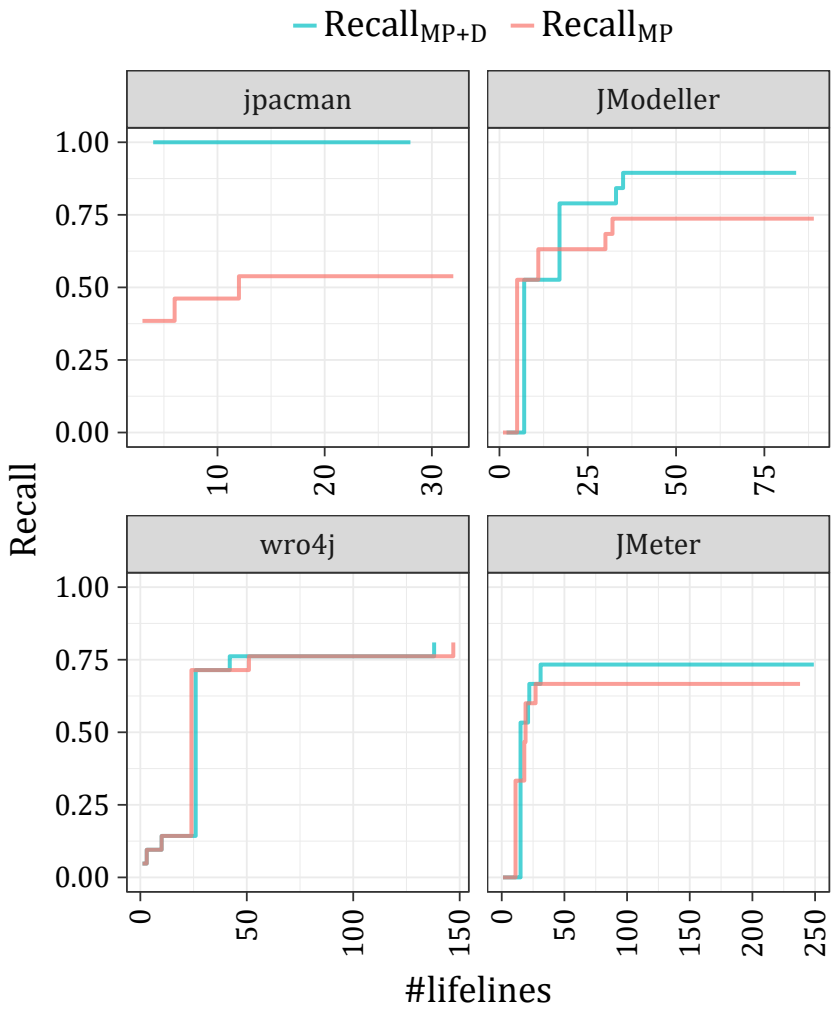

Figure 6.7: Effect of allowing delegate methods in a chain of template/hook method calls (*Recall* vs. #lifelines).

method invocations on average; there are a large number of meta patterns and delegations in the subject systems. Moreover, as shown in Table 6.6, the unification pattern and connection patterns tend to be used more frequently than the other patterns.

We show the numbers of grouped and non-grouped objects in Table 6.7. In Table 6.7, the number outside (resp. inside) the parentheses shows the number of non-temporary objects (resp. all the objects). During a program execution, numerous temporary objects are generated and those are not important for program comprehension. We identified temporary objects using the technique described in Section 5.2.1. Note that those temporary objects are not included

Table 6.5: Numbers of method invocations, meta-patterns, and delegations. The "All" column shows the total number of method invocations in each subject system. The "MP" column shows the number of meta-patterns (i.e., #pairs of template and invoked hook methods). The "Delegation" column shows the number of delegations (i.e., #pairs of delegating and delegated methods) in which the delegate method is not a hook method. Each number is statically counted. Static methods, methods declared in java.lang.Object, and library methods are excluded. The "Rate of MP+D" is calculated by ("MP" + "Delegation")/"All" · 100.

Project	All	MP	Delegation	Rate of MP+D [%]
jpacman	425	62	8	16.5
JModeller	7,627	3,649	100	49.2
wro4j	5,035	1,837	97	38.4
JMeter	24,477	6,560	480	28.8
Average	9,391	3,027	171	33.2

Table 6.6: Number of meta-patterns detected.

Project	Uni	11-RUni	1N-RUni	11-Con	1N-Con	11-RCon	1N-RCon
jpacman	10	0	0	4	42	0	6
JModeller	1,596	0	9	336	1,393	143	172
wro4j	309	0	1	727	664	48	88
JMeter	2,214	3	13	687	2,887	335	421

in an importance-based object ranking R, an input of Algorithm 6.3.

As shown in Table 6.7, on average, 50.2 % of the non-temporary objects are grouped by our technique (i.e., 50.2 % of the objects are either template, hook, or delegate objects).

As shown in Table 6.5 and Table 6.7, our technique uncovered many opportunities for object grouping. In addition, our technique achieved high F scores and *Recall*s for all the subject systems, regardless of the differences in the rates of grouped objects (see Table 6.4, Figure 6.4, and Figure 6.5); that is, objects corresponding to the important concepts tend to utilize template/hook structures or delegations.

Numerous meta patterns and delegations are used in the subject systems. On average, 33.2 % of all the method invocations were hook/delegate method invocations. 50.2 % of all the non-temporary objects were grouped by our

Table 6.7: Numbers of grouped and non-grouped objects. The "Grouped" column shows the number of objects grouped by "MP+D". The "Non-grouped" column shows the number of non-grouped objects. The "Rate of grouped" column is the rate of the grouped objects to all the objects. The number outside (resp. inside) the parentheses shows the number of non-temporary objects (resp. all the objects).

Project	Grouped	Non-grouped	Rate of grouped [%]
jpacman	147 (278)	409 (995)	26.4 (21.8)
JModeller	149 (4,625)	119 (2,368)	55.6 (66.1)
wro4j	107 (240)	277 (1,264)	27.9 (16.0)
JMeter	3,962 (4,543)	392 (1,019)	91.0 (81.7)
Average	1,091 (2,422)	299 (1,399)	50.2 (46.4)

technique. There were many grouping opportunities found by our technique; thus, we expect our technique to have a wide range of application. Moreover, our technique achieved high quality object grouping, regardless of the rates of meta patterns and delegations used; objects of important concepts tended to utilize meta patterns or delegations.

6.4.3.4 Answer to RQ_4

Table 6.8 shows the runtime overhead. For each execution scenario, we measured the execution time five times, both with and without the logging codes for recording an execution trace; we then calculated the average overhead. 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 for *SELogger* as follows: four background threads were used for writing trace data onto a disk; the trace data was recorded in an uncompressed format.

Our technique imposed a runtime overhead of 129.2 % on average. This overhead is relatively small compared with recent scalable dynamic analysis techniques [156, 157], which incur runtime overhead of approximately 100 % to 800 %.

Developers only need to execute an instrumented application once to produce the behavioral visualization. Thus, in many cases, the overhead of our technique is expected to be acceptable in a development phase rather than in a production phase.

Our technique imposed a runtime overhead of 129.2 % on average, which is relatively small compared with recent dynamic scalable analysis techniques.

Table 6.8: Runtime overhead. The "Base" (resp. "With logging") column shows the execution time without (resp. with) logging codes. The "Overhead" is calculated by ("With logging" − "Base")/"Base" · 100.

Project	Base [s]	With logging [s]	Overhead [%]
jpacman	9.84	13.76	39.9
JModeller	5.15	10.24	98.9
wro4j	4.59	6.16	34.0
JMeter	4.97	22.06	344.1
Average	6.14	16.54	129.2

In many cases, this overhead is expected to be acceptable in program comprehension tasks.

Finally, to facilitate reader understanding, we show an example covering a portion of the resulting (summarized) sequence diagram from the *JModeller* case in Figure 6.8. Figure 6.8 provides the knowledge about a behavioral overview of the subject system as follows.

When a user clicks on the canvas to add a new class figure, the tool objects (the lifeline named "10:DelegationSelectionTool") receive a mouse-down event (mouseDown(...)), and a message communicating the addition of a new figure (add(...)) is sent to the view objects (the lifeline named "17:StandardDrawingView"). Then, the view objects are set as containers of the newly added figure object (addToContainer(...)).

A paint-message (paintComponent(...)) is sent from library code. Then, the view objects draw the figures in the view (draw(...)).

When a user clicks on the canvas to add a connector between two classes, the tool objects receive a mouse-up event (mouseUp(...)). After testing whether it is possible to connect the two classes (canConnect(...)), the tool objects add a new connection (connectStart/End(...)).

In the resulting diagram, objects are abstracted at a concept level (for example, *figure* ("0:ClassFigure") or *tool* ("10:DelegationSelectionTool")), which is useful for comprehending the behavioral overview of the subject system. The resulting diagram is expected to be a valuable tool for developers in an early stage of program comprehension.

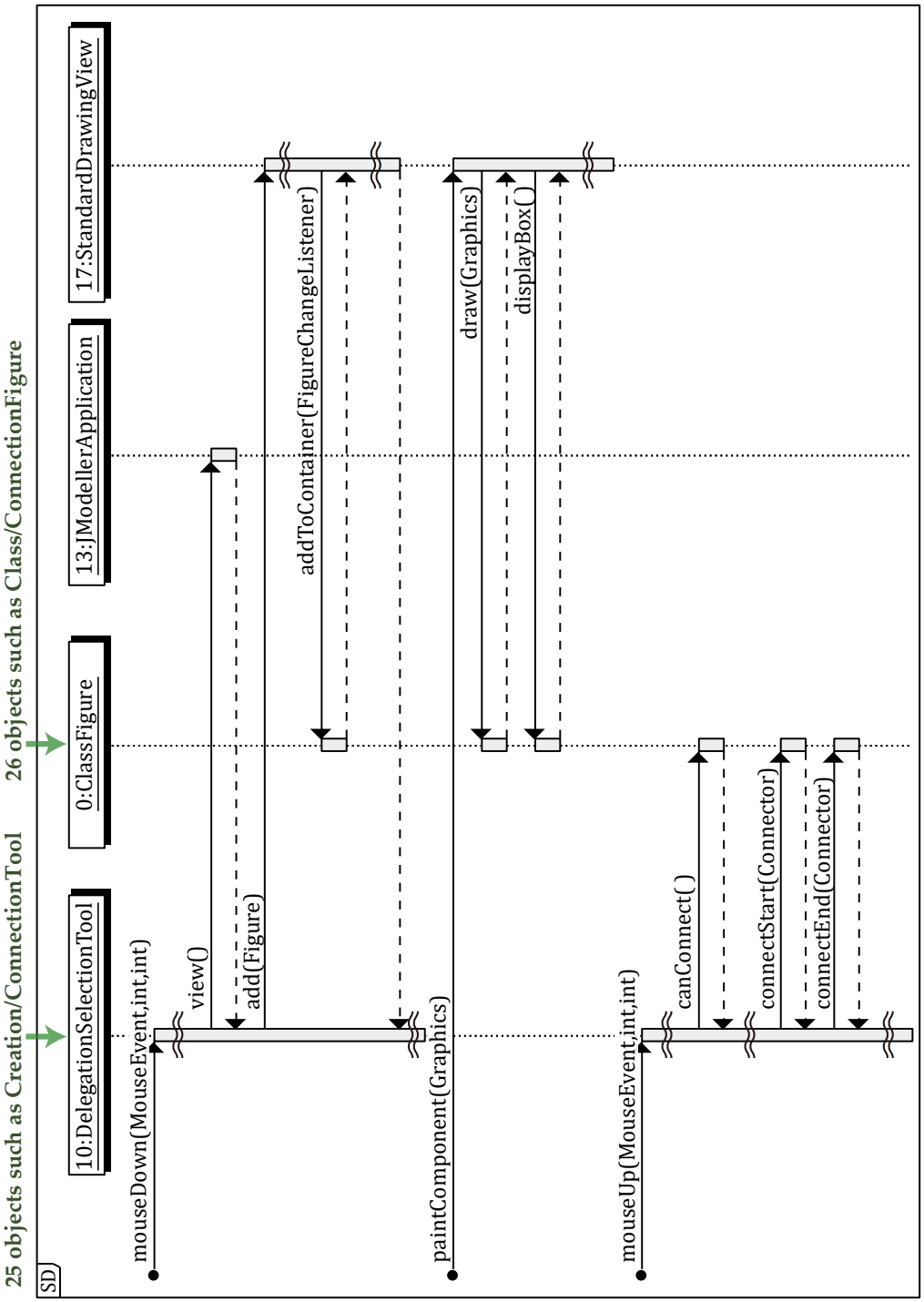

Figure 6.8: Portion of the resulting diagram from the *JModeller* case.

6.5 Threats to Validity

To improve the external validity, we used various types of applications in our experiment: a game application that periodically updates/renders the display according to the frame rate (jpacman); an modeling application written as an event-driven GUI program (JModeller); a command line application that includes file processing (wro4j); a multi-threaded command line application that includes network accesses (JMeter). However, because the number of subject systems is limited, it is unclear whether the results of our experiment can be generalized further.

We defined the ground truths by ourselves. To improve objectivity, we extracted the ground truths from the documents written by developers (except in the case of jpacman); however, the definitions of the ground truths might be incorrect. To mitigate the threat, we listed all the ground truths in this dissertation (Table 6.2) so that subsequent research can use and validate them.

The performance of trace summarization techniques was evaluated under the condition of #lifelines < 30 in our experiment; the condition might be subjective. We decided the condition based on the following reasons, and we consider that it is a valid condition. A sequence diagram whose #lifelines is less than 30 can fit within a few papers; the size can be acceptable for manual investigation. Several studies asked developers/analysts to utilize/access/validate their tools and recovered views, and the number of entities (e.g., classes, components, patterns, etc.) appearing in each of the views was around 15–60 [7, 112, 170]; the number of lifelines we restricted (#lifelines < 30) does not deviate from the range (15–60), and thus is considered to be acceptable for manual investigation.

We did not determine how much software maintenance task time is saved by our summarized sequence diagram; because we focus on the compactness of the diagram and the quality of the object grouping, an evaluation of time savings is out of the scope of this research. Further studies are needed in order to evaluate the usefulness of summarized sequence diagrams in actual maintenance tasks.

6.6 Related Work

6.6.1 Coping with the Scalability Issues of Execution Traces

Many studies have focused on recovering sequence diagrams from execution traces to facilitate program comprehension, testing, debugging, formal verification, etc. [7, 14, 171]. Those existing research has stated that because execution traces contain vast amounts of information, recovered diagrams are often afflicted by scalability issues.

6.6.1.1 Vertical Summarization of Reverse-engineered Sequence Diagrams

The vertical size of a reverse-engineered sequence diagram drastically increases in accordance with the increasing execution time (#messages). Many researchers have thus proposed message reduction techniques.

The primary approach for reducing the vertical size of a reverse-engineered sequence diagram is summarizing (abstracting) repetitive behavior in execution traces [17,18,21,22,24]. Traces contain substantial amounts of repetitive behavior owing to iterative statements or recursive method calls in program code; thus, summarizing the repetitive behavior results in a significant reduction in the vertical size of a reverse-engineered sequence diagram. To detect repetitive behavior in a trace, for example, previous studies utilized debug information [18] or proposed a regular expressions-based approach [24].

An execution trace is often composed of several phases (tasks); thus, phase detection approaches [19,23,25–29], which divide an entire trace into phases, are effective in reducing the vertical size of reverse-engineered sequence diagrams. To detect phases, for example, previous studies examined the creation timing of objects [19] or utilized text mining techniques [29].

Hamou-Lhadj and Lethbridge proposed another type of technique that removes unimportant methods (i.e., utilities) from a trace on the basis of the fan-in/out of each method [20]. By removing unimportant methods (implementation details), they achieved both vertical and horizontal summarization of a reverse-engineered sequence diagram.

6.6.1.2 Horizontal Summarization of Reverse-engineered Sequence Diagrams

Along with the vertical summarizations, horizontal summarizations of reverse-engineered sequence diagrams are an important factor in improving overall practicality. Some existing trace summarization approaches perform horizontal summarization. The technique proposed in this chapter is also categorized as this type.

A straightforward way to reduce the horizontal size of reverse-engineered sequence diagrams is to group objects per class/package or to filter out specified classes/packages. Several tools and techniques provide such grouping/filtering functionalities, such as JIVE [24] and Jinsight [80]. Compared with class/package-name-based grouping approaches, our technique recovers a much higher-level of abstractions: our technique outputs a concept-level abstraction of an execution scenario, and a concept is implemented by classes from several packages

as shown in Table 6.2. If developers try to comprehend behavioral overviews of a medium/larger-sized system that consists of hundreds/thousands of classes, our technique would be more valuable. As the understanding deepens, more finer-grained views (e.g., class-level or object-level views) would become more valuable. Filtering functionalities can be helpful for developers that are (partly) familiar with a subject system and already know class/package names of interest (e.g., in a debugging phase); however, in an early stage of program comprehension, developers do not always know which classes/packages play important roles in an execution scenario.

OGAN is a tool that visualizes interactions between objects related to two classes specified by a user [81]. For each object, OGAN computes the dynamic interaction context, which is a set of classes that use/are-used-by the object. Then, for each class, objects are clustered according to the equality of their dynamic interaction contexts. OGAN selects an object from a cluster for each specified class, and visualizes interactions between those and related objects. If developers already have a certain degree of knowledge about a subject system and know the class names of interest, OGAN is useful for obtaining representative behavior of the classes. Meanwhile, our technique is more suitable when developers do not know the class names that are important in comprehending the design overview of a subject system (e.g., in an early stage of program comprehension).

Dugerdil and Repond proposed a class clustering technique in which each cluster corresponds to a functional entity in a subject system [37]. Their technique divides an entire trace into segments; then, for each loaded class, it creates a feature vector that represents the binary occurrence (presence (1) or absence (0)) of the class in each segment. If the feature vectors of two classes are similar, those classes are gathered into the same cluster.

A trace, which is generated by exercising several features, consists of several functional phases (cf. phase detection approaches). Because the technique proposed by Dugerdil and Repond clusters classes from a functional perspective by segmenting a trace and calculating the similarities of the binary occurrence vectors, each of the constructed clusters tends to correspond to a functional phase. As a consequence, the resulting summarized sequence diagram provides a highly abstracted behavioral view that depicts entire relationships (interactions/flows) among functional entities; thus, their technique is suitable for comprehending a long/complex execution scenario that exercises several features. Compared with their technique, our technique aims at providing a finer-grained abstracted behavioral view. As its input, our technique assumes a simpler trace that exercises single or a few features, and visualizes only important behavior (interactions only among important object groups). Thus, our technique is suitable for comprehending how key objects behave in a simpler

execution scenario that exercises a few features of interest.

Toda et al. proposed an object-grouping technique based on the GoF design patterns [36]. The key idea of using design patterns for clustering is similar to our approach. The technique by Toda et al. focused on the GoF design patterns, which are more concrete than Pree's meta patterns; thus, although the risk of grouping irrelevant objects into the same cluster is low, the number of opportunities for object grouping is quite small. Our technique, which clusters objects based on more primitive design patterns, has a wider range of application.

In a similar approach recently proposed by Noda et al., objects are grouped based on lifetimes and reference relations [48] (this technique was utilized as a baseline technique in Section 6.4). For each important (core) object o_{imp}, the baseline technique identifies a set of objects that are in a composition relation together with o_{imp}, and gathers those objects into the same group. In Section 6.4, we investigated the performance of the baseline technique, and showed that the technique proposed in this chapter outperformed the baseline technique in terms of horizontal summarization of reverse-engineered sequence diagrams.

6.6.1.3 Other Approaches

There are many other types of approaches for handling the massive amounts of information in traces.

Alimadadi et al. identified recurring structures (motifs) in a trace and represented the trace as a hierarchical view [172]. Busany and Maoz proposed a technique that suggested a stopping criteria for a trace analysis (e.g., the sample size of a trace), such that the analysis results could be statistically trusted at a specified level of confidence [173].

Some existing studies presented effective visualization and exploration techniques [30–35]; e.g., a dedicated view for comprehending a large trace [31,32], an interactive tool that effectively explores a sequence diagram [30], and a generic toolkit that provides a set of functionalities for smoothly exploring a massive-scale sequence diagram [35].

Slicing techniques are likewise possible approaches to coping with the massiveness of reverse-engineered sequence diagrams [45,174].

6.6.2 Detecting and Leveraging Design Patterns

A number of design pattern detection techniques have been proposed [97,175] While most existing techniques focused on the GoF design patterns, some research detected and leveraged the Pree's meta patterns, which are the most primitive design patterns [176,177]. Because some GoF design patterns have similar structures, detecting these patterns produces some false positives. On

the other hand, because each Pree's meta pattern is structurally distinguishable, it is relatively easy to detect these patterns without false positives.

Detecting the meta patterns, Hayashi et al. reduced the computation and maintenance costs of the GoF design pattern detection logic [176]. Posnett et al. proposed a meta pattern detector named Thex that worked on Java bytecode and scaled to larger codebases [177]. Our technique also detects meta patterns for constructing object groups corresponding to concepts.

6.6.3 Recovering Architectural Views

Software architecture describes/defines one of the most high-level aspects of a system; it plays crucial roles in at least six aspects of software development such as understanding, construction, and analysis [178]. However, conceptual architecture is often outdated or non-existent because software evolves through continual modifications since the first version has shipped. Numerous studies have presented various types of techniques for recovering architectural views from existing software artifacts (e.g., source code and execution traces) and human expertise [104]. We described the details of existing techniques in Section 2.3.3.

According to the survey by Ducasse and Pollet [104], many architecture recovery techniques require human expertise, which is leveraged to guide software architecture reconstruction (SAR) processes and validate the results. Human knowledge improves the quality of recovered views and suppresses incorrect outputs; however, those are not always available. Compared with those techniques that require human expertise, our technique is a fully automated approach and thus has a wider range of application.

Some studies presented automated approaches to identifying architectural entities (e.g., components) by clustering techniques as described in Section 2.3.3. Most clustering approaches leverage static dependencies for clustering [115], whereas our techniques clusters objects based on static information (i.e., meta pattern instances) and runtime information (i.e., an execution trace). Static-information-based clustering techniques are expected to be promising for recovering an overall structure/picture of a system because static approaches achieve high coverage of analysis. Meanwhile, dynamic approaches are more suitable for recovering behavioral aspects of specific execution scenarios because the dynamism of object-oriented systems can be an insurmountable obstacle to static analysis.

Allier et al. presented a meta-heuristic search algorithm for identifying components by clustering [121], which is close to our approach in terms of utilizing dynamic information (execution traces). A major drawback of the technique by

Allier et al. is to require, as input, a trace set that covers all the functions of a subject system; that is, users are required to be familiar with all the functionalities of the system, which is not always true in an early stage of program comprehension. Meanwhile, our technique takes a single (and simple) execution trace as input; that is, developers that are not familiar with a subject system can utilize our technique.

The abstraction level of recovered views is another difference between architecture recovery techniques and our approach. Architecture recovery techniques generate much more abstracted views (e.g., component diagrams and architectural models), whereas our technique recovers relatively finer-grained views (i.e., object/class-level interaction views). The abstraction level developers require varies across development tasks they undertake. For example, if developers try to refactor/reconstruct the architecture of a system in order to achieve a significant improvement in system performance, architectural views (a higher-level of abstractions) are preferable rather than object/class-level views. If developers are interested in specific use-case/execution scenarios (e.g., for feature modifications or debugging), finer-grained views that depict important aspects of the scenarios of interest would be more valuable. Thus, architecture recovery techniques and our technique are considered to be complementary to one another.

6.7 Summary

The behavior of an object-oriented program can be visualized as a reverse-engineered sequence diagram, which is a valuable tool for program comprehension; however, owing to the massive size of execution traces, reverse-engineered sequence diagrams often suffer from scalability issues.

To address the issues, in this chapter we proposed a trace summarization technique that reduces the horizontal size of reverse-engineered sequence diagrams. Our technique constructs object groups based on Pree's meta patterns, in which each object corresponds to a concept in the domain of a subject system. Then, given an importance-based object ranking, important object groups are identified. By visualizing intergroup interactions only among the important groups, our technique generates a summarized version of a reverse-engineered diagram that depicts a behavioral overview of important concepts in the subject system.

We evaluated our technique using traces generated from various types of open source software. The results showed that our technique outperformed the state-of-the-art trace summarization technique in terms of horizontal summarization of reverse-engineered sequence diagrams. Regarding the quality of

object grouping, our technique achieved an *F* score (resp. a *Recall*) of 0.670 (resp. 0.793) on average, under the condition of #lifelines < 30 (which is acceptable for manual investigation); this is 0.249 (resp. 0.123) higher than that of the baseline technique. Moreover, our technique imposed a runtime overhead of 129.2 % on average, which is relatively small compared with recent scalable dynamic analysis techniques. In many cases, the runtime overhead of our technique is expected to be acceptable in maintenance tasks.

Overall, our technique can recover a summarized sequence diagram that depicts a behavioral overview, while incurring a small runtime overhead. The resulting summarized diagram is expected to be a valuable tool for developers in an early stage of program comprehension.

7

Conclusion

This chapter concludes this dissertation. We revisit the motivation and proposed techniques, and summarize our contributions. In addition, we show some pointers for future research.

7.1 Summary of Contributions

Program comprehension is a pivotal activity in software maintenance. Software documentation is a vital source of information for program comprehension; however, it tends to be outdated or non-existent in many cases. Moreover, comprehending the behavior of an object-oriented system solely from its source code is cumbersome and error-prone due to its dynamism (e.g., dynamic bindings and reflections).

For aiding behavioral comprehension, recovering a sequence diagram from an execution trace, which reflects the actual state of a system, is a promising approach. However, owing to the massive size of an execution trace, reverse-engineered sequence diagrams are often afflicted by scalability issues.

There are many existing techniques for coping with the massiveness of reverse-engineered sequence diagrams. Most existing studies focus on compacting the vertical size of recovered diagrams or effectively exploring massive-scale diagrams: e.g., compaction of repetitive behavior in execution traces and interactive exploration. To address the scalability issues, decreasing the horizontal size of the diagram is likewise essential. Nonetheless, few studies have

addressed this point. Thus, further development and improvement of reduction techniques that focus on the horizontal direction are needed.

In this dissertation, we proposed a fully automated technique for recovering a summarized sequence diagram of a reasonable size. Our technique summarizes a reverse-engineered sequence diagram with a central focus on reducing the horizontal size of the diagram.

Our technique first models an execution trace with our behavior model (B-model) in a tracer-independent format. By analyzing B-model data, we identify core objects of key classes that implement the most important concepts of a subject system. Besides, our technique constructs object groups, in which each group corresponds to a concept of the subject system, based on Pree's meta patterns. Then, we identify important object groups by using the results of core identification and object grouping. Finally, we visualize intergroup interactions only among important object groups, and thereby recover a summarized sequence diagram that depicts a behavioral overview of the system.

We evaluated our technique with traces generated from various types of open source software. The results showed that our technique outperformed state-of-the-art techniques; the resulting diagram of our technique can be a valuable tool in an early stage of program comprehension.

Major contributions this research presented are as follows.

- Behavior model for behavioral design recovery
- Core object identification technique
- Object grouping technique for recovering a summarized sequence diagram

In the following sections, we describe the summary for each contribution.

7.1.1 Behavior Model for Behavioral Design Recovery

Sequence diagram recovery techniques take execution traces as their input. Most existing studies on sequence diagram recovery develop their own tracers, and the formats of traces vary across those tracers.

We presented a behavior model (B-model) that models the behavior of an object-oriented system for behavioral design recovery. With the B-model, we can formalize an execution trace (runtime information) in a tracer-independent format; this enables us to develop several trace analysis techniques independent of tracers. Moreover, we defined mappings from elements of the B-model to elements of a sequence diagram; a B-model event sequence can be easily converted into a sequence diagram, which fills the gap between the representation of an execution trace and that of a sequence diagram.

To demonstrate the feasibility and sufficiency of the B-model, we showed an application, which calculates a slice of a recovered sequence diagram, built on the B-model. Through two slicing examples using a multi-threaded program and a GUI text editor program, we showed that our slicing application successfully recovered sliced sequence diagrams of a reasonable size; it confirms that the B-model holds finer-grained information required for traditional program analyses, and can be a useful tool for behavioral design recovery. In addition, experiments to validate our core identification and object grouping techniques (Section 5.4 and Section 6.4) likewise show the feasibility and sufficiency of the B-model.

7.1.2 Core Object Identification

The most important concepts of a system are implemented by very few key classes. Comprehending the behavior of objects of such key classes is highly important in an early stage of program comprehension.

We presented a technique for identifying core objects of such key classes, which plays important roles in an execution scenario, from a behavioral point of view. Our core identification technique consists of two steps. First, we identify and eliminate temporary objects, which are trivial to comprehend a behavioral overview of a system, by analyzing dynamic scopes of objects based on reference relations and lifetimes thereof. Then, we estimate the importance of non-temporary objects based on their access frequencies, and thereby identify core objects.

The results of our experiment that uses traces generated from various types of open source software showed that our technique outperformed the state-of-the-art trace summarization technique in terms of identifying core objects. Our technique identified a total of 5–11 core objects within top 7–222 (108 on average) important objects, whereas that of the state-of-the-art technique ranged 148–4,378 (998 on average).

Through a detailed examination of the results, we found that because some temporary objects had high access frequencies, pruning temporaries plays an essential role in our importance estimation based on access frequencies. Moreover, the most important factor in our importance estimation formula was the frequency of write accesses; thus, to attach great importance to state changes of objects is expected to be helpful for core object identification.

In terms of practicality, we measured the runtime overhead incurred by our technique. As a result, our technique imposed a runtime overhead of 167.6 % on average; this is relatively small compared with recent scalable dynamic analysis techniques. Thus, the runtime overhead incurred by our technique is expected

to be acceptable in a development phase.

Overall, our technique can identify core objects much more effective than the state-of-the-art technique, with a small runtime overhead.

7.1.3 Object Grouping at a Concept Level for Recovering a Summarized Sequence Diagram

To improve maintainability, in object-oriented programming, a concept is often divided into several classes by using design patterns. From the perspective of program comprehension, it increases the number of design elements to comprehend, and enlarges the amount of effort required for program comprehension.

We presented techniques for constructing object groups and recovering a summarized sequence diagram. Our technique constructs object groups at a concept-level based on Pree's meta patterns that are the most primitive design patterns. We detect all the occurrences of Pree's meta patterns in program code, and then groups template and hook objects involved in the same meta pattern; this reunifies divided concepts as object groups.

Given the results of core object identification, we identify important object groups. By visualizing intergroup interactions only among identified important groups, we obtain a summarized sequence diagram of a reasonable size that depicts object interactions at a concept-level.

In an early stage of program comprehension where developers do not attain a detailed knowledge of a subject system, they prefer highly abstracted views to comprehend behavioral overviews rather than finer-grained views depicting detailed interactions. Because our resulting summarized sequence diagram provides a behavioral overview (an abstracted view) of important object groups (important concepts), our solution can be a valuable tool in an early stage of program comprehension.

The results of our experiments using various types of open source software showed that our technique outperformed the state-of-the-art trace summarization technique in terms of reducing the horizontal size of a reverse-engineered sequence diagram. Regarding the quality of object grouping, under the condition of #lifelines < 30 (which is acceptable for manual investigation), our technique achieved an *F*-score (resp. a *Recall*) of 0.670 (resp. 0.793); this is 0.249 (resp. 0.123) higher than that of the state-of-the-art technique.

Through a detailed examination of the results of our experiment, we found that grouping delegation objects together with template and hook objects improve the quality of object grouping: the *F*-score (resp. the *Recall*) was improved by 0.009 (resp. 0.155).

We investigated the prevalence of meta patterns and delegations in order

to make it clear to what extent our technique (i.e., object grouping based on design patterns) can be applied. The results showed that, on average, 33.2 % of all the method invocations in a subject system are template/hook/delegation method invocations, and 50.2 % of all the non-temporary objects are grouped by our technique. There are numerous grouping opportunities found by our technique; thus, we expect that our technique has a wide range of application. Moreover, our technique achieved high-quality object grouping, regardless of the rates of meta patterns and delegations used; important objects of important concepts tended to utilize meta patterns and delegations.

From the perspective of practicality, we measured the runtime overhead incurred by our technique. As a result, our technique imposed a runtime overhead of 129.2 % on average. This overhead is relatively small compared with recent scalable dynamic analysis techniques; thus, it is acceptable in a development phase.

Overall, our technique can recover a summarized sequence diagram with a small runtime overhead. The recovered diagram depicts a behavioral overview of important concepts in a subject system, which is expected to be a valuable tool in an early stage of program comprehension.

7.2 Opportunities for Future Research

Finally, we show some pointers for future research.

Support for a middle or later stage of program comprehension

In this dissertation, we proposed a technique for recovering a summarized sequence diagram depicting a behavioral overview of a system, which is helpful in an early stage of program comprehension. As the understanding of a system deepens, developers prefer finer-grained view depicting detailed interactions of objects according to tasks they undertake.

Developing techniques for supporting a middle or later stage of program comprehension is a possible direction for future research. For example, it is valuable to develop a technique that takes some queries (e.g., class names of interest) and generates a compact and finer-grained view related to the queries, such as our slicing application described in Section 4.3. Techniques for recovering and digging multi-grained design diagrams in an interactive fashion are likewise expected to be valuable.

Linking behavioral views with other design views

Visualizing the structures and behavior of a system from several different

aspects (e.g., structural, behavioral, and architectural aspects) help developers promote their understandings of a system.

Linking structural or architectural views recovered by existing techniques with behavioral views recovered by our techniques is expected to be effective for aiding program comprehension.

Conformance checking using recovered sequence diagrams

Reverse-engineered sequence diagrams reflect the actual state of a system, whereas forward-designed diagrams (if they are not outdated) state some design constraints that the system should respect. Comparing recovered diagrams and forward-designed ones could reveal some constraint violations that might lead to system failures. Besides, we can regard recovered diagrams as specifications in order to prevent unsafe changes in future development activities. Developing these types of conformance checking techniques is a possible direction for future research.

Bibliography

[1] A. V. Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, August 1995.

[2] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, October 2018.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[4] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

[5] M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Proc. of the International Workshop on Program Comprehension*, pages 181–191, May 2005.

[6] S. G. M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. dissertation, Delft University of Technology, 2009.

[7] 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. *Journal of Software Maintenance and Evolution*, 20(4):291–315, 2008.

[8] B. Cornelissen, A. Zaidman, A. v Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35:684–702, 2009.

[9] S. Bassil and R. K. Keller. Software visualization tools: survey and analysis. In *Proc. of the International Workshop on Program Comprehension*, pages 7–17, May 2001.

[10] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proc. of the International Conference on Program Comprehension*, pages 279–290, 2014.

[11] W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, and R. Tober. Integration of static and dynamic code analysis for understanding legacy source code. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 543–552, October 2016.

[12] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou. Automatic detection of outdated comments during code changes. In *Proc. of the Annual Computer Software and Applications Conference*, volume 01, pages 154–163, July 2018.

[13] Unified modeling language specification. `https://www.omg.org/spec/UML/`.

[14] T. A. Ghaleb, M. A. Alturki, and K. Aljasser. Program comprehension through reverse-engineered sequence diagrams: A systematic review. *Journal of Software: Evolution and Process*, 2018.

[15] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proc. of the International Conference on Software Engineering*, pages 282–291, 2006.

[16] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 535–552, 2007.

[17] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of java program. In *Proc. of the International Workshop on Principles of Software Evolution*, pages 148–154, 2005.

[18] D. Myers, M.-A. Storey, and M. Salois. Utilizing debug information to compact loops in large program traces. In *Proc. of the European Conference on Software Maintenance and Reengineering*, pages 41–50, 2010.

[19] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proc. of the International Workshop on Dynamic Analysis*, pages 8–14, 2008.

[20] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. of the International Conference on Program Comprehension*, pages 181–190, 2006.

[21] M. Srinivasan, Y. Lee, and J. Yang. Enhancing object-oriented programming comprehension using optimized sequence diagram. In *Proc. of the International Conference on Software Engineering Education and Training*, pages 81–85, April 2016.

[22] M. Srinivasan, J. Yang, and Y. Lee. Case studies of optimized sequence diagram for program comprehension. In *Proc. of the International Conference on Program Comprehension*, pages 1–4, 2016.

[23] T. Ishio, Y. Watanabe, and K. Inoue. AMIDA: a sequence diagram extraction toolkit supporting automatic phase detection. In *Proc. of Companion of the International Conference on Software Engineering*, pages 969–970, 2008.

[24] S. Jayaraman, B. Jayaraman, and D. Lessa. Compact visualization of java program execution. *Software: Practice and Experience*, 47(2):163–191, 2016.

[25] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, and A. Shafiee. Stratified sampling of execution traces: Execution phases serving as strata. *Science of Computer Programming*, 78(8):1099–1118, 2013.

[26] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. In *Proc. of the International Conference on Software Engineering Research, Management and Applications*, pages 207–214, May 2010.

[27] H. Pirzadeh and A. Hamou-Lhadj. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. In *Proc. of the International Conference on Engineering of Complex Computer Systems*, pages 221–230, 2011.

[28] H. Pirzadeh and A. Hamou-Lhadj. A software behaviour analysis framework based on the human perception systems. In *Proc. of the International Conference on Software Engineering*, pages 948–951, May 2011.

[29] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah. Exploiting text mining techniques in the analysis of execution traces. In *Proc. of the International Conference on Software Maintenance*, pages 223–232, 2011.

[30] R. Sharp and A. Rountev. Interactive exploration of uml sequence diagrams. In *Proc. of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, 2005.

[31] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. v Deursen, and J. J. v Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[32] B. Cornelissen, A. Zaidman, and A. v Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37:341–355, 2011.

[33] D. Myers and M.-A. Storey. Using dynamic analysis to create trace-focused user interfaces for IDEs. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 367–368, 2010.

[34] H. Grati, H. Sahraoui, and P. Poulin. Extracting sequence diagrams from execution traces using interactive visualization. In *Proc. of the Working Conference on Reverse Engineering*, pages 87–96, 2010.

[35] K. Lyu, K. Noda, and T. Kobayashia. SDExplorer: A generic toolkit for smoothly exploring massive-scale sequence diagram. In *Proc. of the International Conference on Program Comprehension*, pages 380–384, 2018.

[36] T. Toda, T. Kobayashi, N. Atsumi, and K. Agusa. Grouping objects for execution trace analysis based on design patterns. In *Proc. of the Asia-Pacific Software Engineering Conference*, pages 25–30, 2013.

[37] P. Dugerdil and J. Repond. Automatic generation of abstract views for legacy software comprehension. In *Proc. of the India Software Engineering Conference*, pages 23–32, 2010.

[38] L. Tahvildar and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(45):331–361, 2004.

[39] M. Bauer and F. Karlsruhe. Analyzing software systems by using combinations of metrics. In *Proc. of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering (Lecture Notes in Computer Science)*, pages 1–8, 1999.

[40] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.

[41] I. Şora. Helping program comprehension of large software systems by identifying their most important classes. In *Proc. of the International Conference on Evaluation of Novel Approaches to Software Engineering Revised Selected Papers*, pages 122–140, 2016.

[42] I. Şora and D. Todinca. Using fuzzy rules for identifying key classes in software systems. In *Proc. of the International Symposium on Applied Computational Intelligence and Informatics*, pages 317–322, May 2016.

[43] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. dissertation, University of Antwerp, 2006.

[44] A. Zaidman. Scalability solutions for program comprehension through dynamic analysis. In *Proc. of the Conference on Software Maintenance and Reengineering*, pages 327–330, March 2006.

[45] 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 Transactions on Information and Systems*, E95.D(4):959–969, 2012.

[46] K. NODA, T. KOBAYASHI, K. AGUSA, and S. YAMAMOTO. Sequence diagram slicing. In *Proc. of the Asia-Pacific Software Engineering Conference*, pages 291–298, December 2009.

[47] K. NODA, T. KOBAYASHI, and N. ATSUMI. Identifying core objects for trace summarization by analyzing reference relations and dynamic properties. *IEICE Transactions on Information and Systems*, E101.D(7):1751–1765, 2018.

[48] K. NODA, T. KOBAYASHI, T. TODA, and N. ATSUMI. Identifying core objects for trace summarization using reference relations and access analysis. In *Proc. of the Annual Computer Software and Applications Conference*, pages 13–22, July 2017.

[49] K. Noda, T. Kobayashi, and K. Agusa. Execution trace abstraction based on meta patterns usage. In *Proc. of the Working Conference on Reverse Engineering*, pages 167–176, October 2012.

[50] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proc. of the International Conference on Software Maintenance*, pages 602–611, 2001.

[51] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *Proc. of the International Conference on Automated Software Engineering*, pages 602–613, September 2016.

[52] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, August 2016.

[53] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proc. of the International Conference on Program Comprehension*, pages 279–290, 2014.

[54] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini. Call graph construction for java libraries. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 474–486, 2016.

[55] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[56] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 264–280, 2000.

[57] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. of the International Conference on Software Maintenance*, pages 13–22, August 1999.

[58] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proc. of the International Conference on Software Engineering*, pages 56–66, May 2009.

[59] Java platform debugger architecture. `https://docs.oracle.com/en/java/javase/11/docs/specs/jpda/architecture.html`.

[60] ASM. `https://asm.ow2.io/`.

[61] Javassist. `http://www.javassist.org/`.

[62] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 535–552, 2007.

[63] V. K. Palepu, G. Xu, and J. A. Jones. Dynamic dependence summaries. *ACM Transactions on Software Engineering and Methodology*, 25(4):30:1–30:41, January 2017.

[64] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[65] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. of the International Conference on Software Maintenance*, pages 190–198, 1998.

[66] Y. Arimatsu, Y. Ishida, K. Noda, and T. Kobayashi. Enriching api documentation by relevant api methods recommendation based on version history. In *Proc. of the International Workshop on Dynamic Software Documentation*, pages 15–16, Sep. 2018.

[67] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proc. of the International Conference on Software Engineering*, pages 428–439, 2016.

[68] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proc. of the International Conference on Automated Software Engineering*, pages 509–519, 2018.

[69] A. Marcus. Semantic driven program analysis. In *Proc. of the International Conference on Software Maintenance*, pages 469–473, September 2004.

[70] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proc. of the International Conference on Software Maintenance*, pages 469–478, September 2006.

[71] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, July 1989.

[72] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. J. Malton. Using design recovery techniques to transform legacy systems. In *Proc. of the International Conference on Software Maintenance*, pages 622–631, November 2001.

[73] Visual Studio IDE. `https://visualstudio.microsoft.com`.

[74] Visual Paradigm. `https://www.visual-paradigm.com/`.

[75] A. M. Fernández-Sáez, M. R. V. Chaudron, M. Genero, and I. Ramos. Are forward designed or reverse-engineered uml diagrams more helpful for code maintenance?: A controlled experiment. In *Proc. of the International Conference on Evaluation and Assessment in Software Engineering*, pages 60–71, 2013.

[76] M. H. Osman, M. R. V. Chaudron, and P. v. d Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *Proc. of the International Conference on Software Maintenance*, pages 140–149, September 2013.

[77] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proc. of the International Conference on Program Comprehension*, pages 110–121, 2014.

[78] X. Yang, D. Lo, X. Xia, and J. Sun. Condensing class diagrams with minimal manual labeling cost. In *Proc. of the Annual Computer Software and Applications Conference*, pages 22–31, 2016.

[79] P. C. Quinn and R. Bhatt. Perceptual organization in infancy: Bottom-up and top-down influences. *Optometry and Vision Science*, 86(6):589–594, May 2009.

[80] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.

[81] S. Munakata, T. Ishio, and K. Inoue. OGAN: Visualizing object interaction scenarios based on dynamic interaction context. In *Proc. of the International Conference on Program Comprehension*, pages 283–284, 2009.

[82] P. Dugerdil and S. Jossi. Computing dynamic clusters. In *Proc. of the India Software Engineering Conference*, pages 65–74, 2009.

[83] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu. SEAT: a usable trace analysis tool. In *Proc. of the International Workshop on Program Comprehension (IWPC'05)*, pages 157–160, May 2005.

[84] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (VET): An interactive plugin-based visualisation tool. In *Proc. of the Australasian User Interface Conference*, pages 153–160, 2006.

[85] D. F. Jerding and J. T. Stasko. The information mural: a technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271, July 1998.

[86] A. Zaidman, B. D. Bois, and S. Demeyer. How webmining and coupling metrics improve early program comprehension. In *Proc. of the International Conference on Program Comprehension*, pages 74–78, June 2006.

[87] F. Perin, L. Renggli, and J. Ressia. Ranking software artifacts. In *Proc. of the Workshop on FAMIX and Moose in Reengineering*, 2010.

[88] I. Şora. Finding the right needles in hay: Helping program comprehension of large software systems. In *Proc. of the International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 129–140, April 2015.

[89] I. Sora and C. Cernzanu-Glvan. Investigating attributes that characterize important classes in object oriented systems. In *Proc. of the International Symposium on Applied Computational Intelligence and Informatics*, pages 1–5, May 2018.

[90] D. Steidl, B. Hummel, and E. Juergens. Using network analysis for recommendation of central software classes. In *Proc. of the Working Conference on Reverse Engineering*, pages 93–102, October 2012.

[91] P. Meyer, H. Siy, and S. Bhowmick. Identifying important classes of large software systems through k-core decomposition. *Advances in Complex Systems*, 17(07n08), 2014.

[92] Y. Ding, B. Li, and P. He. An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering*, 2016(3858637), 2016.

[93] M. Hammad, M. L. Collard, and J. I. Maletic. Measuring class importance in the context of design evolution. In *Proc. of the International Conference on Program Comprehension*, pages 148–151, June 2010.

[94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[95] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[97] J. DONG, Y. ZHAO, and T. PENG. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855, 2009.

[98] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, November 2006.

[99] N. Pettersson. Measuring precision for static and dynamic design pattern recognition as a function of coverage. In *Proc. of the International Workshop on Dynamic Analysis*, pages 1–7, 2005.

[100] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the International Workshop on Dynamic Analysis*, pages 29–32, 2003.

[101] H. Huang, S. Zhang, J. Cao, and Y. Duan. A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems and Software*, 75(1):69–87, 2005. Software Engineering Education and Training.

[102] N. Medvidovic, A. Egyed, and P. Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *Proc. of the International Software Requirements to Architectures Workshop*, pages 61–68, May 2003.

[103] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. In *Proc. of the International Conference on Software Engineering*, pages 901–910, May 2013.

[104] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[105] C. Liu, B. v Dongen, N. Assy, and W. M. P. v. d Aalst. Component behavior discovery from software execution data. In *Proc. of the Symposium Series on Computational Intelligence*, pages 1–8, December 2016.

[106] C. Liu, B. v Dongen, N. Assy, and W. M. P. v. d Aalst. Component interface identification and behavioral model discovery from software execution

data. In *Proc. of the International Conference on Program Comprehension*, pages 97–107, 2018.

[107] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: a system for discovering architectures from running systems. In *Proc. of the International Conference on Software Engineering*, pages 470–479, May 2004.

[108] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 271–283, 1998.

[109] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *Proc. of the International Workshop on Program Comprehension*, pages 237–244, May 2003.

[110] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proc. of the European Conference on Software Maintenance and Reengineering*, pages 309–318, March 2004.

[111] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. of the International Conference on Software Maintenance*, pages 34–43, October 2002.

[112] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proc. of the European Conference on Software Maintenance and Reengineering*, pages 112–121, March 2005.

[113] D. Amyot and G. Mussbacher. User requirements notation: The first ten years, the next ten years. *Journal of Software*, 6(5):747–768, May 2011.

[114] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *Proc. of the International Conference on Software Maintenance*, pages 72–81, September 2004.

[115] A. Christl, R. Koschke, and M.-. Storey. Equipping the reflexion method with automated clustering. In *Proc. of the Working Conference on Reverse Engineering*, November 2005.

[116] S. Kebir, A. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Proc. of*

the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pages 181–190, August 2012.

[117] M.-S. Choi and E.-S. Cho. A component identification technique from object-oriented model. In *Proc. of the International Conferences on Computational Science and Its Applications*, pages 778–787, 2005.

[118] S. Mishra, D. Kushwaha, and A. Misra. Creating reusable software component from object-oriented legacy system through reverse engineering. *Journal of Object Technology*, 8(5):133–152, July 2009.

[119] H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56(1):99–116, 2005. New Software Composition Concepts.

[120] A. Shatnawi, A.-D. Seriai, H. Sahraoui, and Z. Alshara. Reverse engineering reusable software components from object-oriented apis. *Journal of Systems and Software*, 131:442–460, 2017.

[121] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *Proc. of the International Symposium on Component-Based Software Engineering*, pages 216–231, 2010.

[122] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proc. of the European Conference on Software Maintenance and Reengineering*, pages 47–55, March 2002.

[123] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 2018. early access. DOI: `https://doi.org/10.1109/TSE.2018.2827384`.

[124] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proc. of the International Conference on Automated Software Engineering*, pages 371–382, November 2009.

[125] N. Walkinshaw and M. Hall. Inferring computational state machine models from program executions. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 122–132, October 2016.

[126] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 35–44, 2007.

[127] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proc. of the International Conference on Automated Software Engineering*, pages 295–306, 2009.

[128] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian specification learning for finding api usage errors. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 151–162, 2017.

[129] D. Fahland, D. Lo, and S. Maoz. Mining branching-time scenarios. In *Proc. of the International Conference on Automated Software Engineering*, pages 443–453, November 2013.

[130] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the International Conference on Software Engineering*, pages 213–224, 1999.

[131] M. Brünink and D. S. Rosenblum. Mining performance specifications. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 39–49, 2016.

[132] Y. Kang, B. Ray, and S. Jana. APEx: Automated inference of error specifications for c apis. In *Proc. of the International Conference on Automated Software Engineering*, pages 472–482, 2016.

[133] jpacman. `http://code.google.com/p/jpacman/`.

[134] A. Hamou-Lhadj and T. C. Lethbridge. A metamodel for the compact but lossless exchange of execution traces. *Software & Systems Modeling*, 11(1):77–98, February 2012.

[135] M. Weiser. Program slicing. In *Proc. of the International Conference on Software Engineering*, pages 439–449, 1981.

[136] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[137] L. Alawneh, A. Hamou-Lhadj, and J. Hassine. Towards a common metamodel for traces of high performance computing systems to enable software analysis tasks. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering*, pages 111–120, March 2015.

[138] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, 2004.

[139] T. Wang and A. Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30(2):10:1–10:49, March 2008.

[140] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[141] Object constraint language specification. `https://www.omg.org/spec/OCL/`.

[142] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the International Conference on Software Engineering*, pages 221–230, May 2001.

[143] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34, 2002.

[144] Java virtual machine profiler interface. `https://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/jvmpi.html`.

[145] M. B. Zaidi, J. Ha, and E. E. Johnson. Lossless trace compression. *IEEE Transactions on Computers*, 50:158–173, 02 2001.

[146] T. Chilimbi, R. Jones, and B. Zorn. Designing a trace format for heap allocation events. In *Proc. of the International Symposium on Memory Management*, pages 35–49, 2000.

[147] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 59–70, 2008.

[148] SELogger. `https://github.com/takashi-ishio/selogger`.

[149] T. Matsumura, T. Ishio, Y. Kashima, and K. Inoue. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proc. of the International Conference on Program Comprehension*, pages 253–257, 2014.

[150] wro4j. `http://wro4j.github.io/wro4j/`.

[151] JHotDraw. `http://www.jhotdraw.org/`.

[152] jEdit. `http://www.jedit.org/`.

[153] Apache JMeter. `http://jmeter.apache.org/`.

[154] Apache Ant. `http://ant.apache.org/`.

[155] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 324–341, 1996.

[156] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller. Feedback-directed instrumentation for deployed javascript applications. In *Proc. of the International Conference on Software Engineering*, pages 899–910, 2016.

[157] J. Huang. Scalable thread sharing analysis. In *Proc. of the International Conference on Software Engineering*, pages 1097–1108, 2016.

[158] A. Lienhard, S. Ducasse, and T. Gırba. Taking an object-centric view on dynamic information with object flow analysis. *Computer Languages, Systems and Structures*, 35(1):63–79, April 2009.

[159] R. Vanciu and M. Abi-Antoun. Ownership object graphs with dataflow edges. In *Proc. of the Working Conference on Reverse Engineering*, pages 267–276, 2012.

[160] M. Abi-Antoun, Y. Wang, E. Khalaj, A. Giang, and V. Rajlich. Impact analysis based on a global hierarchical object graph. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering*, pages 221–230, 2015.

[161] W. Pree. Meta patterns — a means for capturing the essentials of reusable object-oriented design. In *Proc. of the European Conference on Object-Oriented Programming*, pages 150–162, 1994.

[162] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., 1995.

[163] JModeller. `https://sourceforge.net/p/jhotdraw/svn/HEAD/tree/trunk/`.

[164] Become a programming picasso with JHotDraw. https://www.javaworld.com/article/2074997/swing-gui-programming/become-a-programming-picasso-with-jhotdraw.html.

[165] wro4j design overview. https://wro4j.readthedocs.io/en/stable/DesignOverview/.

[166] Apache JMeter user's manual. https://jmeter.apache.org/usermanual/index.html.

[167] Apache JMeter wiki: JMeterTestExecution. https://wiki.apache.org/jmeter/JMeterTestExecution.

[168] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. In *Proc. of the KDD Workshop on Text Mining*, 2000.

[169] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. Technical Report #00-034, University of Minnesota, Minneapolis, MN, USA, 2000.

[170] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. *Science of Computer Programming*, 36(2):267–284, 2000.

[171] D. Lo and S. Maoz. Towards succinctness in mining scenario-based specifications. In *Proc. of the International Conference on Engineering of Complex Computer Systems*, pages 231–240, April 2011.

[172] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Inferring hierarchical motifs from execution traces. In *Proc. of the International Conference on Software Engineering*, pages 776–787, 2018.

[173] N. Busany and S. Maoz. Behavioral log analysis with statistical guarantees. In *Proc. of the International Conference on Software Engineering*, pages 877–887, 2016.

[174] Y. Ishida, Y. Arimatsu, L. Kaixie, G. Takagi, K. Noda, and T. Kobayashi. Generating an interactive view of dynamic aspects of api usage examples. In *Proc. of the International Workshop on Dynamic Software Documentation*, pages 13–14, September 2018.

[175] A. Ampatzoglou, S. Charalampidou, and I. Stamelos. Research state of the art on GoF design patterns: A mapping study. *Journal of Systems and Software*, 86(7):1945–1964, 2013.

[176] S. HAYASHI, J. KATADA, R. SAKAMOTO, T. KOBAYASHI, and M. SAEKI. Design pattern detection by using meta patterns. *IEICE Transactions on Information and Systems*, E91.D(4):933–944, 2008.

[177] D. Posnett, C. Bird, and P. Devanbu. Thex: Mining metapatterns from java. In *Proc. of the Working Conference on Mining Software Repositories*, pages 122–125, May 2010.

[178] D. Garlan. Software architecture: A roadmap. In *Proc. of the Conference on The Future of Software Engineering*, pages 91–101, 2000.

Acronyms

11-Con 1:1 connection pattern.

11-RCon 1:1 recursive connection pattern.

11-RUni 1:1 recursive unification pattern.

1N-Con 1:N connection pattern.

1N-RCon 1:N recursive connection pattern.

1N-RUni 1:N recursive unification pattern.

API Application programming interface.

AST Abstract syntax tree.

B-model Behavior model.

CHA Class hierarchy analysis.

COIT Core object identification technique.

CRG Class relation graph.

CTF Compact trace format.

DAG Directed acyclic graph.

DSL Domain specific language.

FPS Frames per second.

GA Genetic algorithm.

GoF Gang of Four.

GUI Graphical user interface.

HITS Hyperlink-induced topic search.

HPC High-performance computing.

HTTP Hypertext transfer protocol.

JPDA Java platform debugger architecture.

JVM Java virtual machine.

JVMPI Java virtual machine profiling interface.

KLOC Kilo lines of code.

MP Our object grouping technique based on Pree's meta patterns, which disallows delegate methods to appear in a chain of template and hook method calls. The details are described in Section 6.3.2.

MP+D Our object grouping technique based on Pree's meta patterns, which allows delegate methods to appear in a chain of template and hook method calls. The details are described in Section 6.3.2.

MPI Message passing interface.

MSC Message sequence chart.

MTF2 MPI (message passing interface) trace format2.

MVC Model-view-controller.

OCL Object constraint language.

ORG Object reference graph.

OSS Open source software.

PofEAA Patterns of enterprise application architecture.

POSA Pattern-Oriented Software Architecture.

REA Reference escape analysis.

RTA Rapid type analysis.

SA Simulated annealing.

SAR Software architecture reconstruction.

SD Sequence diagram.

STEP-DL STEP definition language.

TF-IDF Term frequency-inverse document frequency.

UCM Use case map.

UI User interface.

UML Unified modeling language.

Uni Unification pattern.

VTA Variable-type analysis.

Curriculum Vitae

Personal Data

Full name
Kunihiro NODA

Date of birth
February 13th, 1987

Education

April 2005 – March 2009: B.Sc., Engineering
School of Engineering, Nagoya University, Japan

April 2009 – March 2011: M.Sc., Information Science
Graduate School of Information Science, Nagoya University, Japan

April 2016 – March 2019: Ph.D., Engineering
Department of Computer Science, School of Computing, Tokyo Institute of Technology, Japan

Work Experience

April 2011 – February 2016
Software engineer at an automotive company, Aichi, Japan

June 2016 – February 2019
Research assistant at Tokyo Institute of Technology, Japan

December 2017 – May 2018
Teaching assistant at Tokyo Institute of Technology, Japan

April 2018 – March 2019
JSPS research fellowships for young scientists (DC2), Japan

Publications and Awards

Major Publications

1. **Kunihiro Noda**, Takashi Kobayashi, and Noritoshi Atsumi, "Identifying Core Objects for Trace Summarization by Analyzing Reference Relations and Dynamic Properties," IEICE Transaction on Information and Systems, vol.E101-D, no.7, pp.1751–1765, Jul. 2018.

2. **Kunihiro Noda**, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi, "Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis," in Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC), pp.13–22, Turin, Italy, Jul. 2017.

3. **Kunihiro Noda**, Takashi Kobayashi, and Kiyoshi Agusa, "Execution Trace Abstraction based on Meta Patterns Usage," in Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), pp.167–176, Kingston, Canada, Oct. 2012.

4. **Kunihiro Noda**, Takashi Kobayashi, Shinichiro Yamamoto, Motoshi Saeki, and Kiyoshi Agusa, "Reticella: An Execution Trace Slicing and Visualization Tool based on a Behavior Model," IEICE Transaction on Information and Systems, vol.95, no.4, pp.959–969, 2012.

5. **Kunihiro Noda**, Takashi Kobayashi, Kiyoshi Agusa, and Shinichiro Yamamoto, "Sequence Diagram Slicing," in Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC), pp.291–298, Penang, Malaysia, Dec. 2009.

Other Publications

1. Kaixie Lyu, **Kunihiro Noda**, and Takashi Kobayashi, "Toward Interaction based Evaluation of Visualization Approaches to Comprehending Program Behavior," in Proceedings of the Second International Workshop on Mining and Analyzing Interaction Histories (MAINT), Hangzhou, China, Feb. 2019 (to appear).

2. Yoshiya Ishida, Yuu Arimatsu, Lyu Kaixie, Go Takagi, **Kunihiro Noda**, and Takashi Kobayashi, "Generating an Interactive View of Dynamic Aspects of API Usage Examples," in Proceedings of the Third International Workshop on Dynamic Software Documentation (DySDoc), pp.13–14, Madrid, Spain, Sep. 2018.

3. Yuu Arimatsu, Yoshiya Ishida, **Kunihiro Noda**, and Takashi Kobayashi, "Enriching API Documentation by Relevant API Methods Recommendation based on Version History," in Proceedings of the Third International Workshop on Dynamic Software Documentation (DySDoc), pp.15–16, Madrid, Spain, Sep. 2018.

4. Kaixie Lyu, **Kunihiro Noda**, and Takashi Kobayashi, "SDExplorer: a generic toolkit for smoothly exploring massive-scale sequence diagram," in Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC), pp.380–384, Gothenburg, Sweden, May 2018 (***Best Tool Paper Award***)

5. Maaki Nakano, **Kunihiro Noda**, Shinpei Hayashi, and Takashi Kobayashi, "Mediating Turf Battles! Prioritizing Shared Modules in Locating Multiple Features," in Proceedings of IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pp.363–368, Turin, Italy, Jul. 2017.

6. 平ノ内 奎太, **野田 訓広**, 小林 隆志, ‘‘仮想ファイルシステムを用いたプログラム内部状態観測ツールの試作", 電子情報通信学会技術研究報告, SS2018-13, pp.155–160, Jul. 2018.

7. 原口 大和, **野田 訓広**, 小林 隆志, ‘‘メソッド入退出情報を利用した階層的欠陥箇所特定支援手法", 電子情報通信学会技術研究報告, SS2018-12, pp.149–154, Jul. 2018.

8. 中野 真明貴, **野田 訓広**, 小林 隆志, 林 晋平, ‘‘実行トレースの共通性分析に基づく機能開始点の特定", 電子情報通信学会技術研究報告, SS2017-73, pp.51–56, Mar. 2018.

9. **野田 訓広**, 小林 隆志, 渥美 紀寿, ‘‘実行トレース抽象化を目的とした参照関係・アクセス解析によるコアオブジェクト特定", 情報処理学会研究報告ソフトウェア工学 (SE), 2017-SE-195(2), pp.1–8, Mar. 2017 (***Student Research Award, IPSJ Computer Science Research Award for Young Scientists***).

10. **野田 訓広**, 小林 隆志, “リバースエンジニアリングによる実用的な設計情報復元に向けて”, ウインターワークショップ 2017・イン・飛騨高山, pp.63–64, Jan. 2017.
11. **野田 訓広**, 小林 隆志, 阿草 清滋, “メタパターン適用情報に基づくオブジェクトの協調動作履歴可視化ツール”, 情報処理学会研究報告ソフトウェア工学 (SE), 2011-SE-171(1), pp.1–10, Mar. 2011.
12. **野田 訓広**, 小林 隆志, 山本 晋一郎, 阿草 清滋, “高精度なデータ依存解析に基づくシーケンス図スライシング手法”, 情報処理学会研究報告ソフトウェア工学 (SE), 2009-SE-163(31), pp.233–240, Mar. 2009.

Awards

1. Best Tool Paper Award, The 26th IEEE/ACM International Conference on Program Comprehension, May 2018.
2. IPSJ Computer Science Research Award for Young Scientists, Jul. 2017.
3. Student Research Award, IPSJ/SIGSE, Mar. 2017.