EXTENDING SEQUENCE DIAGRAMS FOR BETTER COMPREHENSION OF PROGRAM CONTROL-FLOW

BY

TAHER AHMED MOHAMMED GHALEB

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

DECEMBER 2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **TAHER AHMED MOHAMMED GHALEB** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

[illegible]
Dr. Musab Alturki (Adviser)

[illegible]
Dr. Khalid Aljasser (Co-adviser)

[illegible]
Dr. Moataz Ahmed (Member)

M Alshayeb
Dr. Mohammad Alshayeb (Member)

[illegible]
Dr. Mahmood Niazi (Member)

[illegible]
Dr. Abdulaziz Alkhoraidly
Department Chairman

[illegible]
Dr. Salam A. Zummo
Dean of Graduate Studies

12/5/16
Date

I dedicate my thesis work to my family and friends. A special feeling of gratitude to my loving parents, whose words of encouragement and push for tenacity ring in my ears. An exclusive dedication to my wife, whose care, patience, and motivation have had a great influence in completing this thesis. I also wish to dedicate this work to my son, daughter, brothers, sisters, and friends. This work would not be possible without their support. I would also like to sincerely thank all teachers who taught me, as their encouragements and motivations have laid the foundation for this work.

ACKNOWLEDGMENTS

First of all, I would like to thank almighty Allah for giving me the ability and strength to work on and accomplish this thesis. I wish to give my sincere appreciation to my advisors Dr. Musab Alturki and Dr. Khalid Aljasser for all their efforts and countless hours of reflecting, reading, encouraging, guiding, and most of all patience throughout the entire work of this thesis. They have been my mentors, teachers, and fathers. I feel very lucky to have them as supervisors for my thesis.

I am also grateful for the committee members of my thesis: Dr. Moataz Ahmed, Dr. Mohammad Alshayeb and Dr. Mahmood Niazi who were more than generous with their expertise and precious time. They have been dedicating valuable time out of their busy schedule in order to provide useful feedback to improve the quality of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME:	Taher Ahmed Mohammed Ghaleb
TITLE OF STUDY:	Extending Sequence Diagrams for Better Comprehension of Program Control-flow
MAJOR FIELD:	Computer Science
DATE OF DEGREE:	December 2015

The reverse engineering of sequence diagrams from software systems is an effective approach that can facilitate the comprehension of the behavior of programs for maintenance or learning purposes. After studying the state-of-the-art techniques, we observed that less research has been conducted for investigating how to precisely reflect program control-flow in the UML sequence diagram (SD). Actually, the lack of notations in the standard UML sequence diagram makes it unable to represent most of the interactions and flow of control of any given program. This leads to either losing such information or imprecisely representing them in the produced sequence diagrams, which in turn reduces program comprehension. The main objective of the thesis is to introduce an approach that employs a static program analysis for reverse engineering of sequence diagrams extended with notations that can *expressively and precisely map the program control-flow represented in the source code. Our proposed approach composes three main techniques: control-flow information extractor, interactions tracer, and trace visualizer. Gathering program control-flow has been achieved using an extensible compiler, called Polyglot, that facilitates recovering the design of Java programs represented by Abstract Syntax Trees (ASTs). The collected information are then traced using a query-based tracer to identify all entry points, branches, and calls in the program and preserve them in an extended notation of UML 2.0. Finally, the resulting traces are visualized using UML sequence diagrams extended with new expressive notations. A proof-of-concept prototype of the proposed approach has been implemented and then validated over various samples of case studies that cover all our SD extensions, in addition to a well-known open-source Java project called Greenfoot. Finally, we have conducted a controlled experiment to evaluate our proposed SD extensions The evaluation was focused on measuring the effectiveness of the extensions for program comprehension as well as their complexity and precision compared with the UML standard of sequence diagrams. The obtained results indicate that having extended notations in sequence diagrams can significantly improve the understandability of the control-flow of programs. Although they could be somewhat complicated, they still have an advantage to providing a precise representation about the program control flow. Finally, we conclude this thesis by highlighting the main characteristics and limitation of our work and the possible directions for research in the future.*

ملخص الرسالة

الاسم الكامل: طاهر أحمد محمد غالب

عنوان الرسالة: توسيع المخططات التسلسلية لفهم تدفق عناصر التحكم بالبرامج بشكل أفضل

التخصص: علوم الحاسب الآلي

تاريخ الدرجة العلمية: ديسمبر 2015

الهندسة العكسية للمخططات التسلسلية من الأنظمية البرمجية هي وسيلة فعّالة لتسهيل فهم البرامج لغرض الصيانة أو التعلم. في بداية هذا العمل، قمنا بعمل مسح أدبي لكل التقنيات في هذا المجال واستكشاف ميزاتها وعيوبها. في الواقع، لاحظنا أن تلك التقنيات ترتكز على تحليل البرامج إما بشكل ساكن أو ديناميكي أو الإثنان معاً وذلك لدعم فهم البرامج. بعد استكشاف الفجوات خلالها، قررنا تطوير طريقة خاصة بنا لفهم البرامج بشكل فعّال تختلف عن تلك الموجودة سابقاً من ناحية الهدف والمضمون. نحن نسعى من خلال تقنيتنا المقترحة إلى تغطية العديد من الفجوات التى لم تُعالج من قبل وذلك بتوفير حلول موسعة ومرنة تتركز على زيادة كفاءة فهم البرامج وذلك من خلال استخدام العديد من الإضافات للمخططات التسلسلية والتي نسعى من خلالها لعرض التفاعلات والأحداث الموجودة داخل البرامج بترميزات معبرة، سهلة الفهم، وغير معقدة. تتكون التقنية المقترحة في هذه الرسالة من ثلاث عمليات رئيسية وهي (1) استخراج المعلومات عن البرامج المراد فهمها من شفرة المصدر الخاصة بها (2) تحليل هذه المعلومات لتتبع كل التفاعلات الموجودة فيها (3) عرض هذه التفاعلات بشكل مرئي على واجهة المستخدم الرسومية. تتبلور حصيلة هذه الرسالة في إنتاج نموذج للتقنية المقترحة ومبنى خصيصاً للتعامل مع البرامج المصصمة بلغة الجافا وتم اختباره والتحقق من دقة عمله بتطبيقه على بعض مشاريع البرمجية المفتوحة المصدر. إضافةً إلى ذلك، تم تقييم المنتج النهائي باستخدام تجربة محكمة تقوم بقياس مدى فهم المستخدمين للبرامج باستخدام الطريقة المقترحة مقارنة بالطريقة القياسية وذلك من خلال العديد من المهام التي صممت لهذا الغرض. بعد تحليل نتائج هذه التجربة تبين أن الإضافات التي قمنا بتوسيع المخططات التسلسلية بها كانت بسيطة وغير معقدة، فقد ساعدت المستخدمين على فهم البرامج والأكواد البرمجية بوقت أسرع وبدقة أكبر مما هي عليه في حالة استخدام المخططات التسلسلية الإعتيادية.

CHAPTER 1

INTRODUCTION

1.1 Overview

Software engineers may spend a considerable amount of time with an extensive effort looking over source codes to deduce the structure and/or behavior of software systems before performing any software maintenance task. The reason behind this is related to a poor or nonexistent documentation and specification of that software systems [1]. Such exhausting investigation would reduce the productivity of software engineers as it is not carried out in a systematic manner. Therefore, reverse engineering techniques have been introduced to facilitate the analysis of software components and extract them into a readable and manageable format.

Reverse engineering (RE) is one of the most significant endeavors in software engineering that facilitates recovering and understanding the structure and the behavior of software systems [2]. RE is usually accomplished through a set of processes that, typically, include: analyzing a program statically or dynamically (based on the source code or byte/binary code), and then the collected information is then transformed into a higher level and more abstract representation or models [3]. This representation might then be usable for different activities. For example, it can help in conforming design and behavior of the currently implemented system to its design specifications [4].

In general, reverse engineering of software behavior is more challenging than that of software structure [5]. In particular, existing techniques of analyzing software interactions utilize different methods for identifying software components. The overall process comprises a set of operations: parsing programs, tracing interactions within which, and visualizing the resultant behavior using manageable and understandable diagrams/views. Object construction at runtime, methods polymorphism and other dynamic operations make this process more complicated. The identification of such behavioral operations requires sophisticated techniques to analyze source code and/or trace program execution.

The typical way to represent the program behavior and its interactions is by using sequence diagrams (SDs). Sequence diagrams have the capability of showing the program behavior using a series of messages representing the interactions between classes and objects in that program. In addition, sequence diagrams have been provided as a UML standard, which involves a set of notations that can reflect the program control-flow showing all paths a message can go through.

This thesis conjoins a set of correlated concepts concerning program comprehension, including reverse engineering, program analysis, and program visualization.

1.2 Problem Statement

During our review of the state-of-the-art techniques that aim to improve the understandability of software systems, we deduced that their final representations of program interactions using the standard UML sequence diagrams is still not sufficient for a thorough and precise comprehension of software interactions. In addition, a limited research emphasizes on the appearance of the resulting sequence diagrams and the amount of information about program interactions to be represented within, whereas much research is available for other kinds of views that represent program structure (e.g., class diagrams).

Motivated by this, we aim in this thesis to introduce new extensions to the UML notation of sequence diagrams along with extended elements and fragments. The ultimate purpose of such extensions is to precisely reflect the control-flow of programs for enhancing the ability to understanding the anticipated interactions of programs and how they are represented in the source code. Some of these extensions are simple in a way they extended the UML standard without adding a new notation. In other words, they utilize the existing UML elements of sequence diagrams and use them in a different way. On the other hand, other extensions considered adding new notations to the UML standard represented by a set of connected and meaningful elements.

In order to get an overall summarization of the research problems addressed in this thesis, we formalize them using the following three research questions:

RQ1: How can UML sequence diagrams be extended to effectively enhance the representation of program interactions to improve program comprehension?

Since the main purpose of program comprehension-related techniques is to facilitate understanding the behavior of programs, representing them in a static (lifeless) sequence diagram is still inadequate and does not fully satisfy the purpose if utilized as is. This, indeed, requires improving the current model of UML sequence diagrams with extensions that can increase the understandability of software behavior.

RQ2: How can Polyglot be extended to gather more useful information about program control-flow?

Polyglot is an extensible Java compiler that allows programmers to extend Java or create new domain-specific programming languages [6]. We have specifically chosen Polyglot as it facilitates defining new constructs, syntax, and semantics in a modular way so that the base compiler code is not modified. In addition, it employs the visitor design pattern to access all nodes of the Abstract Syntax Tree (AST). Actually, Polyglot has never been utilized for reverse engineering purposes. Hence, adapting it for analyzing programs and then extract all information about their interactions and control-flow structures that can later be used for program understanding, for sure, requires to be extended properly. Extending Polyglot involves extending its Node Factory along with the compilation passes and visits. should be considered through the extending process as they allow accessing the prospective extensions at any stage of compilation.

RQ3: How to trace program interactions in a way that can generate the representation of the extensions required?

Since Polyglot can assist in recovering the overall program design and control-flow, it is necessary to trace the sequence of interactions of that program to retrieve its expected behavior. This requires developing a set of queries and heuristics capable of identifying all entry point of the program, control structures, different kinds of call messages, and other information that can enhance program understandability.

1.3 Thesis Objectives

Our main objective in this thesis is to introduce the design and specification of a set of extensions to the UML notation of sequence diagrams that can precisely capture the flow of control in the program's source code and consequently enhance program comprehension. We also aim to conduct a static program analysis by (1) extending Polyglot to properly gather information about of Java programs and their control flow; (2) performing a set of queries and heuristics for tracing all interactions within programs; and (3) producing sequence diagrams with extended notations that can precisely reflect the actual flow of control of such programs.

To carefully achieve our objectives, we take into consideration the restrictions of the current notation of UML sequence diagrams along with the limitations of the state-of-the-art techniques. We also endeavor to fulfill our goals by taking into account the research questions RQ1, RQ2, and RQ3 stated earlier.

To carefully address the research questions, it is important to ascertain the following visualization goals:

Preciseness and Expressiveness: all the control flow information of the program should be displayed in the produced diagrams abstractly and precisely. This includes all kinds of loops (e.g., while, for or do-while), calls (simple, nested, chained or recursive) and conditions and alternatives (e.g., 'if', 'if-else', 'switch', '?:', or even 'try-catch-finally'). Also, the target lifelines should refer to classes and objects at the same time, in addition to some others that can represent the interactions between internal system objects with external entities (such as, system console, file system, GUI, remote objects, or system libraries).

Scalability: the resulting diagrams may display all the interactions expected to be executed in the program runtime. Doing so would for sure complicate the resulting diagrams and make them unreadable or untraceable, especially when complex systems are used. Therefore, the produced diagrams should employ scalability facilities that can enable users to visualize both simple and complex programs, without impacting their effectiveness and expressiveness.

Usability and User Interaction: to make the resulting diagrams usable and useful, a number of facilities may be embedded with the elements visualized. Each element in the diagram can, for example, convey its location in the source code, so that whenever a user needs to access the source code of a certain element, he/she can directly navigate to it. In addition, showing documentation comments of methods and lifelines may also assist users in understanding the functionality of a certain element in the diagram.

1.4 Thesis Contributions

The intention of this thesis is to address the mentioned research problems in a way that satisfies our objective. This is accomplished by introducing a program analysis technique that reverse engineers sequence diagrams from the source code of Java programs statically. This technique was constructed using Polyglot [6], an extensible compiler for Java programming language. Essentially, the main purpose of Polyglot is to allow developers to create their own programming languages by extending the compiler of Java instead of doing so from scratch.

Making sequence diagrams more expressive actually requires extending the standard notations of sequence diagram provided in the UML standard with new convenient notations. Consequently, classical tools used for visualizing sequence diagrams would not assist us in achieving our goal. Therefore, we have built our own visualization tool that is capable of reflecting the information gathered via program analysis.

In this thesis, we come up with six correlated contributions. First, we have conducted a thorough review of the current approaches proposed in the literature in this context. Second, we have constructed a language extension that can support reverse engineering sequence diagrams from Java source code. Third, a query-based tracer of program interactions has been developed. Fourth, a set of extensions to the UML sequence diagrams have been designed. Fifth, a trace visualizer has been implemented to produce extended sequence diagrams of the given program. Finally, a controlled experiment has been conducted to evaluate the proposed SD extensions.

1.4.1 A Comprehensive Literature Survey

A detailed literature survey of reverse engineering and program analysis techniques that rely on reverse engineered sequence diagrams as a main output. This survey is composed of brief descriptions of the techniques along with their features and limitations. Then, these techniques have been qualitatively evaluated based on different evaluation criteria.

Outcome: The outcome here is a comprehensive review paper that summarizes and evaluates program analysis techniques and demonstrates their current limitations along with the limitations of their produced sequence diagrams. This paper also presents the different aspects and applications of program comprehension with a concentration on the comprehension of program control flows and how it can be enhanced in the future.

1.4.2 Program Information Extractor as a Language Extension

An extension to the Polyglot compiler is accomplished to support capturing useful information about program behavior. The nature of Polyglot is to parse Java programs by visiting program constructs using the visitor design pattern. During its visits, it checks the validity to the compilation process and produces proper error messages if an error in the syntax or semantics occurs. However, some useful information about programs are not collected by the Polyglot parser generator. For example, documentation comments are not parsed, but we could get a recent Polyglot extension that is capable of doing so. Some other parts of the program are not explicitly defined as nodes of the Abstract Syntax Tree. For example, the if node consists of three nodes: condition, consequence, and alternative. These nodes are defined as *Stmt* nodes, which means that the visitor cannot identify in which part of the construct a method is invoked. Therefore, we have extended Polyglot $NodeFactory$ with the proper AST nodes that enable the visitor to identify them throughout the compilation process.

Outcome: The outcome here is a Polyglot extension, which is given the name *JRev*. This extension is deployed as an executable jar file that can be used by other tracing programs. In addition, the output of this extension, if applied to a certain Java program, would be generated as a log file that is structured as an XML (Extensible Markup Language) tree containing all useful information about programs. Actually, this generated XML document would not represent the sequence of interactions or the behavior(s) of the parsed program. However, it contains all constructs used in the program, class declarations, method declarations, method calls, object constructions, etc. that can be later traced in order to recognize the sequence of method calls that exists in the given program.

1.4.3 A Query-based Tracer of Program Interactions

Once the XML file that contains the information about a certain program is generated, it is fed to the tracer that can follow the sequence of interactions of that program represented by method invocations. This tracer begins with the main method(s) of the program and goes step-by-step through each call individually until it reaches the end of the program. While tracing, the trace logs the information about control-flow constructs as well as the methods invoked in an XMI (XML Metadata Interchange) representation. The resulting XMI file can eventually be viewed by a visualization tool.

Outcome: The upshot of this contribution is another paper that talks about a novel reverse engineering tool that can capture program information using an extensible compiler, and then trace interactions within that program using XPath queries. The paper discusses the feature provided by this tool along with the methodology employed to make the resultant XMI file compatible with the traditional sequence diagrams visualizers.

1.4.4 Extensions to UML Sequence Diagrams

A set of UML sequence diagram extensions is proposed in this thesis. These extensions are meant to enrich sequence diagrams with adequate information that make users understand what is happening in the program without the need to go to its source code. This means that the extended sequence diagrams should not only display the messages between objects and classes of the program, but also the information that users may need to comprehend the program well. For example, comments do not tell anything about program behavior. However, providing them appropriately in sequence diagrams can help users identify the purpose or description of a certain class/object/method.

Outcome: Our product of this contribution is a paper that proposes a set of extension to UML sequence diagrams. The description of these extensions, their purposes, shapes, and XMI representations are elaborated. In addition, we have discussed a strategy that can make extended sequence diagrams readable by other visualization tools.

1.4.5 Trace Visualizer

The motivation behind the construction our own visualization is two fold. First, all conventional sequence diagram visualizers are not capable of rendering the extended elements of sequence diagrams proposed in this thesis. Second, current visualizers lack some scalability and navigation facilities that might boost the program understandability.

Outcome: A proof-of-concept prototype is provided with the facilities intended to uphold program comprehension. The implementation of this prototype involved deploying it as tool dependent on the language extension, interactions tracer, and sequence diagram extensions proposed in this thesis.

1.4.6 Controlled Experiment

In order to evaluate the extended sequence diagrams generated by our technique, we have carried out a controlled experiment that involved an computer-based questionnaire to be filled by a number of participants. The questionnaire is composed of various comprehension tasks that cover the major SD extensions designed in this thesis.

Outcome: The major outcome of this contribution is the conclusions we have obtained regarding our technique and our proposed SD extensions. These conclusions are useful in a way that helped us get an impression about how our work is effective and efficient for program comprehension.

1.5 Thesis Outline

The reminder of this work is structured as follows. Chapter 2 presents a literature review of techniques that aim to improve program comprehension through the use of reverse-engineered sequence diagrams. It also demonstrates the main characteristics of each technique and evaluates the effectiveness of their visualized output. Chapter 3 introduces the proposed program analysis approach along with the prospective extensions to UML sequence diagram that may increase the understandability of software behavior. Chapter 4 highlights the implementation of all stages of our proposed technique as a proof-of-concept prototype. Chapter 5 discusses the evaluation of our proposed extensions to the sequence diagram. To this end, we have used a well-known open-source Java project called *Greenfoot*.

The evaluation was conducted through a controlled experiment, wherein a group of students participated in the evaluation across a computer-oriented questionnaire. Finally, Chapter 6 concludes the thesis, lists the limitation of the approach proposed, and recommends a set of directions in which future work can be carried out.

CHAPTER 2

LITERATURE REVIEW

Reverse engineering of sequence diagrams refers to the process of extracting meaningful information about the behavior of software systems in the form of appropriately generated sequence diagrams [2]. This process has become a practical method for retrieving the behavior of software systems, primarily those with inadequate documentation. Different kinds of approaches have been proposed in the literature for the sake of producing a series of interactions from a given software system, which can later be used for so many purposes. The reason for the wide diversity of approaches is the need to offer sequence diagrams that can cater for the users' specific goals and needs, which can vary wildly depending on the users' perception and understandability of visual representations and the target application domains.

Various techniques have been introduced in the literature with a capability of analyzing software systems, tracing interactions within which, and producing corresponding diagrams that represent the expected or actual behavior of that systems. Each technique mostly relies on a certain tool to accomplish program parsing and tracing. In this chapter, we introduce an extensive exposition of sequence diagrams, their usage, and their current restrictions that can reduce the understandability of software behavior. In addition, we present a set of aspects of program comprehension that should, completely or partially, be considered by any of the proposed techniques. Moreover, existing program analysis techniques aimed to produce sequence diagrams as their output are demonstrated and evaluated based on well-defined evaluation criteria and attributes. The evaluation results can indicate how useful are these techniques towards program comprehension based on how much information about programs they provide. After that, an analysis of the evaluation results is elaborated with a detailed discussion.

2.1 Program Visualization with Sequence Diagrams

As the ultimate objective of reverse engineering is to express meaningful information to humans, a considerable attention in the literature has been given to the generation of understandable and useful visualizations [7]. Visualizing the program behavior is one of the challenges that can directly affect how meaningful the retrieved program information is. Visualizing all possible interactions of non-trivial programs (i.e., complex systems) can produce more complicated sequence diagrams, which are likely to exceed the cognitive abilities of human beings. On the other hand, hiding portions of the interactions of small programs (e.g. introductory student programs) from the produced sequence diagrams may result in

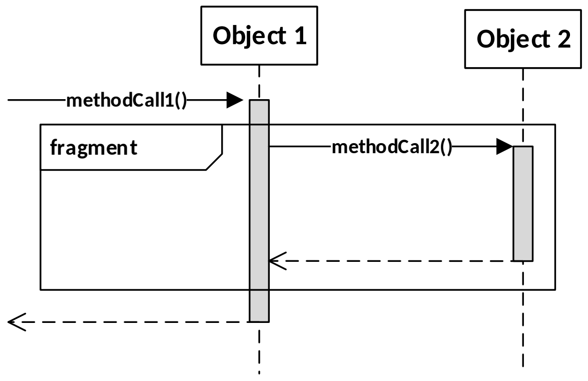


Figure 2.1: An example of a UML sequence diagram

losing important information that might have helped in understanding the programs thoroughly.

2.1.1 UML Sequence Diagrams

Based on the Unified Modeling Language (UML) standard, sequence diagrams can model the flow and interactions of a software system visually with the use of incoming and outgoing messages [8], which facilitate understanding and validating the program logic. In UML 2.0, several more advanced notational elements have been added to provide an expressive representation tool for various programming constructs and functions. As shown in Fig. 2.1, the notational element named fragment provides consistent places where interactions can be categorized into fragments. For example, a frame might be used to represent interactions executed inside alternation or loop constructs; e.g. if, for, etc.

2.1.2 Limitations of UML Sequence Diagrams

Vertical expansion: As they represent a series of interactions, sequence diagrams by their nature can only expand vertically. This makes it difficult for users to have an overall vision of the system behavior, especially when there is a large number of interactions.

Sparse entity behavior: In sequence diagrams, it is easy to observe messages communicated between different objects via method calls. However, it is sometimes difficult for users to recognize what interactions executed inside a certain method as the only possible way to achieve this is to track all incoming and outgoing messages occurring within the execution bar of that method. On the other hand, recognizing all interactions connected to a specific object/class is also tedious, since it also requires tracking all messages received and sent by that object. Therefore, representing only the objects that receive the messages in such interactions is important in order to have more precise sequence diagram, which indeed helps in increasing program understandability [9].

Restricted fragments: Fragments introduced in the latest version of UML (i.e., UML 2.0 [10]) reduced most of the complexity of the sequence diagram representation by abstracting interactions executed in certain blocks of codes called fragments. Although the use of fragments has proved useful in providing better views of program control-flow, fragments do not adapt well with some unstructured control-flow constructs. For example, the *break* fragment defined by UML 2.0 exits from its immediate enclosing loop fragment, which makes it impossible to jump over a number of levels of nesting. This particular limitation was addressed by Rountev et al. [11] who proposed a generalized break fragment that enables exiting from a certain surrounding fragment.

Representing messages using arrows: In the standard UML sequence diagram, all kinds of messages are represented using left-directed or right-directed arrows. Despite the availability of different shapes of arrows in the standard UML sequence diagram, messages can only be represented by single arrows. This kind of representation may not be valid for all kinds of messages that can appear in a program. For example, how can a call that contains several calls inside its arguments be represented using such arrows? The only possible solution is to represent every call (the original call and the calls nested within which) by a separate arrow. This representation, however, is not precise since it actually captures a different scenario where the calls are sequenced rather than nested.

2.1.3 Other Forms of Visualization

Using standard diagrams like UML Sequence Diagrams [8] is normally preferred over non-standard tools to represent the behavior of software systems. Tools that comply with standard rules typically enjoy more usability than others (due to increased interoperability and consistency of generated specifications) and are supported by a wider development community. To enhance program comprehension, several standard and non-standard visualization forms have been used in the literature to represent the program structure and behavior. Although standard visualizations are popular, many users are using other non-standard methods of visualization to overcome their limitations and fulfill specific needs of program comprehension that are not directly supported.

2.2 Program Comprehension

Program comprehension, or program understanding, is defined as the activity of identifying the different aspects of software systems, including the structure and behavior [12]. This activity is important for various purposes, such as maintenance, inspection, extension or reuse of existing software systems. Studies in the literature address program comprehension from different perspectives, such as theories, methods, tools and cognitive process [13].

Challenges in understanding programs are concerned with the complexity of existing (legacy) systems and the amount of information they hide behind their implementation. These challenges are mostly faced when the source code of such systems is not available. Nevertheless, it is also the case even with the availability of source code, especially if no (or inadequate) documentations are extant. Therefore, research trends in this context have been diversified into different areas of software understanding, and the majority of them focus on software visualization [14].

Program analysis techniques, either static or dynamic, represent one of the key approaches that supports program comprehension [15, 7]. They facilitate the extraction of program components into readable and manageable formats. In the case of understanding the structure of software systems, static analysis is indeed sufficient, whereas it needs to be incorporated with dynamic analysis in order to get an overall overview of program behavior.

On the other hand, software visualization are the essential aid of increasing the level of program understandability. Techniques, and most of them are tool-supported, have been proposed in the literature with the goal of enhancing program comprehension through the use of various styles of visualization representing either program from structure or behavior traces.

In general, techniques that aim to improve program comprehension are usually evaluated using controlled experiments, which involve the preparation of a set of tasks that relate to comprehension activities [16]. The main objective of such experiments is to measure the time spent by users to respond to the predefined comprehension tasks and evaluate the correctness of their responses. Such a method of evaluation has widely been used by several research who could evaluate their tools in comparison with other approaches [17, 14, 18].

Our main observation in this context is the absence of relevant studies for evaluating sufficiency and suitability of UML sequence diagrams for program comprehension. As we will see later in this chapter, several techniques used to use sequence diagrams as a main visual representation of program interactions, but none of them has investigated its adequacy for understanding program behavior. Instead of introducing new extensions to the UML notations of sequence diagrams, some techniques incorporated other diagrams in order to supply more information about programs, while the former approach can be more appropriate (details on this are available in sections 2.4 and 2.5).

2.3 Aspects of Program Comprehension

Satisfying the needs of a wide range of stakeholders of the reverse-engineered sequence diagrams requires considering various aspects of comprehending programs. For such an objective, the resulting sequence diagrams need to either include all programs' information or employ different versions of them, where each version covers a particular aspect of comprehension. Actually, having all program information to be shown in a single sequence diagram can complicate it, and its understandability will hence be reduced. Therefore, having in between options may be desirable to cover a large scale of users.

On the other hand, sequence diagrams can be abstracted into a higher level of representation. In other words, instead of showing all objects with their interactions in the diagram, deducing and displaying certain metrics about the number of messages, paths or fragments may provide a better understanding of the overall breadth and depth of the behavior of the system.

2.3.1 Original System's Documentation

The fundamental goal of reverse engineering techniques that recover program's behavior and represent it by means of sequence diagrams is to recover the system's interaction that was originally established in the system's design phase [2]. The purpose of such retrieval may differ from one user to another. In some cases, the system's documentation could be lost and the software engineer needs to restore it from the standing system. In other cases, the developer might need to match the original design with the design of the existing implementation of a system to identify the positive/negative changes that have happened throughout the system's implementation.

2.3.2 Program Control Flow

It is common to have users interested in acquiring sequence diagrams that can capture exactly the control flow of the written programs [19]. Control flow provides an abstraction of all paths, branches, and jumps within which program interactions and messages between objects are executed. Techniques that depend on dynamic analysis always render interactions once they are executed. This, however, leads to the production of all series of interactions in a way that users will miss the different paths where the messages have gone through (i.e., control flow is not preserved). Some dynamic-analysis-based techniques can build specific representations of execution paths, which may not necessarily match the source code exactly. For example, a sequence of messages performed using a *recursive* call may be demonstrated in the sequence diagram as a for or a *while* loop. Although this kind of rendering is generally acceptable, it does not precisely capture the control flow set out in the source code. Therefore, static analysis of the source code of programs is the preferred approach in retrieving the exact program control flow [11].

2.3.3 Program Interactions at Runtime

Interactions carried out throughout program execution constitute a significant percentage of sequence diagrams. Demonstration of such interactions is actually desired by almost all kinds of users. In this perspective, users are interested in understanding all possible program scenarios that can be executed through different parameters [20]. Techniques in this context are not aware of what exactly is being processed in the source code of the program. Instead, they are concerned with the interactions being processed throughout the program runtime [21]. This means that users are only interested in observing the actual interactions executed during the system runtime, regardless of what actually happens in the program control flow [22]. For example, users in this manner would like to inspect the different paths that interactions are directed to within one run of a program, which means that optional (or *opt*) and alternative (or *alt*) fragments might not be desirable in the resulting diagram. A similar case is applied to the actions executed within loops. In this case, users wish to see all the messages sent from one object to another, instead of seeing a *loop* fragment.

The timing of program interactions is also important in certain domains of applications [23]. Time needed to execute every particular message in a sequence diagram can indicate the time required to execute the method representing that message. In addition, the time spent on sending a request from an object to receiving it by another object is a useful sign of the performance of the environment under which a program is running (i.e., a single machine or a network node).

2.3.4 Concurrent and Distributed Interactions

Real program interactions may occasionally need to be enriched with information related to the actual source that triggered them. Expressly, messages communicated between objects are not always sent to or received from the same thread in the program, and objects themselves might not reside on the same machine where the program is being run. Therefore, techniques have been proposed in the literature to represent concurrent and distributed interactions essentially by visualizing the threads and machines that participated in these interactions [5, 24, 25, 26, 27].

2.3.5 Program Performance

Here, the amount of resources (i.e., CPU time, memory or storage space, etc.) consumed by every action performed in the system is the most important piece of information to be visualized and presented to users, epecially for real time systems [28, 29]. This helps to figure out the objects, messages, threads, or any other entities that exhaust system resources more than others in a certain program. UML sequence diagrams have been extended with special annotations to support the visualization of schedulability, performance, and time of a program to assist developers and users monitor performance issues in their programs [30, 31, 32].

2.3.6 Security Analysis

A less common, yet important, aspect of program comprehension using sequence diagrams is checking security properties of systems. Some techniques have been developed to reverse engineer the security properties of given programs [33, 34, 35]. These techniques usually focus their analysis on the user access roles to trace all possible steps they might follow to achieve their tasks. Sequence diagrams in this aspect are useful for deducing the permissions for each user role, understanding security vulnerabilities (e.g., SQL injection), modeling security patterns, or visualizing the security-critical interactions between system objects [33]. To support the visualization of such interactions, several security-enhanced UML extended notations have been proposed in the literature [36, 34, 37].

2.4 Existing Surveys of Program Analysis Techniques

In the literature, surveys in this context were conducted to summarize and discuss characteristics of those techniques from different perspectives. Pacione et al. [38] in 2003 introduced a comparison between the performance of dynamic software visualization techniques. In 2004, Hamou-Lhadj [39] discussed the benefits and drawbacks of trace exploration techniques and showed how they work and how they could be enhanced. Briand et al. [5] in 2006 and Cornelissen [7] in 2009 presented existing program analysis techniques in the literature that are only based on dynamic analysis. The main objective of these two reviews is to show the various research efforts, methodologies, and target applications and systems of the reviewed techniques. Another survey was introduced in 2011 by Cornelissen et al. [14] who summarized all the related techniques along with the applications in which they were validated.

In accordance with our investigation of current reviews in this context, we noticed that none of them has conducted an evaluation of the existing sequence diagram-based techniques for their effectiveness and usefulness towards program comprehension. In other words, the amount and kind of information equipped with the resulting sequence diagrams determine how understandable they are and, hence, indicate how effective these techniques are.

Therefore, we have been motivated by this to conduct a thorough study of existing program analysis techniques that contributed to improving the understandability of software interactions by means of sequence diagrams. The state-of-the-art techniques are classified herein based on the type of analysis they employ (i.e., static, dynamic, or hybrid program analysis).

2.4.1 Program Analysis Approaches

Program analysis methods that are used for the systematic retrieval of design and behavior of software systems can be either static, relying solely on the source code, or dynamic, where execution traces of the program are analyzed [40]. It is also common to combine both kinds of analysis within the same technique to be able to achieve certain program comprehension tasks that would otherwise be hard or impossible to achieve. For example, in object-oriented systems, polymorphism, and dynamic binding make it difficult for static analysis alone to anticipate runtime behavior. Furthermore, certain assumptions and filtering policies are usually made when performing program analysis for comprehension to cope with the complexity of systems and the needs of users. For example, in dynamic analysis of execution traces, it might be useful to assume that there are no unstructured control-flow constructs (e.g., 'goto') in the source code as their presence may complicate both: the analysis process and the resulting diagrams.

2.4.1.1 Static Analysis

Static analysis is a software exploration process that relies on the source code of the program in order to derive both: the structure and behavior of software systems including all interactions (i.e., method calls) between objects [41]. Static analysis is usually accomplished without executing the intended programs [42]. This kind of program analysis essentially works by parsing program source files and logging all interactions between internal system components. Starting from the main entry point of the program (e.g., *main* method), the analyzer keeps track of all object construction operations and method invocations between them, while taking into account all constructs that control the flow. For every particular interaction in the program, information obtained through the analysis are usually logged into memory as interaction traces to be used by the visualization process at a later stage.

Although static analysis is an efficient analysis method that can easily recover the structure and design of a program, it can only provide a conservative approximation of the runtime behavior of the program, which may lead to less precise behavioral representations. This limitation poses many challenges when analyzing programs that make use of dynamic features. For instance, in object-oriented languages with dynamic class loading (like in Java), the interactions between those classes cannot actually be captured or even depicted via static analysis of source code. Another example is the challenge of analyzing distributed software systems that employ multi-users or multi-threaded processes that might have parallel interactions [5]. Furthermore, users (or threads) in such systems can be added or removed at runtime, further complicating the analysis task. In addition, capturing the communication between threads or remote objects is not likely to occur in static analysis because such information can only be gathered during the execution of the program. However, using static analysis, it is sometimes possible to show an expectation of how such interactions would be executed in the program at runtime.

Nevertheless, based on [43] and the recommendation in [44], incorporating behavioral information through the static analysis of source code is essential for any technique that aims to visualize dynamic execution traces.

The study by [45] formally demonstrated the relationship conformance of differently generated sequence diagrams. Sequence diagrams of a certain artifact might be reused in an application with various changes in that specific application. In addition, the names of lifelines, messages, and system variables may be changed as well. Therefore, it is important to verify that the sequence diagrams of an application conform to the ones in the artifact, by avoiding name conflicts, and that was the main objective of the work presented by Lu and Kim [45].

Korshunova et al. [4] presented a tool that can reverse engineer class, sequence, and activity diagrams of any given C++ system by parsing its source code and then extracting its AST. The derived structural and behavioral models were represented and stored in XMI files [46], which represent UML elements in an XML format. Objects and method calls were used to construct sequence diagrams while Activity Diagrams were generated from the conditional and loop constructs.

2.4.1.2 Dynamic Analysis

Dynamic analysis of software systems is concerned with investigating their behavior at runtime [47]. This kind of analysis does not require the availability of source code, although it may make use of it when present. Indeed, having the source code facilitates injecting tracing code snippets into programs that can help in capturing all interactions taking place at runtime. This can be done by logging all required information about method calls along with their callers and callees. Alternatively, several other dynamic analysis-based techniques do not require the source code to be available. Instead, they depend on (customized) debuggers that can supply them with the required information about program behavior [21].

Although dynamic analysis can recover precise information about the actual behavior of a system, it is challenged by the problem of coverage. A run of the program explores a sample execution trace typically covering only a few behaviors out of many more possible behaviors. Increasing coverage, and hence obtaining a complete view of the program's behavior, requires multiple executions of the program with varying parameters representing different execution scenarios. Having multiple runs, however, entails the generation of multiple sequence diagrams, posing the other challenge of combining these diagrams into a single comprehensive one [5].

Another limitation of dynamic analysis is that the information it gathers always represents the series of interactions in a program without considering the control flow constructs through which these interactions passed through [48]. It, therefore, lacks the ability to capture information about whether these interactions passed through conditional alternatives (e.g., **if**, **switch**, **exception** handling, etc.), repetition loops (e.g., **for**, **while**, **do-while**, etc.), or recursive calls. Notice that such information can be easily collected using static analysis of the source code, if performed well. To allow dynamic analysis-based techniques to identify such kind of information, it is helpful to utilize the availability of source code to inject certain scripts for recognizing the changes in the program control flow. Such scripts will do their job while the source code of the program is instrumented using multiple traces [5].

Another challenge is that dynamic analysis can only work with complete software systems, rather than software fragments or components. This means that dynamic analysis of subsystems may require them to be instrumented by automatically employing program stubs (e.g. a specially crafted main class) to enable the collection of sample executions of certain program components [49].

In [50], a concise and clear representation of program execution was proposed by summarizing sequence diagrams using state diagrams. In that study, it was stated that state diagrams had never been previously used to represent program execution. However, user participation in the summarization process is required in that technique. The process of summarization could be performed after executing the program multiple times.

Briand et al. [5] introduced a dynamic analysis technique of program execution that works by creating several aspect-oriented snippets (in AspectJ) that can capture object creation and method calls, and then log all interactions into appropriate trace files. They addressed the issue of distributed systems, and how message senders and receivers can be identified when objects are executed at different network nodes. Their resulting sequence diagrams were partial in a way that each individual trace has its own sequence diagram, which concludes that a single combined diagram is not produced.

Oechsle et al. [51] targeted students by proposing JAVAVIS to help them understand the interactions executed at runtime of small-sized programs. They represented each active method by an object diagram, whereas the whole program was represented using a single sequence diagram. Their approach was well presented as a visual debugger providing a graphical user interface through which a user can run the program step-by-step and visually observe what is happening, control the speed of the animations, and select the classes required to be displayed. Each loop iteration is represented as a separate block of messages. Calls to system classes' methods are filtered out and not shown in the sequence diagram.

In [26], a monitoring tool called Kieker was introduced to continuously (or on-demand) observe the behavior of Java programs. Their instrumentation strategy was based on AspectJ as well. Their main focus was on Web applications in which the behavior differs from one user to another. The purpose of their approach was to minimize the monitoring overhead as much as possible. Performance analysis and evaluation of the approach were demonstrated in [52]. Their experimental evaluation showed that Kieker could achieve a smaller linear overhead compared with others, which makes it potentially appropriate in industrial settings.

The goal of the technique proposed in [53] was to reduce sequence diagram size by removing the less important details and methods from them, especially, local objects. This technique applies the dominance algorithm on the dynamic call graphs to compute the dominance relation between objects. Based on their previous tool (called Amida [54]), they proposed two different but complementary tools: the first one performs the dynamic analysis of the program execution while the other statically analyzes the produced trace file. Their experiments showed that their approach could remove 40% of the objects from execution traces. However, the elimination of such objects reduces the precision of the resulting diagram in that it may miss important details about the program behavior.

Ziadi et al. [55] introduced a technique that performs dynamic analysis of

Java programs without the need for source code; i.e., only the Java bytecode was needed. Each trace of program execution was represented as a Labeled Transition System (LTS), a variant of classical finite automata. All labeled transition systems related to one program are then merged using the k-tail algorithm [48].

Sequence diagrams are then identified as regular expressions to eventually generate sequence diagrams that conform to all traces. A step-by-step description of the approach was presented. While conditional alternatives and iterations are hard to recognize via dynamic analysis, this approach was able to detect them while merging the different traces. They could achieve that by using the k-tail algorithm that had the ability to go through the different paths in the finite states and detect all possible blocks of interactions. However, the authors stated that the k-tail algorithm can sometimes be inaccurate, and more accurate algorithms are needed to be investigated.

2.4.1.3 Hybrid Analysis

Recently, several techniques have been developed that combine the two types of analysis together: static and dynamic. Such techniques are considered to be more effective and efficient as the results produced from one analysis is complemented by the other's results [56, 57]. Hybrid techniques exploit the power of both kinds of analysis while limiting their weaknesses. However, the major challenge of this type of techniques concerned with merging and compacting the diagrams generated from the two different analysis approaches, which was also taken into account in a recent study [58].

From a different perspective, hybrid techniques are considered to be time-consuming as the instrumentation overhead increases with the implementation of the two types of analysis [56]. In addition, the multiple sequence diagrams generated by dynamic analysis and the one generated by the static analysis all need to be merged into a single comprehensive diagram to represent the entire program behavior [43].

Labiche et al. [56] presented a reverse engineering technique that combines both static and dynamic analyses. Their main objective was to reduce the instrumentation overhead required by the dynamic analysis, by collecting only a small amount of runtime information that cannot be derived from static analysis, like threads. Other information can be obtained via static analysis, which in turn collects the control flow information to eventually generate separate UML scenario diagrams per each trace. This technique, however, does not produce a complete sequence diagram that combines the different generated sequence diagrams.

Myers et al. [43] introduced a technique that uses both static and dynamic analyses to collect information about programs. Their objective was to improve the visual appearance of the generated sequence diagrams. They did so by introducing an algorithm that compacts a large amount of information of call/message interactions between system objects. Although this kind of abstraction is useful, it can adversely affect precision as many interactions will be hidden from the user.

Another trend of reverse engineering techniques focuses on web-based applications [33, 59, 60]. In web-based applications, techniques were usually used to represent web pages as objects in the resulting diagrams, while the transactions between them represent the interaction messages. Alalfi et al. [33] described how PHP2XMI, a reverse engineering tool, can be used to retrieve role permissions of web pages at the access level and represent them in sequence diagrams. They evaluated PHP2XMI at the entity level and investigated how it can be used to test other security vulnerabilities of web applications, such as SQL injection. Their approach was able to filter execution traces related to database insertions, and automatically exclude any information that might complicate the comprehension process, and eventually produced the corresponding sequence diagrams that represent the interactions between users and web pages.

2.5 Evaluation of the Approaches

In this section, we present an extensive evaluation of the state-of-the-art techniques that produce reverse-engineered sequence diagrams, and study their expressiveness, usability and usefulness for program comprehension purposes. This is accomplished by first defining a set of evaluation attributes and then projecting them onto the techniques reviewed in order to compare and contrast their strengths and limitations and to identify gaps that can potentially be filled.

2.5.1 Evaluation Attributes

A total of sixteen evaluation attributes is identified and listed in Table 2.1. The particular selection of these attributes was motivated by our intention to focus on showing how understandable, scalable and usable the diagrams produced by each of the techniques being compared. The understandability of a technique is measured by the amount of information about interactions each technique can retrieve from programs. The diversity of objectives, languages, dependent techniques and kinds of diagrams can express how usable and applicable these techniques are to a wide range of software systems. We also present the case studies used by the different techniques to assess their scalability.

Some of the attributes have newly been identified throughout our review of the related techniques, while some others, such as language, diagram, target lifelines, conditions, loops, tool support, and threads have been inspired by other research works [5, 7] . Table 2.1 summarizes these attributes with some remarks that indicate their impact on the evaluation. A subset of these attributes is not actually helpful in evaluating the techniques towards program comprehension, such as the language, kind of diagram, and dependent tools. Such attributes can help users to select the appropriate technique based on their needs. On the other hand, the remaining attributes have a direct impact on the evaluation of the effectiveness of techniques in understanding programs. Some attributes have been chosen to represent a set or group of attributes. For instance, the 'loop identification' attribute refers to the ability of the technique in gathering information about the different kinds of iterative constructs, such as **for**, **while**, **do-while**. What made us go for such grouping is that some techniques were restricted to parse only a limited number of forms of each category. This is due to the fact that having the ability to gather one form of loop constructs, for example, indicates the possibility of applying it to other forms as well. Therefore, techniques that can capture at least one form of a certain attribute are considered to be satisfying that attribute.

Table 2.1: Evaluation attributes

	Attribute	Description
1	Dependent technique(s)	Lists all supporting tools or techniques employed by a given technique.
2	Major Objective(s)	Describes the main objective of the technique on which the authors concentrated.
3	Case Study	Describes whether the technique has been empirically tested and validated on real projects.
- -	- - - - - - - - - - - - - - -	- -
4	Condition Identification	Indicates the ability of the technique in capturing alternatives in a program (if, switch, etc.).
5	Loop Identification	Indicates the ability of the technique in capturing iterative loops (for, while, etc.).
6	Recursive calls	Indicates the ability of the technique in capturing recursive calls.
7	Threads Identification	Indicates the ability of the technique in identifying communicating threads.
8	Multi-users Identification	Indicates the ability of the technique in identifying communicating users.
9	Compaction	Indicates whether the technique filters the messages to summarize the resulting diagram.
10	Multiple scenarios	Indicates whether a technique performs the instrumentation with different scenarios.
11	Merging Diagrams	Indicates whether the technique combines scenario diagrams into a single sequence diagram.
12	Tool Support	Indicates whether the technique has an available tool or it is just a prototype.
13	Analysis	Indicates whether the technique has a static, dynamic or hybrid analysis.
14	Language(s)	Identifies the type of programming language on which the technique works.
15	Target lifeline(s)	Refers to the main entities that interact with each other in a given software system.
16	Diagram(s)	Lists all possible output visualization diagrams used by the technique.

Table 2.2: Major characteristics of the reviewed program analysis techniques

Ref.	Dependent technique(s)	Major Objective(s)	Case Study
[45]	Unknown	Conformance of different SDs	JHotDraw and Monetary Access Control
[4]	Columbus/CAN, DOT	Coordinating objects and messages	30 KLoCs, 60 KLoCs projects
[5]	Unknown	Distributed systems	A library system
[50]	JIVE	Deriving SCD from SD	Dining PhilosopherâĂŹs problem.
[51]	JDI	Learning programming	Small-sized programs
[26]	UMLGraph, R, GNU, Plotutils, Graphviz.	Reduce time overhead + multi-user web apps.	iBATIS JPetStore, SPECjAppServer2004
[61]	Their tool, Amida [54]	Compacting SDs	jEdit, Gemini, Scheduler, LogCompactor
[53]	Their tool, Amida [54]	Compacting SDs	An enterprise web application
[62]	Oberon [63]	Compacting SDs + User-Interaction	Kepler + A Compiler Construction Framework
[27]	JVM Profiler Interface	A framework for distributed systems analysis	A Technical Report System (TRS)
[64]	MAS [65], ATL [66]	Understanding API usage	Not mentioned
[67]	Rigi	Overlapping info of static and dynamic views	FUJABA project
[21]	JExtractor, Rigi, SCED	Deriving SCD from SD	FUJABA project
[56]	Their work in [5]	Reduce time overhead	5 large systems and 2 small programs.
[43]	Unknown	Compacting SDs	Eclipse IDE, HSQLDB, Jetty web server platform
[55]	K-tail algorithm [48]	Merging SDs	A project with 500+ classes and interfaces, with 25000 lines of code.

2.5.2 Analysis and Discussion

This section presents a detailed comparative evaluation of the most relevant techniques to our study based on the attributes introduced in the previous section.

Table 2.3: Evaluating the program analysis techniques

Ref.	Condition Identification	Loop Identification	Recursive calls	Threads Identification	Multi-users Identification	Compaction	Multiple scenarios	Merging Diagrams	Tool Support	Analysis	Language(s)	Target Lifelines	Diagram(s)
[45]	✓	✓	·	·	·	·	·	✓	✓	Static	Java & Prolog	object	SD
[4]	✓	✓	·	·	·	·	·	·	✓	Static	C++	object	SD, CD & AD
[5]	✓	✓	·	·	✓	·	✓	·	·	Dynamic	Java	object	SD
[50]	·	·	·	·	·	✓	✓	✓	·	Dynamic	Java	object	SD & SCD
[51]	·	·	·	·	·	·	·	✓	✓	Dynamic	Java	object	SD & OD
[26]	·	·	·	·	✓	·	✓	✓	✓	Dynamic	Java	class	SD, MC, CDG & TD
[61]	·	✓	✓	·	·	✓	✓	✓	✓	Dynamic	Java	object	SD
[53]	·	✓	·	·	·	✓	✓	✓	✓	Dynamic	Java	object	SD
[55]	✓	✓	·	·	·	·	✓	✓	·	Dynamic	Java	object	SD
[62]	·	·	·	·	·	✓	✓	·	✓	Dynamic	Oberon	object	SD
[27]	·	·	·	✓	✓	✓	✓	·	✓	Dynamic	Java	object	SD
[64]	✓	✓	·	·	·	·	✓	✓	·	Dynamic	Java	class	SD & SMD
[67]	·	·	·	·	·	✓	✓	✓	✓	Static & Dynamic	Java	class	SD & SCD
[21]	·	✓	·	·	✓	✓	✓	✓	·	Static & Dynamic	Java	class	SD & SCD
[56]	✓	✓	·	·	·	·	✓	·	·	Static & Dynamic	Java	object	SD
[43]	✓	✓	·	✓	·	✓	✓	✓	·	Static & Dynamic	Java	object	SD

SD: Sequence Diagram, AD: Activity Diagram, MC: Markov Chains, TD: Timing Diagram, SCD: State Chart Diagram, CD: Class Diagram, OD: Object Diagram, CDG: Component Dependency Graphs, SMD: State Machine Diagram.

Table 2.2 and Table 2.3 show the major characteristics of each technique and the application of that technique to each particular attribute, respectively.

A direct rendering of program execution behaviors into a sequence diagram may result in an awkward and hard-to-comprehend output. To avoid such a problem, a program analysis technique has to be smart enough to identify control flow patterns including alternatives, repetitions, threading, or even recursive calls. Several techniques exist in the literature that tackles these issues using either static information from source code or helper algorithms to compact messages that relate to a loop or a condition into a single fragment. Other techniques rely on merging such messages in other kinds of diagrams rather than sequence diagrams (e.g., State Chart diagrams or Markov Chains).

From Tables 2.2 and 2.3, it can be observed that each technique intends to capture behaviors that were not addressed by others while missing some important ones. Some of them introduced similar approaches with previous ones, but with different objectives like reducing time overhead, or compaction of the collected information. Time overhead is an important issue that has to be taken into account when building any software tool. However, the ultimate goal of the users of program analysis techniques is the understanding of programs in an expressive way rather than going through source code.

AS sequence diagrams have been introduced as a UML standard, the are basically delivered in the system's specification documents to guide programmers, in the design phase of forward engineering, about what would happen in the software implementation. Almost all techniques try to adapt sequence diagrams to demonstrate the interaction between objects (or just classes), but in a static fashion. In other words, users need to have more interactive views, like [68], that exactly reflect what is going on in the program execution. For example, the programmer may need to see the time taken by each message to be completely executed, or when exactly one message comes after another. Such information cannot be represented in the standard UML sequence diagram, as it only shows the order of messages.

Another issue is related to the kind of entities to be represented in the resulting lifelines of sequence diagrams. Having objects only may lead to ignoring calls to static methods of certain classes. Similarly, using only classes as lifelines without their constructed objects may hide information about the interactions executed between objects within the same class. Therefore, both classes and objects should appear in the produced sequence diagrams in order to provide a thorough representation of all interactions of a certain software system. From the table given, we can observe that the state-of-the-art techniques differ in this matter. The majority of them use objects as lifelines while a few of them represent the class-level interactions.

It can also be noticed that even though dynamic analysis is useful in identifying the multi-threaded interactions, most of dynamic analysis-based techniques in the literature did not include such information in their analysis, and they could not identify the interactions executed between those threads. In other words, to better recognize the interactions between different threads, the resulting sequence diagram should display, for example, that an object O_1 from a thread T_1 is communicating with another object O_2 from a different thread T_2. Producing more effective diagrams may require developing new reverse engineering techniques (or even extending existing ones) program comprehension and reverse engineering techniques to be able to capture such information during the analysis. In addition, to visualize such information, one also needs to extend an existing visualizing tool to incorporate the new types of information in the existing notation of the UML standard of sequence diagrams.

It is clear that the state-of-the-art techniques do not usually focus on the amount of information that should be involved in the resulting diagrams to effectively understand a program. Adequate information in the resulting view will assist users to comprehend programs without having to go through the source code all over again. This will increase the users' productivity and help them conduct their maintenance tasks effectively.

CHAPTER 3

PROPOSED APPROACH

To effectively accomplish our objectives, we have proposed our approach in a form of extensions to existing techniques and standards introduced in the literature. These extensions are demonstrated in our work in three different aspects, namely, UML sequence diagram extensions, Polyglot extensions, and tool features extensions.

3.1 Extensions to UML Sequence Diagrams

Program comprehension is the activity of understanding the static and/or dynamic aspects of computer programs, namely the structure and behavior. Program visualization tools play a vital role in this regards. The goal of program comprehension-supporting methodologies is to facilitate understanding of computer programs effectively through cognitively understandable views. This can be achieved by visualizing the several aspects of the program in a way that gives the user an overall outlook of the program structure or behavior. This by itself requires enriching the produced views with as much information as possible so that the user will not have to go to the source code to understand the program. In addition, such techniques should not distract users with too many forms of visualizations, but at the same time, they should not use a single visualization with too many elements.

Sequence Diagrams (SDs) are very useful in representing the behavioral aspects of computer programs, including the control-flow and the communication between objects. Indeed, SDs provide a set of visual elements or notations that can help users to understand what is happening or what would happen in the program execution. These elements have to be used in an appropriate way so that they do not disperse user attention.

Although UML sequence diagrams are considered the standard views for representing the behavior of programs, they are restricted to a set of limited notational elements. Such notations are useful for representing some of program interaction scenarios, but many other scenarios cannot be shown using this restricted notations. This, in fact, reduces the effectiveness of comprehending programs since much information are either hidden from the user or represented in a different fashion that does not reflect the actual representation given in the source code, which ends with a wrong comprehension of the actual program behavior.

We intend in this thesis to extend sequence diagrams with appropriate notations that assist in reflecting the actual behavior of programs. These notations would make SDs more expressive in representing program interactions and control-flow. We firstly provide a simple extension that can distinguish all interactions related to static initialization or implemented in static blocks for other interactions. Another extension is concerned with showing variables and the place in which their variables are changed. Lifelines in the standard sequence diagram are represented in a general way so that all kinds of objects and classes appear with the same shape. We intend to extend the lifeline notation of the UML sequence diagram to make classes and objects of a certain category appear with a notation different from other classes or objects. In addition, objects that are constructed using the *new* keyword are represented using a specific notation in the UML sequence diagram. Similarly, creating lifelines after type casting operations is not obvious in the UML SD. To this end, we provide a new notation that explicitly represent type casting operations. However, objects that returned from called methods do not have a specialized representation. Therefore, we provide an extension that is capable of representing such created lifelines. The next set of extensions are concerned with the kinds of method calls a program may contain, namely nested calls, chained calls, and recursive calls. Moreover, we provide an extension to sequence diagram fragments. This extension targets demonstrating the messages that are fired from method calls inside the fragments' operands. With respect to fragments, we also extend them to make them capable of representing $try - catch - finally$ blocks.

3.1.1 Static Initialization

In Java, the JVM starts by executing all static initializations before accessing the method of the program. This includes the static block and declarations of the static variables, if any, where the sequence of their execution depends on the order of their appearance in the source code. In case the program, represented by its main class, has a declaration of static variables, all object construction calls are executed firstly. If the source code of the constructors used for object creation is available in the same program, then all interactions performed inside their body would also be executed. The other case is with the variables that are initialized using method calls, which indeed applies the same scenario, except the message type here is different (i.e., the message kind of object creation is 'create' while in the case of method calls, it is 'asynchCall'). Regarding the static block, all interactions inside this block will also be executed. Fig. 3.1 demonstrate the representation of the static lifeline along with a representation for the static block.

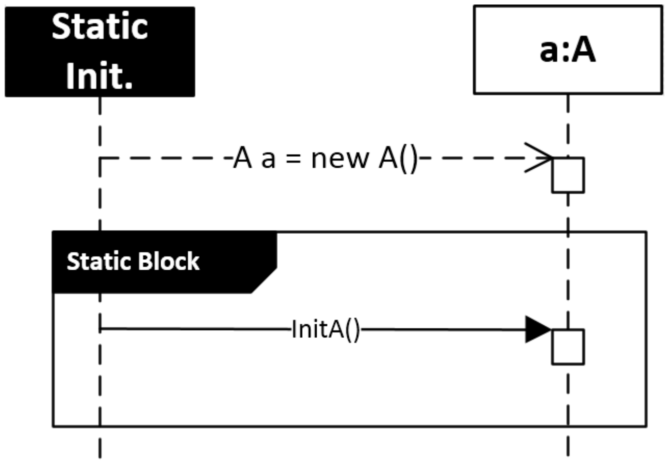


Figure 3.1: Notation representing static initialization

After tracing all static initializations, the main method of the program will be accessed. Actually, this is only the case of accessing the main class of the program. There is also another case where static initializations are executed first, which can happen when creating objects of classes that have static initialization.

3.1.2 Variable Declarations and Usage

Usually, sequence diagrams do not display variable declarations and assignments. However, they show variable usage inside the fragments and call messages. The appearance of such variables in the diagram may confuse the user as there is no information about that variable provided in the diagram, like their types or where their values are changed.

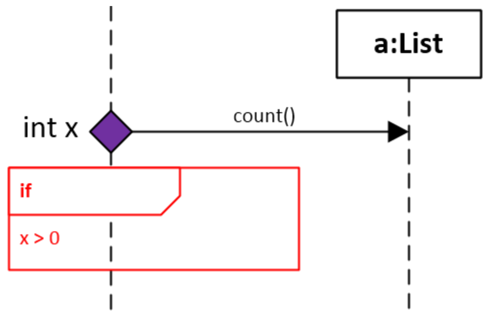


Figure 3.2: Notation representing variables

This issue could be resolved by displaying the variable declarations and assignments in the diagram in a way that users can recognize them whenever they want to know more information about them. Our extension is concerned with showing variables in the produced sequence diagram using a new notation as shown in Fig. 3.2. It is clear that there exist a declaration for the variable x of an integer type represented using the diamond. That variable is then be used in the condition of the 'if' statement that comes after the declaration.

3.1.3 Extended Notations for Lifelines

Some interactions in software systems are executed in collaboration with external entities rather than the ones that internally exist in the program. For instance, a program might interact with libraries of the programming language to use methods of reading from/writing to the system console, file system, GUI control, remote object or even another program. Programs can also call methods from the libraries of the operating systems. Although there can be a large number of lifelines in the diagram, but most of them may relate to each other in different characteristics.

Horizontally displaying a considerable number of lifelines in the sequence diagram would complicate the diagram and would decrease its readability, which may also confuse users since there might be too many of them shown in one diagram. On the other hand, hiding them would decrease the expressiveness of the diagrams since the user may sometimes need to know about them, especially if the program is relatively small.

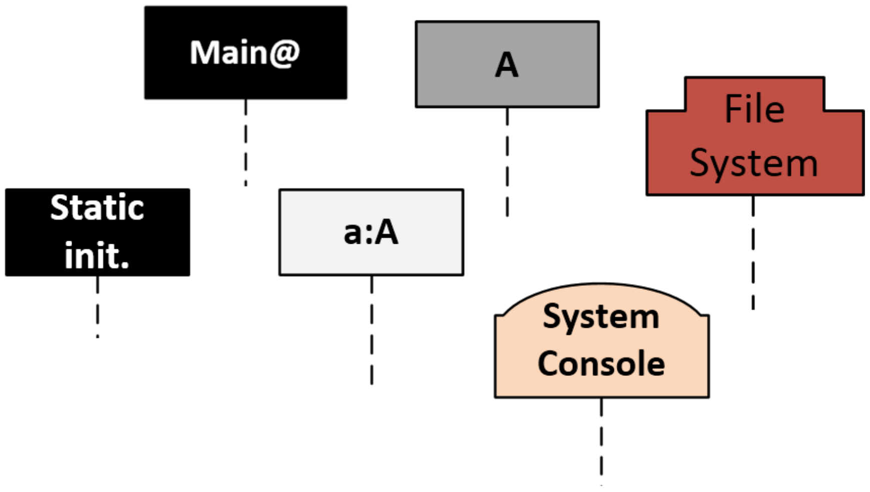


Figure 3.3: Notation distinguishing lifeline categories

In our work, we resolve this problem by proposing a new way for representing lifelines that classify lifelines into different categories, where each category is represented by a special notation that distinguishes it from the others, as shown in Fig 3.3. All classes from a certain category This can significantly help users identify external the lifelines that hold the interactions between the main objects of a program and other external entities.

The core advantage of this extension is that users can now recognize the different kinds of lifelines along with the interactions that are connected to them. In addition, users can also, with a tool-support, hide a specific lifeline category that might not interest them while showing the most important ones. For example, all lifelines that relate to the system console can be hidden while interactions with the file system can be shown.

The categories of lifelines that could appear in Java projects include, but not limited to, the following:

- Local lifelines:
  - Local object: this lifeline represents the source of messages that are sent/received by an instance of a certain class.
  - Local class: this lifeline represents the source of messages that are considered as static methods, which are usually called using the class name.
  - Main Method: this lifeline represents the source of all messages that are originated by the main method of the program.

  - Static Initialization: this lifeline represents the source of all messages that are originated during the static initialization of the program.

- Non-local lifelines:
  - Remote Object: this lifeline represents the source of all messages that are communicated between program's objects/classes and other objects that are located on a remote machine (e.g., RMI objects).
  - External Process: this lifeline represents the source of all messages that are communicated between program's objects/classes and other processes in the system (e.g., through the 'Runtime' class).
  - System Console: this lifeline represents the source of all messages that are communicated between the program and the system console (input/output).
  - GUI: this lifeline represents the source of all messages that are communicated between the program and the graphical user interface (GUI) (input/output).
  - File System: this lifeline represents the source of all messages that are communicated between the program and the file system (input/output).
  - Network Socket: this lifeline represents all classes or objects that relate to interacting Java programs with local and Internet networks, such as downloading files, opening web pages, transferring files, etc.

– Database Connectivity: this lifeline represents the classes or objects used for communicating a java program with databases, which can include updating and querying the database or dealing with its metadata.

– Other language libraries: this lifeline represents the source of other kinds of messages that are sent/received by instances or classes of libraries of Java.

3.1.4 Lifelines from Returned Objects

Object construction can be achieved in several ways in Java. The first approach to instantiate objects is with the use of the 'new' statement, which immediately calls the constructor of the intended class. This particular object creation already has a special notation in the standard sequence diagram, which represents it using a dashed message arrow. Another way of constructing objects is by calling methods that have the ability to return representative objects that can be assigned to the object needed to be constructed. Unfortunately, the UML standard of sequence diagrams does not provide a facility of showing such object construction. All what it does is representing the call to the method and its returned value using message arrows without showing the lifeline that should be created from that return operation. However, a lifeline representing the created object will be shown whenever that object is used in later stages of the sequence diagram.

We have addressed this particular issue and came up with a way that can precisely reflect the flow of the operations discussed above. Actually, our approach in this context does not introduce a new notation to the sequence diagram, but it utilizes the currently available notations in the UML standard to accomplish the goal. Representing such a flow of control in our approach is shown in Fig. 3.4 that reflects the interaction carried out in the below statement.

```
A a2 = a1.getObj();
```

It is clear now that the returned value of the method invoked caused the creation of the lifeline.

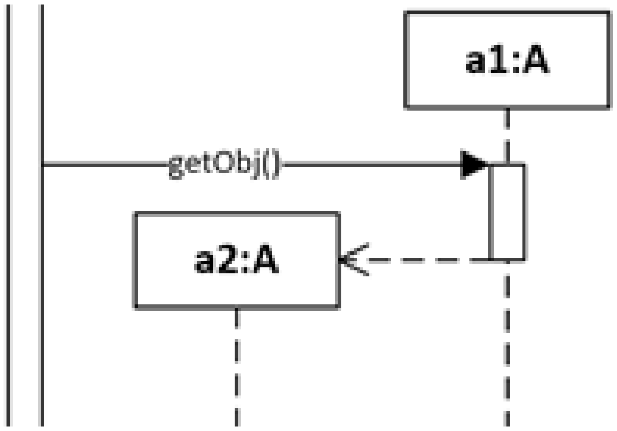


Figure 3.4: Representing lifelines created from return objects

3.1.5 Type Casting

In this extension, we just complement the previous extension with a notation that can cover all aspects of object creation. This aspect is related to creating objects of one type from objects of other types, which requires maintaining an appropriate type casting operation that can convert objects of one type to another. Representing type casting is not supported by the UML sequence diagram assuming that users will be able to implicitly assume that object type is converted from another type.

In our approach, we do address this kind of operation while generating sequence diagrams by introducing a new SD notation shown in Fig. 3.5 that represents such operations visually. We tried to make the shape of this notation as meaningful as possible by representing it as an adapter that can accept an input of a certain type and produces an output of a different type. The diagram shown in the figure is produced from the following statement:

```
A a = (A) b.getObj();
```

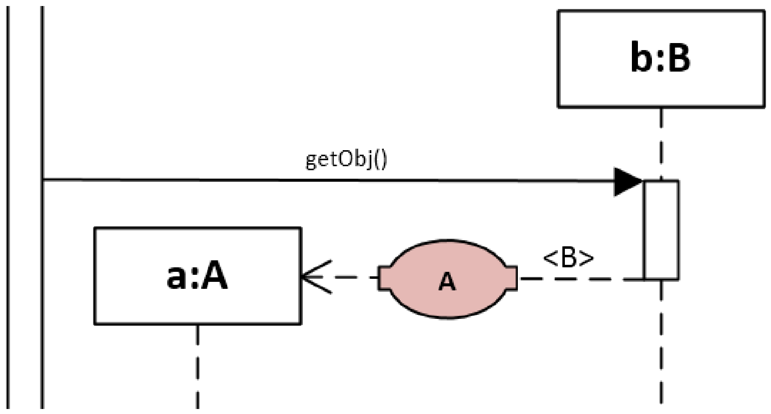


Figure 3.5: Notation representing type casting operations

3.1.6 Nested Calls (Calls Inside the Arguments of Other Calls)

Method calls may sometimes have nested calls in their parameters. Indeed, the program in such cases starts with invoking the methods inside the call parameters and after that, it invokes the main method call. Invoking the method calls that reside inside the parameters of another method call can be either in a left-to-right or right-to-left order. In our case, since we focus on Java, the left-to-right associativity is considered.

Now, let us take the following code representing two nested calls, where the call to the method 'getObj' of the object 'b' is nested with the call to the method 'setObj' of the object 'a';

```
    a.setObj(x, b.getObj());
```

In the standard sequence diagram, this set of method calls is represented in two different ways as shown in Fig. 3.6, depending on the tool that uses the UML standard of sequence diagrams. It is obvious that the first representation (Fig. 3.6a) shows the exact number of interactions, but does not precisely reflect the actual flow of control. The other representation simplified the whole statement using only one message arrow, which does not reflect the actual number of interactions.

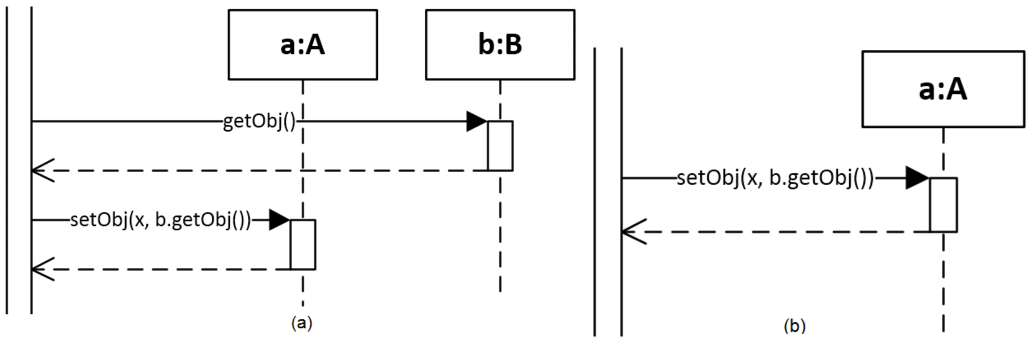


Figure 3.6: Representing nested calls in the standard SD

To overcome this kind of problems, we propose a new SD notation that can distinguish this kind of method calls from other kinds of calls. This notation is shaped like a box entitled with the main method call, and all other calls that are to be executed are shown as messages inside that box, as shown in Fig. 3.7.

This notation reflects the actual flow of control with the exact number of interactions. As shown in the figure, the method 'setObject' is firstly called but lastly

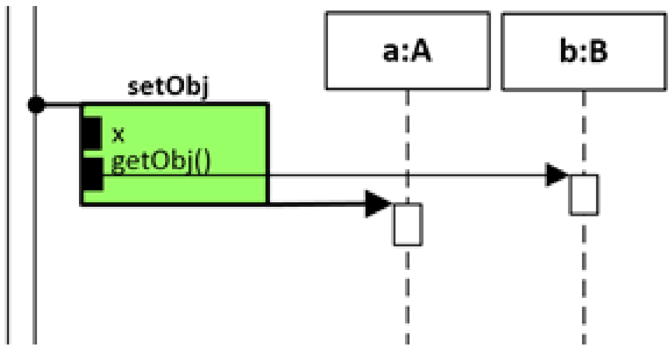


Figure 3.7: Notation representing nested calls

executed. Also, the parameter x does represent an interaction while the second parameter, the method ‘getObj’, represents a nested interaction that is executed inside and before its enclosing method. This methodology is also applicable to the interactions of the type ‘return’ and ‘new’ in case they involve method calls within their parameters.

3.1.7 Chained Calls

It is common in object-oriented programming to see what is called chained calls. Chained calls are the set of calls that depend on each other in their execution. This means that the object needed to call one method is returned by its preceding method call. As an example, we provide the following chain of calls:

```
a.getString().indexOf("a");
```

We can see that the method ‘getString’ of the object ‘a’ will be called and will return an anonymous object of type ‘String’, which will be used to call the method ‘indexOf’ with a string parameter. The standard sequence diagram deals with such calls separately, which means that the diagram will show that these two calls are independent and do not depend on each other (as shown in Fig. 3.8).

However, it shows the exact number of interactions, but take in mind that some tools represent the whole statement using one message while some others only represent the first method call of the chain.

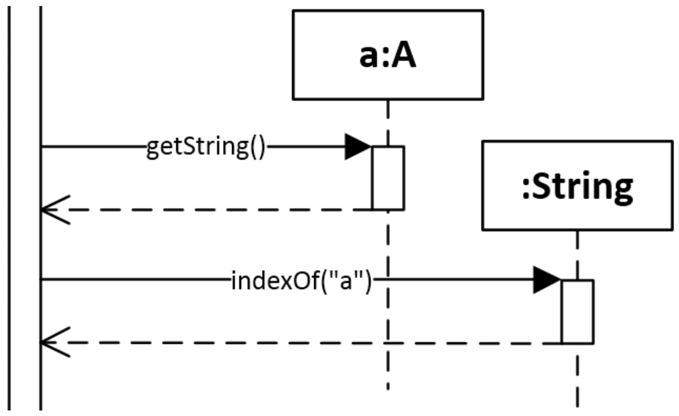


Figure 3.8: Representing chained calls in the standard SD

We have targeted this case by providing a new notation that can precisely represent the chain of the calls and the dependence of one call on another. As demonstrated in Fig. 3.9, the notation is intuitively expressive and reflects the actual flow of control and the exact number of interactions.

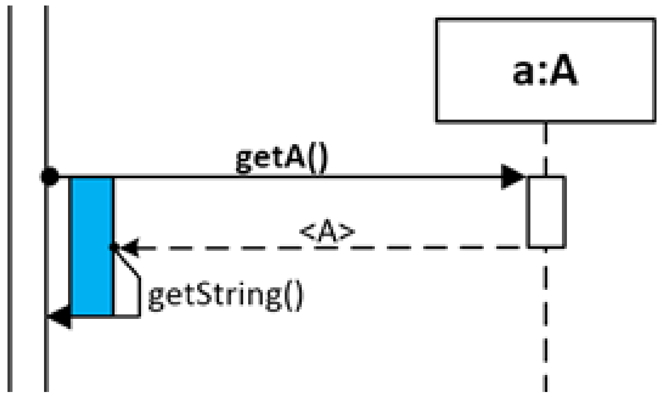


Figure 3.9: Notation representing type chained calls

3.1.8 Fragment's Operands with Calls

Sometimes, it happens that the specifications of a certain fragment contain a method call or a set of method calls. We refer by specifications to all guarding operands of fragments. For instance, the specification of the 'loop' fragment is represented by the initialization, condition, and iteration sections of the loop guard. Likewise, alternative and optional fragments, such as if, switch, and conditional operator in Java, have their own guard specification represented by the associated condition. Now, if such a specification contains a method call, then this method call will neither be represented in the standard XMI nor visualized in the standard layout.

For example, if we have the following 'if' construct in Java:

```
if(a.hasObj()) {
  // code
}
```

Inside the operands of this condition, there is a call to the method 'hasObj' of the object 'a'. The standard sequence diagram does not care about whether the condition is based on a call or not. It represents the condition as text in the specification part of the fragment as shown in Fig. 3.10.

On the other hand, our extended notation to the sequence diagram tackles this problem by allowing calls inside the operands to be represented as messages to the corresponding lifelines as shown in Fig. 3.11.

A similar case can happen with loops, such as 'for' loops, where the operands part of the loop is composed of method calls, like the following example:

```
for(int i=0; a.hasObj(); incr(i)) { }
```

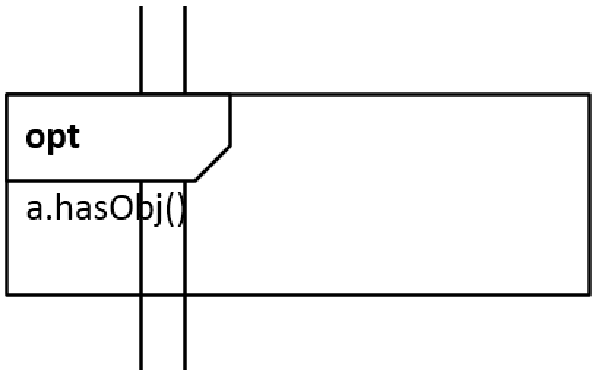


Figure 3.10: Representing conditional operands with calls in the standard SD

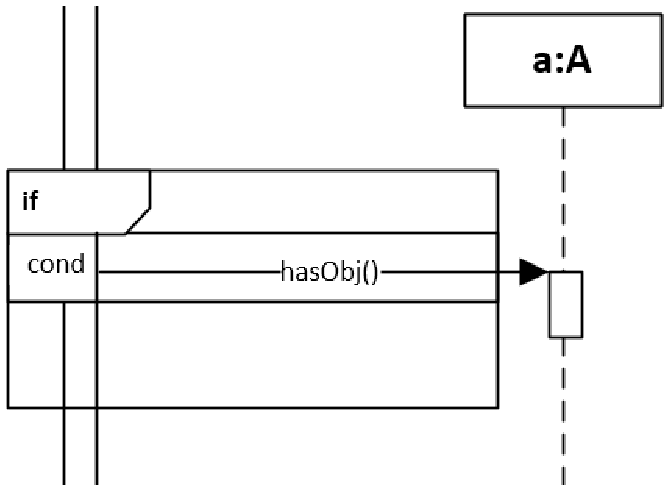


Figure 3.11: Representing conditional operands with calls using our notation

As we can see, the associated conditional and iteration statements in the above examples are based on values returned from method calls. These methods may exist in the self-class/object or others. Therefore, representing that method call as a message in the resulting SD would add more information that can help in understating what exactly is going on in the source code, as represented in Fig. 3.12.

Similar to the case of the conditional fragments, the standard sequence diagram represents what is written in the operands as text in the produced fragment, like the one presented in Fig. 3.13 for the same above example.

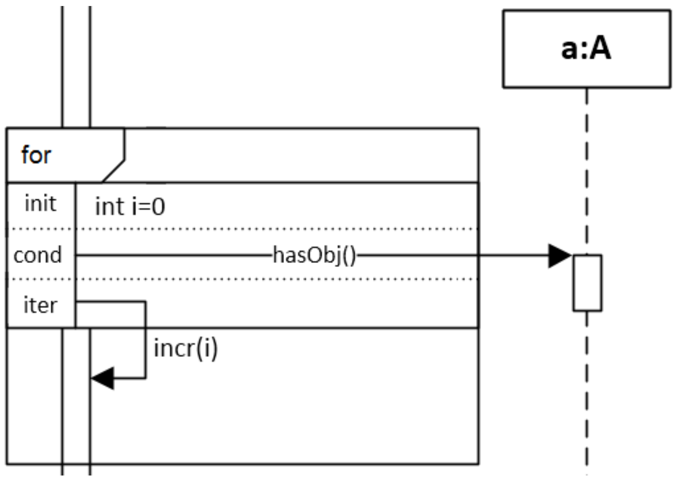


Figure 3.12: Representing loop operands with calls using our notation

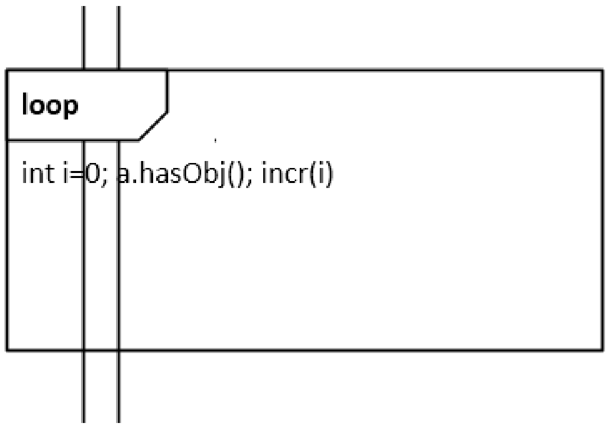


Figure 3.13: Representing loop operands with calls in the standard SD

3.1.9 Recursive Calls

Recursive calls are repetitive calls (just like loops) that make a sequence of commands executing many times. However, unlike loops, there is no specific control structure in programming languages that can indicate the presence of recursive calls. Therefore, this kind of repetitions is unlikely to be captured through the static analysis of the source code. On the other hand, dynamic analysis techniques can detect this kind of repetition but without knowing whether they are recursive calls or normal loops.

One problem behind the inability of static analysis to distinguish recursive calls is that the methods may not only be called from themselves, but also from other methods that are already called by the original method. For example, if method1 calls itself, then a recursive call happens. Also, if method1 calls method2, and then method2 calls method1; then this is also considered as recursion as well.

This issue can simply be resolved by tracking the calls through a stack structure. Whenever a call to a certain method happens, its signature is pushed into a stack. Whenever the end of such method is reached, it is popped. The indication of a recursive call can only occur if a call happens to a method that already exists in the stack. Let's take the case of the following example:

```
void fact(int n){
    fact(n-1);
}
main(){
    fact(5);
}
```

As we can see, the method 'fact' is recursive as it calls itself (assume that we are inside the object 'a'). In the standard representation shown in Fig. 3.14a, it is represented as a self-message, while not all self-messages are recursive since they represent calling methods of the same lifeline. On the other hand, our new notation shown in Fig. 3.14b could expressively represent the recursive call so that users can identify it from the first sight.

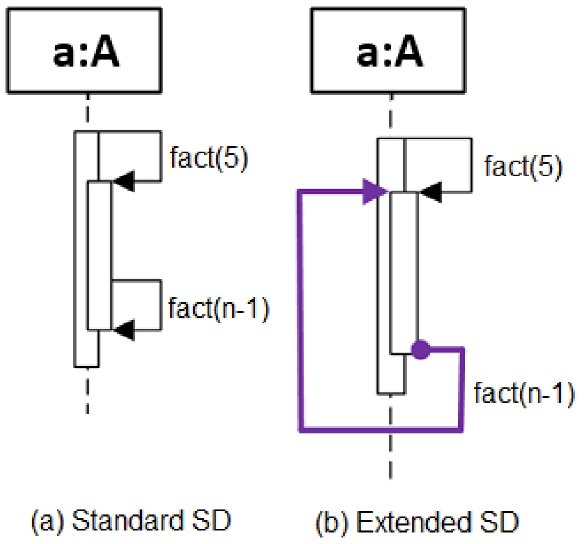


Figure 3.14: Representing recursive calls using our proposed notation

3.1.10 Documentation Comments (Javadoc)

It can happen that a program contains some classes or methods with meaningless names. For example, a class with the name $c1$ or a method with the name $m1$. However, such classes or methods can have Javadoc comments that describe what do they do. Actually, the UML sequence diagram will be able to show such classes and methods with their own names without any further description about their purpose (i.e., Javadoc comments are not shown), which would require users to go to the source code or the system documentation (i.e., Javadoc) seeking the description of each particular class/method. This, indeed, consumes much time and consequently reduces user's productivity.

Our extension here is focused on attaching Javadoc comments of classes or methods, if any, in a given program with the resulting sequence diagrams using a specific notation (see Fig. 3.15). Actually, we do not aim to make such notations appear throughout the diagram for two reasons. First, rendering them on the

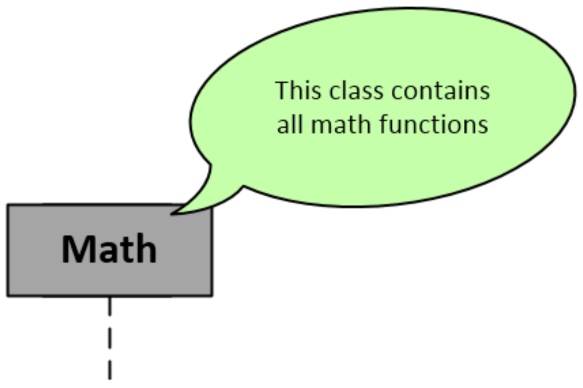


Figure 3.15: Representing recursive calls using our proposed notation

diagram can complicate it and decrease its readability. Second, users may not be interests in knowing the comments of each class or method. Therefore, our approach addresses this issue by making such comments hidden in the normal mode. Once a user hovers or right clicks the mouse on a certain lifeline or message, it corresponding comments will appear immediately on the screen.

3.1.11 Locations of Elements in Source Code

Typically, the main goal of reverse engineering is assisting users to understand and then maintain software systems. This means that users may intend to modify the source code after figuring out certain problems. Using conventional SDs enforces users go to source code and search for the specific locations intended to be modified. Our objective in this regards is to embed information about the source code location (file path and start/end line numbers) of each element of the SD. Whenever a user needs to re-engineer the software, he/she can double-click on the particular elements needed to be revised, and our SD would navigate them their locations in the source code.

3.2 The Proposed Process Model

In order to reverse engineer sequence diagrams from a given Java software system, a sequence of consecutive processes are to be accomplished. Each of these processes supplies its output as an input to the following process. This supports feeding each process with the required information to enable the integration of the entire process. The entire process model is shown in Fig. 3.16, where a Java software system is analyzed statically. Using the information collected from the static analysis, the expected system behavior is traced starting from the main entry point in it, and ending with a trace file that contains the representation of the corresponding SD. Eventually, this file is visualized using the proper SD elements (the standard and the extended).

3.2.1 Model of Gathering Program Information

Once a user specifies the software project intended to be reverse engineered, the process of information collection (shown in Fig. 3.17) starts to gather all required information from the source code of the given project. If the project consists of several source files, each file will be parsed to get the AST within which, and then each node of that AST will be visited individually.

When the parser comes across a declaration node (such as class, interface, method, or constructor declaration), it logs its signature and then checks whether it contains inner nodes inside its block or not. If yes, it keeps collecting them one-by-one, and proceed with the proper action based on the type of each node.

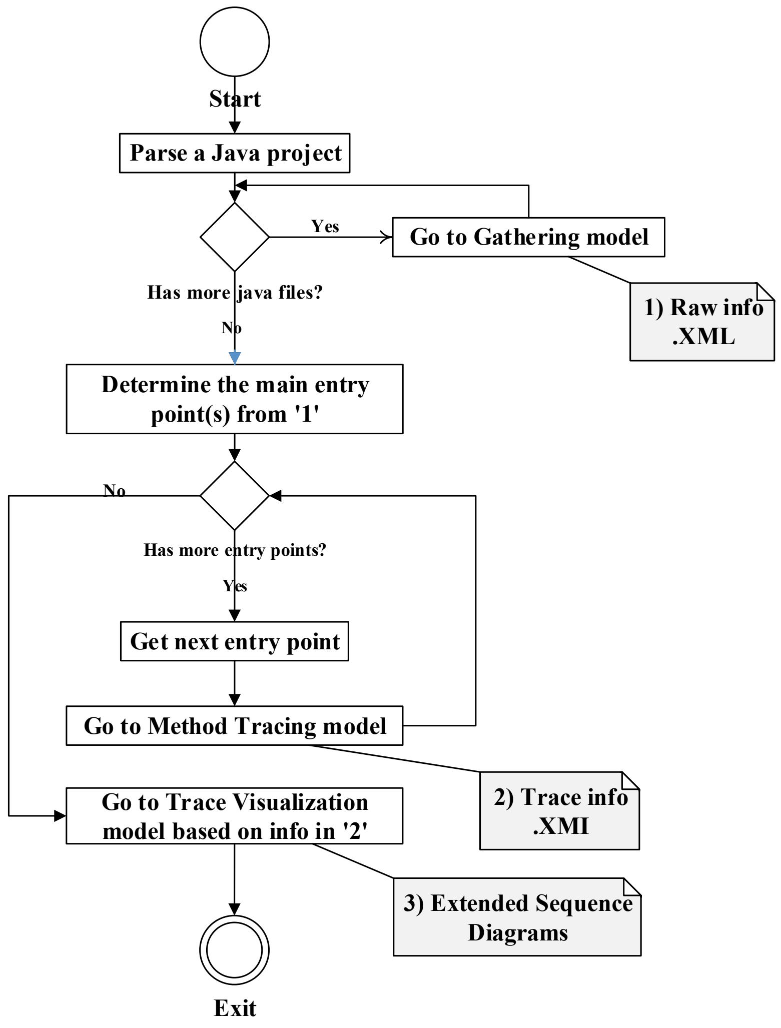


Figure 3.16: The entire process model

Whatever the type of the node is, its signature and position are logged (i.e., the file and line number where it is located). Then, other information regarding that particular node will be investigated, including its child nodes.

With respect to control structure nodes (i.e., for, $while$, $do-while$, if, $switch$, (? :) operator, $try-catch$ clause), the first reaction of the visitor is to record their start and end positions. After that, their child nodes are visited, including the block of inner statements (i.e., the body), if any, and a different action is applied to each inner statement based on its kind. Child nodes are actually different from one control structure to another. Table 3.1 demonstrates the various control structure nodes with their possible child nodes. With respect to the body of such nodes, it can either be a block of statements or a single statement. Some control structures can have either option.

Table 3.1: Types of control structure nodes and their child nodes

Control structure node	Child nodes	Block	or	Single statement
for statement	Initializer | Condition | Increment	✓		✓
while statement	Condition	✓		✓
do..while statement	Condition	✓		
if statement	Condition | Consequence | Alternative	✓		✓
switch statement	Condition | Case | Default	✓		
(? :) operator	Condition | Consequence | Alternative			✓
try − *catch* clause	Try | Catch | Finally	✓		

Method call nodes are the most important portions of the code as they represent the actual interactions between objects and classes of a project. This interaction can either be achieved locally, internally or externally. Local interactions refer to the communication of system components with each others. Internal interactions denote to links created between system objects and classes with the ones of other same-language projects or of the programming language itself. External calls are represented by the invocations of operating system libraries, remote methods, or even the methods that try to read/write to the memory, file system, command prompt, GUI, network sockets, etc. For each kind of these calls, the visitor examines whether the method call involves other calls inside its parameters or not (i.e., nested calls). If yes, it registers their signatures and types and then continues to the next node in the parent block.

Variable declarations and assignments are also important. Their importance is expressed by their occurrence in method calls. In other words, it is common to

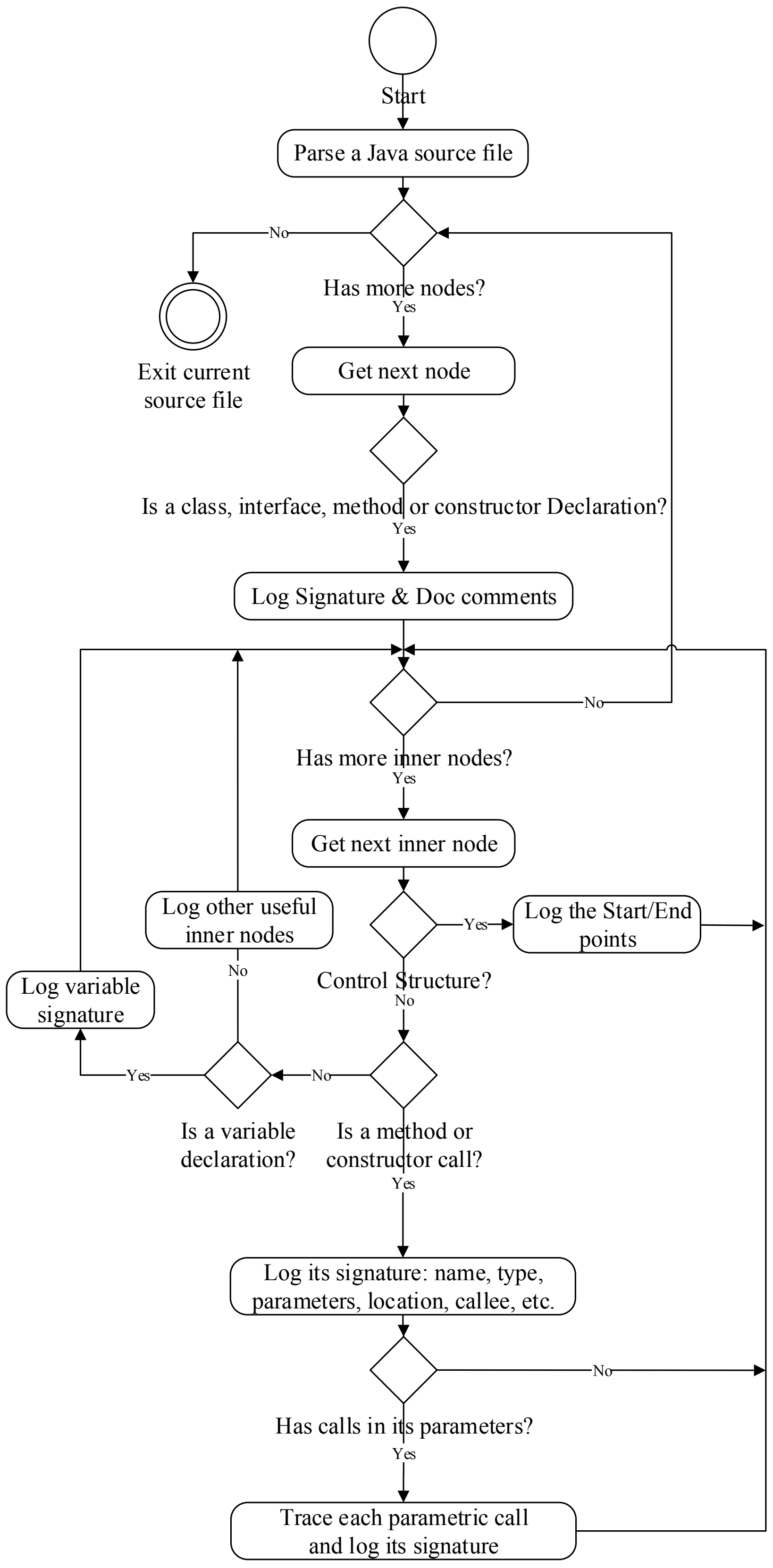

Start
Parse a Java source file
No
Has more nodes?
Yes
Exit current source file
Get next node
Is a class, interface, method or constructor Declaration?
Yes
Log Signature & Doc comments
No
Has more inner nodes?
Yes
Get next inner node
Yes
Log the Start/End points
Log other useful inner nodes
Log variable signature
No
Control Structure?
No
Yes
No
Is a variable declaration?
Is a method or constructor call?
Yes
Log its signature: name, type, parameters, location, callee, etc.
No
Has calls in its parameters?
Yes
Trace each parametric call and log its signature

Figure 3.17: Model of extracting program information

have variables in controls structures and method calls. Variables can be used in the conditions, method call parameters or as a recipient for the values returned by method calls. Therefore, having their information in the final output would help in understanding the overall sense of any element.

The information collection process keeps up gathering program nodes and transcribing their signatures in a temporary storage file. Once the visitor finds no more nodes in a certain project, it wraps up the log file to be utilized by the tracer in the immediate subsequent step.

3.2.2 Model of Tracing Program Interactions

All information gathered by the visitor are essentially required to feed the tracer, which will capture the actual behavior and the series of interactions of the given program. Once the tracer acquires the collected program information, it begins to trace method calls executed in the program.

Actually, it is not necessary to have the main method to appear at the onset of the program. This implies that the tracing process will not read the collected information in sequence. Instead, it initiates the tracing by looking for the main entry point(s) in the program.

It can happen in some programs to have multiple scenarios represented by having multiple main methods. One of the unique features of our technique is the ability to identify all entry points (i.e., main methods) of the program and produce the corresponding sequence diagrams based on the selected ones(s). This signifies that users will be notified about the multiplicity of main methods in the program (if any), and they will decide on either to trace all different scenarios of the program or choose the ones they prefer.

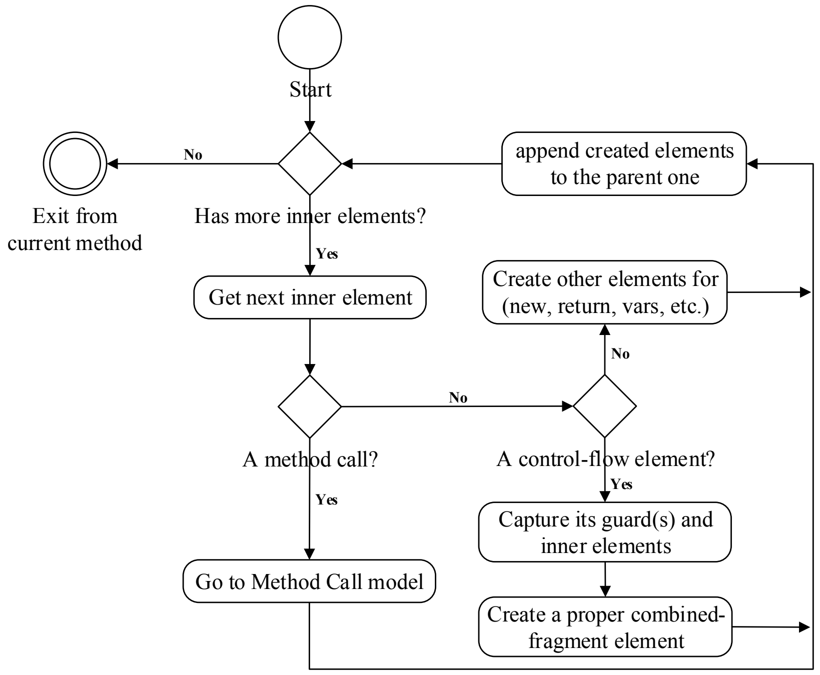

Start
No
append created elements to the parent one
Exit from current method
Has more inner elements?
Yes
Create other elements for (new, return, vars, etc.)
Get next inner element
No
No
A method call?
A control-flow element?
Yes
Yes
Capture its guard(s) and inner elements
Go to Method Call model
Create a proper combined-fragment element

Figure 3.18: Model of tracing interactions of program

Another unique feature of our proposed technique is that the tracer is able to trace method calls even if the program cannot be executed due to the presence of certain type of errors. For instance, if the program has an invocation to a non-defined or out of scope method (say `m1`), then the compiler will produce an error message saying "error: cannot find symbol `m1()`". However, our program collector is still able to gather program information and, subsequently, the tracer can trace method calls and visually notify the user about the errors.

As shown in Fig. 3.18, each element in that method is taken and logged into an XMI representation of SD. Whenever a control structure element is found, the tracer logs a fragment element with the proper signature and register the start and end of the control block. Other elements that do not require processing are directly logged with their suitable element types.

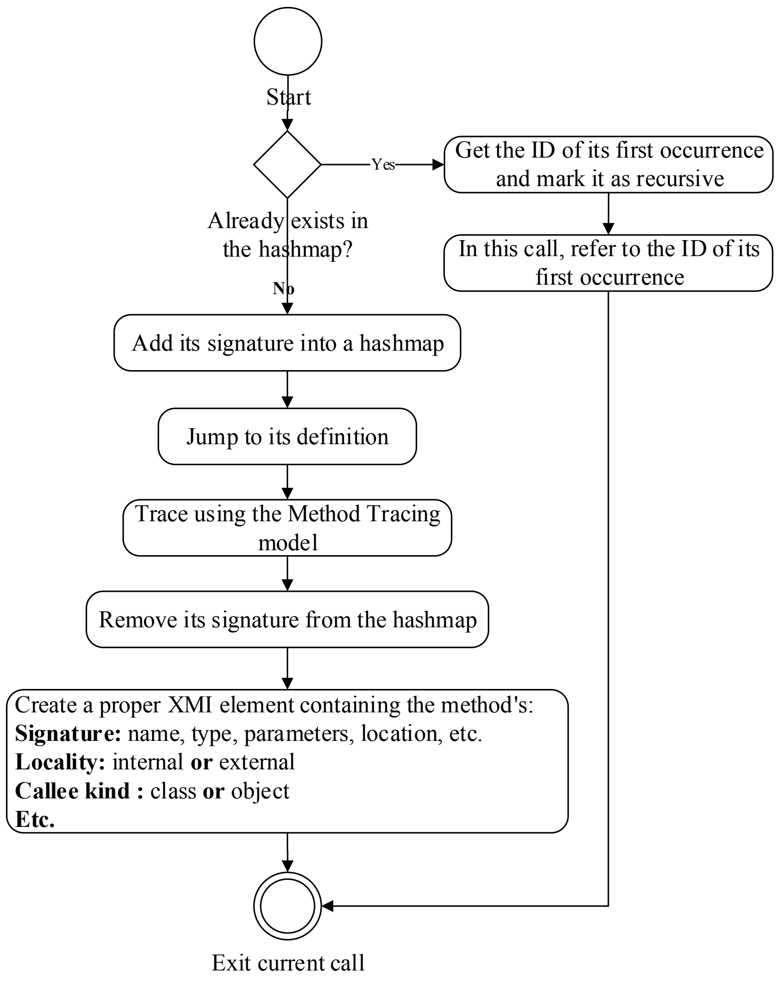

Start
Already exists in the hashmap?
Yes
Get the ID of its first occurrence and mark it as recursive
In this call, refer to the ID of its first occurrence
No
Add its signature into a hashmap
Jump to its definition
Trace using the Method Tracing model
Remove its signature from the hashmap
Create a proper XMI element containing the method's:
Signature: name, type, parameters, location, etc.
Locality: internal or external
Callee kind : class or object
Etc.
Exit current call

Figure 3.19: Model of tracing a certain method call

Whenever a method call element appears, it is required to determine the availability of the called method and its object to locate them in the raw program information. After locating the anticipated method definition, the tracer jumps to it and starts to process the method call model shown in Fig. 3.19. In the calling process model, the tracer pushes the method signature into a stack to allow the identification of recursive calls. Just before pushing, once the tracer finds out that the method signature is already in the stack, then that method is marked as recursive. After that, the tracer repeats the same process done in the main method until it reaches the end of the method body. At that stage, the method signature is popped from the stack and logged with its proper tag.

3.2.3 Model of Visualizing Program Scenarios

The visualizing model is demonstrated in Fig. 3.20. In this model, all elements generated by the tracer are visualized using their appropriate visual elements (i.e., notations). The process here starts with looking at the trace file generated by the previous process (i.e., program interactions tracer). If the program contains static initializations, then all interactions to be executed in the initialization will be originated from special kind of lifelines that is specifically used for such kinds of interactions. If not, the main method will be considered as the starting point of that program. In case there exist different main methods distributed in different classes, then that program is considered to have various entry points. Thus, the visualizer will take each entry point as a separate scenario and produce different diagrams representing the different scenarios.

Each tag in the trace file has its own attributes. Therefore, the visualizer fetches all the important information about each element from its attributes, sub-elements, or even from other elements that are referred to by this particular element.

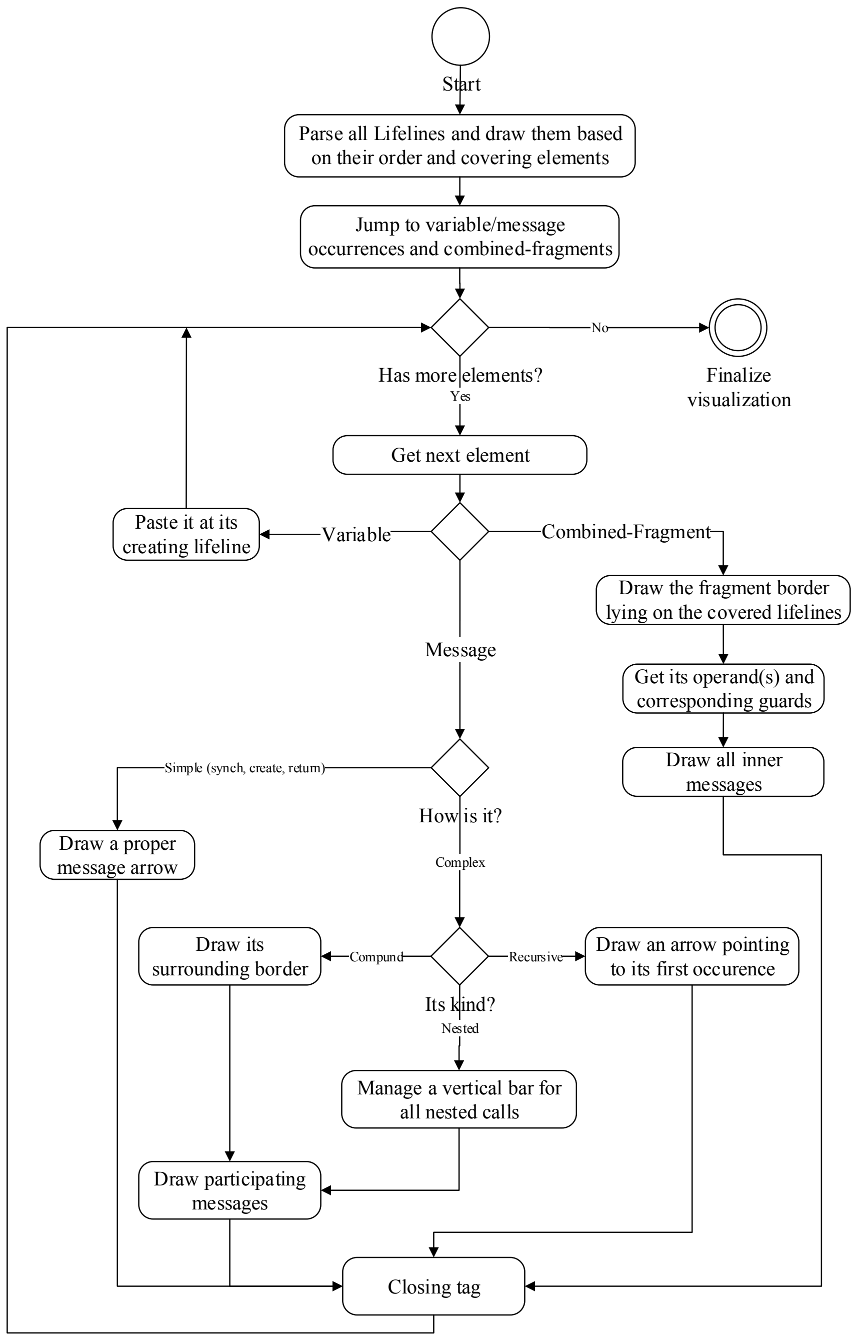

Start
Parse all Lifelines and draw them based on their order and covering elements
Jump to variable/message occurrences and combined-fragments
No
Has more elements?
Finalize visualization
Yes
Get next element
Paste it at its creating lifeline
Variable
Combined-Fragment
Draw the fragment border lying on the covered lifelines
Message
Get its operand(s) and corresponding guards
Simple (synch, create, return)
Draw all inner messages
How is it?
Draw a proper message arrow
Complex
Draw its surrounding border
Compund
Recursive
Draw an arrow pointing to its first occurence
Its kind?
Nested
Manage a vertical bar for all nested calls
Draw participating messages
Closing tag

Figure 3.20: Model of visualizing program traces

CHAPTER 4

IMPLEMENTATION

In this chapter, we introduce all details related to the technical implementation of the proposed approach. This includes the environment has been used as an infrastructure to our implementation, programming languages, and frameworks. In addition, we show how each stage of our technique is implemented and present the rationale behind choosing each of them.

4.1 Selected Programming Language

In our work, we have selected the Java programming language as a target language on which our extensions would be built. Java was selected due to its popularity and the availability of various open-source extensible compilers as well as open-source projects. In addition, most of the related work in the literature used Java in their implementations, which helps in conducting fair comparisons between them and our approach.

The version of Java grammar that is by default used by Polyglot is 1.4. However, recent versions (1.5, 1.6, and 1.7) of Java grammar have been extended to Polyglot so that developers can use the version they prefer while making extensions. In our work, we have just conducted our work with the default grammar used by Polyglot, that is 1.4, and linking it with other versions would not be a problem. What motivated us to do so is that extending the base compiler that is linked to older versions of grammar will make it compatible with recent versions of the grammar, but is not the case if we accomplished it using a recent version. In other words, extending the compiler based on newer versions of grammar will not make it compatible with older releases of Java grammar.

4.2 Polyglot, an Extensible Compiler for Java

Polyglot [6] is an open-source compiler framework for Java that enables developers to extend the Java compiler in a modular way (i.e., without touching the base compiler). In addition, it facilitates source code analysis, debugging, and constructing new domain-specific programming languages.

Polyglot has the capability of parsing Java source programs and generating the Abstract Syntax Tree (AST) by utilizing the Visitor design pattern [69]. Visitor design pattern allows capturing of nodes and their children easily, along with the ability to revisit the newly created nodes. This mechanism can duly be adapted to accomplish a static program analysis for the sake of reverse engineering of program interactions. Information on class declarations, object creations, methods calls, as well as the control flow of programs can easily be captured and logged in appropriate formats, which can then be used for tracing all call sequences in order to construct the sequence diagrams that reflect the corresponding Java program.

Nevertheless, some of Java constructs are abstracted in Polyglot, which leads to the anonymous visiting of child nodes. This means that, in some cases, the visitor cannot expose which calls relate to which sub-nodes. For example, if the visitor finds an `if` node, it visits its child nodes in sequence, including the condition, consequent and alternative nodes. However, it cannot identify what interactions are executed in each child node. Therefore, it is necessary to properly extend Polyglot to allow gathering as much useful information about programs as possible.

4.3 Extending Polyglot to Extract Program Information

Polyglot was not capable of capturing the *Javadoc* comments (or any other comments) because they are basically ignored by the parser that Polyglot is based on. Therefore, we tended to extend Polyglot and its parser's rules to enable them to accomplish such an activity. However, after communicating Polyglot community, we could get an extended version of it that was already developed by another person. Recently, in August 2015, Polyglot has been released with the 2.7.0 version, and that version covers this particular feature.

4.3.1 Extending the Compiler Node Factory

In Polyglot, some of the AST, epecially the child, nodes are anonymous. This means that they are visited as expressions or statements where the information about where exactly they are executed is missed. For example, 'If' and 'Conditional' nodes represent if statements and '? :' conditional operators, respectively. Each one of these nodes contains three child nodes inside it: 'Condition', 'Then', and 'Else' parts. In the 'If' node, Then and Else parts are statement nodes of the type 'Stmt' node, while in the 'Conditional' node they are expression nodes of the type 'Expr' node. In both 'If' and 'Conditional' nodes, the Condition part is of the type 'Expr'.

When the Polyglot's visitor-pass visits these child nodes, the $NodeVisitor$ cannot determine what gets executed in each part. For instance, if the 'if' statement has a method call in the 'Condition', one in the 'Then' part, and another one in the 'Else' part, then our $NodeVisitor$-based information gatherer captures them as three consecutive 'Call' child nodes in the 'If' node. As mentioned above, this is because that 'Condition', 'Then', 'Else' parts are not marked in the AST tree.

To resolve this issue, we tried different approaches to properly extend Polyglot. In one approach, we faced a difficulty in configuring the type system, while another one forced us to modify the base compiler. We eventually continued our work with the approach that could effectively achieve the goal. Here, we only discuss two approaches that we have implemented for extending Polyglot.

4.3.1.1 First Approach

We have extended the 'Stmt' and 'Expr' nodes (represented as interfaces) to have a marker variable that determines whether a statement or expression is accessed in a 'Condition', 'Then', or 'Else' part. This marker is defined as an integer variable that can be assigned to one of the values shown in Table 4.1. Each newly created statement will initially be assigned a value of 'NONE'. This approach is helpful to mark the statements and expressions of the nodes that are composite of child nodes.

Table 4.1: Flags used to mark 'Stmt' nodes

Flag	Value	Anticipated Parent Node(s)
NONE	0	All
COND	1	If, For, While, Do, Conditional
THEN	2	If, Conditional
ELSE	3	If, Conditional
INIT	4	For
ITER	5	For
TRY_CLAUSE	5	Try
CATCH_BLOCK(S)	6	Try
FINALLY_BLOCK	7	Try

To thoroughly achieve this kind of marking for statements and expressions of the 'If', 'For', 'Conditional' and other pre-listed nodes, their constructors have also been extended to manage the initialization of such expressions and statements with the appropriate marker values. This allows the information collector to identify where other blocks of statements and expressions are accessed throughout the source code.

4.3.1.2 Second Approach

The problem of Approach1 is that we were required to modify the base Polyglot compiler. As an alternative Approach, the base compiler is not affected as everything is embedded within our own extension. In this approach, each of the nodes that may compose of certain blocks of statement is extended in our own Node Factory. In other words, the extended node factory will have an extension of each node, named as `'Ext_'+<NodeName>`, where child nodes are accessed in a systematic way. For example, the 'If' node is represented by the 'If' interface and `'If_c'` class in the base compiler. Therefore, it is extended and given the interface name of `'Ext_If'` and class name `'Ext_If_c'`; where `'Ext_If'` extends `'If'`, and `'Ext_If_c'` extends `'If_c'` and implements `'Ext_If'`. In addition, the constructor of `'Ext_If_c'` invokes its super constructor and supplies it with the needed parameters.

The 'visitChildren' method is the routine that is responsible for visiting child nodes of any composite statement. This means that visiting child nodes can be controlled through this method. By default, Polyglot's compiler visits child nodes of a statement in sequence taking into account the left-associative principle of Java. Therefore, we have overridden such a method in each of the extended nodes to allow injecting our child-node identification code before visiting every child node. For example, in the `'Ext_If_c'` node, we could identify the visits of child nodes as shown below:

```
@Override
public Node visitChildren(NodeVisitor v) {
  // Notifying the InfoGatherer pass

  // initialization statement(s) is visited here
  List<ForInit> inits = visitList(this.inits, v);
  // initialization statement(s) is terminated here
  // condition expression is visited here
  Expr cond = visitChild(this.cond, v);
  // condition expression is terminated here
  // iteration statement(s) is visited here
  List<ForUpdate> iters = visitList(this.iters, v);
  // iteration statement(s) is terminated here
  // body is visited here
  Stmt body = visitChild(this.body, v);
  // body is terminated here
  return reconstruct(this, inits, cond, iters, body);
}
```

4.3.2 Adapting the Grammar rules

Since we have already extended the AST node factory, it is necessary to modify the grammar rules, that are directly connected to all extended AST nodes, to cope with the changes that have been made. This includes replacing the non-terminals connected to the old version of the node factory with proper references to its extended version.

For instance, the grammar rule that is matched by the compiler whenever a 'for' statement occurs in the source code is represented by the non-terminal 'for_statement'. Typically, this non-terminal will immediately create a new node of 'For', which is represented by the one defined in the Node Factory of the base compiler. Hence, we have mutated this node construction to refer to our extended version of the for-statement, which is represented in our Node Factory by the node 'Ext_For'.

As an example of how the grammar was adapted, we provide the extension of the 'if' node, we customized one of the grammar rules that access this particular node as follows:

```
drop { if_then_else_statement }
extend if_then_else_statement ::=
  IF:n LPAREN expression:a RPAREN statement_no_short_if:b
  ELSE statement:c
  {: RESULT = parser.nf.ExtendedIf(parser.pos(n, c),
                             a,
                             b,
                             c);
  :}
```

4.3.3 Extending the Compiler Visits

Polyglot has many compiler visits that are prepared to accomplish the complete compilation that results in *.class* files. Each visit is responsible for a particular functionality that is separate from the other visits. To allow Polyglot to gather program info and log them into our intended format, we have extended the 'Node-Visitor' visit class in Polyglot with another visit class called 'InfoGatherer'. This compiler visit performs all our reverse engineering functionality, including the collection of elements and their signatures and logging them as an XML tree.

Our extended visit overrides several method of the 'NodeVisitor' visit, like 'begin'/'finish' and 'enter'/'leave' methods. The methods `begin` and `end` are just used to open the log file for writing and to close it after logging all needed information, respectively. The `'enter'` and `'leave'` methods are called each time the compiler passes enter a new node in the AST tree and exit from that node, respectively. Actually, all needed information of Java programs are collected in these two methods where the kind of nodes required to be gathered and logged are specified. Actually, our visits escape some of the nodes that are not important in our technique. In other words, some nodes in Polyglot can be nested with too many child nodes. Therefore, we limited our collector to specific types of nodes that we are interested in.

4.3.4 Extending the Compiler Passes

Polyglot employs a set on compiler passes that are executed one after another in order to comprehensively analyze the source code from all perspectives. In our work, we have extended the Polyglot scheduler with two co-related passes shown in Fig. 4.1 with gray-filled boxes.

The first pass is placed after the Polyglot parsing pass, and is responsible for gathering the program info regardless of its correctness. This means that all information is gathered and logged even if errors exist in the source code. After that, Polyglot continues with the other passes until it reaches the Validation pass which guarantees that the program is fully checked across all aspects including type checking, reachability checking, etc. If the program is validated successfully, then our second pass is reached and it starts gathering information again, but now

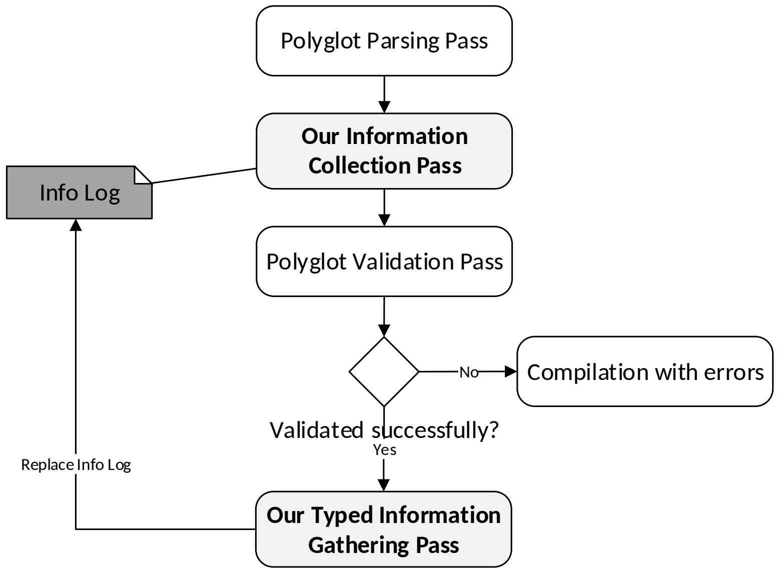


Figure 4.1: Extended Polyglot passes

they are enriched with types. Otherwise, the program has compilation errors and our second pass is not reached, which means that the tracer and visualizer would continue their work with a dirty program.

4.3.5 XML Output Format

As we saw in Chapter 2, the state-of-the-art techniques for reverse engineering usually use XML representation for storing the raw information collected from parsed programs. Such information are generated either at runtime (i.e., through dynamic analysis) or by analyzing the source code (i.e., static analysis). This is useful as it allows to trace specific scenarios of a program.

Likewise, in our case, we have used the same representation (i.e., XML files) but with the introduction of new XML tags, elements, and attributes. Basically, each XML document must have a root element, which in our case named either "<Project>" in case there are semantic errors in the program or "<Typed_Project>"

if the program is free of errors. All other program elements (or nodes) are stored inside this root element.

Class and interface declarations are represented using <Class> element with an attribute called 'type' that identifies whether it is a 'class' or 'interface'. Similarly, method and constructor declarations are represented using <Method> element with an option 'type' that indicates whether it is a 'method' or 'constructor'. Method and constructor calls are represented using a <Call> element, where an attribute is set to indicate whether the callee is defined within the program or it is external. External callees are the ones that contain methods that are not defined in the program itself, but from other packages like language or system libraries. All elements contain the three attributes: 'file', 'posStart' and 'posEnd' to determine their correct locations (i.e., line numbers in the their contained Java source code). This can help in navigating the source code at any time during visualization, or even during our implementation.

Let us take the below sample class. It is clear that this class has a Javadoc comment and contains only one method, which is composed of a variable declaration and an 'if' condition. The 'if' condition has actually two branches, one is its consequence while the other is the alternative. In the consequence, we have a call to the method 'factorial' while in the alternative we have a return statement.

```
/**
```

```
  This is the Main class of the program.
*/
public class Simple
{
   public static void main(String args[])
   {
      int i;
      if(i > 0)
      {
         factorial(i);
      }
      else
         return;
   }
}
```

Once the above class is parsed using our information gatherer, it generates its corresponding XML file that contains all the important information about that program, as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<TypedProject>
  <Class name="Simple" package="null" type="class" modifiers="public"
         extends=""    implements=""   posStart="4" posEnd="16"
         filepath="C:\myprojects\Simple.java">
        <Javadoc>
           This is the Main class of the program.
        </Javadoc>
        <Method name="main"           params="String args" type="method"
                paramTypes="String[]" modifiers="public static"
                returnType="void"     posStart="6" posEnd="15">
         <Declare name="args" type="String[]" posStart="6" posEnd="6"/>
         <Declare name="i"    type="int"       posStart="8" posEnd="8"/>
         <ControlFlow type="if" params="i &gt; 0" posStart="9" posEnd="14">
            <Branch type="Cond" params="i &gt; 0"/>
            <Branch type="Then" params="i &gt; 0" posStart="10" posEnd="12">
               <Call name="factorial" callee="Simple" calleeType="Simple"
                     params="i" paramTypes="int" chained="No" nested="No"
                     recursive="No" posStart="9" posEnd="9">
                     <Arg type="var" name="i" posStart="9" posEnd="9"/>
               </Call>
            </Branch>
```

```
          <Branch type="Else" params="" posStart="13" posEnd="14">
            <Return params="fact" posStart="14" posEnd="14"/>
          </Branch>
        </ControlFlow>
      </Method>
  </Class>
</TypedProject>
```

4.4 The Query-based Tracer of XML-stored Program Interactions

To extract the actual program behavior we need to trace the calls in a given program starting from the main method(s). Indeed, the program might have more than one entry point, where each of which can generate a different scenario. The user can participate in determining the main entry point to the corresponding program in order to have the desired program behavior.

The tracing technique is composed of a set of heuristics built in Java program that employs XPath [70] parsers and queries to access, load, and search for particular information from the XML log file generated by our previous technique built as an extension of Polyglot. The tracer launches its work by executing an XPath query that returns the entry points of the program. If one entry point exists, then the tracer starts tracking method calls recursively until it reaches the end of the entry point. If there are multiple entry points in the program, the tracer informs the user about the existing main methods in the project. As a result, the user is asked to participate in the determination of whether to proceed with a particular entry point or let the tracer go through each of them independently in order to produce all possible obtainable scenarios in the program.

While tracing, the tracer keeps notes about the lifelines, objects, fragments, messages, and lifelines of the corresponding scenario. These notes are stored in an XMI document that represents the resulting sequence diagram. As mentioned earlier in this report, several extensions to SD have been proposed to enrich it with expressive information that helps in program comprehension, the XMI representation of SD is extended with new tags, elements, and options that fulfill the goal of such extensions. These extensions are intended to be added to the XMI representation in a way that keeps it backward compatible with the standard UML SD. In other words, though our representation of SDs uses extended XMI representation, it can be visualized using the conventional tools.

4.4.1 Extended XMI Representation

To support our objectives in this thesis, we have extended the standard notation of UML sequence diagrams with XMI tag and visual elements that are capable of carrying all the required information about the proposed extensions.

Each of our proposed extensions has its own XMI representation, represented by elements and attributes. To show the extended XMI of our proposed extension, we have captured screenshots of applying these extensions to selected sample programs.

4.4.1.1 Extension Creation and Identification

We have assigned our extension an XML namespace called "xmlns:xxmi". This namespace is defined in the main "<uml:Model>" element that encompasses all elements of any given sequence diagram. This namespace is used as a qualifier for our proposed elements and attributes. For instance, static blocks are represented in our extended XMI representation using the element "<xxmi:staticBlock>". Likewise, extended attributes of standard element are given their names aligned with the 'xxmi' namespace as prefixes; e.g., for the '<message>' element, a new attribute called "xxmi:isRecursive" has been proposed. Note that attributes of the new extended elements are not given the prefix 'xxmi:'.

4.4.1.2 Generating elements' identifiers (IDs)

Each element in the generated XMI representation is given a unique identifier. This identifier is important when one element refers to another element in the XMI file. We have differentiated elements identifiers in our XMI representations with proper prefixes to help us recognize the type of the element during visualization. Types of element identifiers include the IDs of the messages, lifelines, and fragments and so many others representing additional and assisting elements as well the elements representing our proposed notations.

The format of the automatically generated IDs is given by the triple **"<prefix><underscore><number>"**, where '<prefix>', '<underscore>', and '<number>' represent placeholders for the appropriate prefix type, '_', and a type-based unique number, respectively. For each element type, the unique number begins counting from one, in a 6-leading-zeros style. For example, the first element of the type 'message' is given the id of 'message_0001', while the first element of the type 'lifeline' is given the id of 'lifeline_0001' in the same XMI file.

4.4.1.3 Element location in the source code

For each element described below, we observe the occurrence of the following attributes:

```
xxmi:file="C:\tests\cases\Sample.java"
xxmi:posStart="13"
xxmi:posEnd="13"
```

These attributes hold information about the location of a certain element in the source code, which assist in navigating the source code of that element. The location is represented by the file path, starting line, and ending line of any particular element.

4.4.1.4 Opening/Closing enclosing elements

For representing a block of calls, we have used an approach that is different from the typical approach. In other words, it is typical to use opening and closing tags for including a set of elements inside another enclosing one. This approach works well in our XMI representation, but the problem occurs when we tried to make it compatible with the standard XMI model. If another standard tool tried to parse our extended XMI, it would not be able to fetch the interactions that are inside an enclosing element that is created by our methodology. The reason behind this problem is that other tool always ignores non-standard elements and attributes, which means that the entire extended block (with its inner interactions) will be skipped.

Therefore, we have introduced another solution to this problem to make our output compatible with other standard tools. To this end, instead of using opening/closing tags for a certain block of interactions, we have used opening/closing elements. To open a new enclosing element we assign it an ID followed by an attribute that marks it as it is an opening tag; i.e., *'xxmi:status="start"'*. To close this enclosing element, we use another value for this attribute that refers to the closing mark; i.e., *'xxmi:status="end"'*. Now, if our extended XMI is parsed by other tools, only the opening and closing elements will be ignored, meaning that all interactions happening inside them will be readable.

4.4.1.5 Static initialization

For interactions accomplished during the static initialization process that is usually accessed by the JVM before accessing the main method, we have created an additional lifeline in the resulting SD layout that will represent the source of all these interactions. This lifeline is colored with a special color to distinguish it from other lifelines and given the name of the format "Static@<MainClassName>", where the '<MainClassName>' represents a placeholder for the name of the main class of the program.

In order to correctly capture the initialization process, we have made a test case that involves a set of static classes, variables, methods, and block. We scattered different message-printing methods to recognize which part is executed before. After the execution, we have realized that the execution is based on their appearance in the source code.

```
<xxmi:staticBlock xmi:id="staticblock_0001"
```

```
                  xxmi:status="start"
                  filepath="C:\tests\cases\Sample.java" posStart="9" posEnd="11"
/>
   <!--Sequence of Variable and Message Occurrences & CombinedFragments-->
<xxmi:staticBlock xxmi:status="end"/>
```

All interactions executed within static blocks are placed between the opening (i.e., with *xxmi:status="start"*) and closing (i.e., with *xxmi:status="end"*) elements of the static block.

4.4.1.6 Variable declarations

In this extension, we provide a new elements called "<xxmi:var>" that preserves the general information of variable declarations, including name, data type, type (i.e., local or member), location, etc., as shown below:

```
<xxmi:var xmi:id="var_0001"
          name="str"
          type="String"
          xxmi:kind="Local"
          xxmi:hasAssign="Yes"
          xxmi:file="C:\tests\cases\Sample.java" xxmi:posStart="13" xxmi:posEnd="13"
/>
```

Each variable-declaration element has an ID, name, and type. The attribute "xxmi:kind" differentiate local variables from the variables that are declared as parameters of a given method/constructor. If the variable has assigned a values, then the "xxmi:hasAssign" attribute will be given the values ‘Yes’; otherwise, a ‘No’ value will be given. In addition, the location of the variable in the source code is provided as well.

Another element of the type "<fragment>" and event ‘defaultEventOfVarOccurrency' will be generated to reflect the occurrence of such variable declaration. This element will have its own ID along with a pointer to the ID of the intended variable declaration.

```
<fragment xmi:id="varOccur_0001"
          var="var_0001"
          event="defaultEventOfVarOccurrency"
          xmi:type="xuml:VarOccurrenceSpecification"
          xxmi:status="start"
          covered="lifeline_0001"
/>
   <!--Sequence of interactions used for variable assignment-->
<fragment event="defaultEventOfVarOccurrency" var="var_0001" xxmi:status="end"/>
```

As we can see, the variable occurrence contains opening and closing fragments. This is needed for enclosing all interactions that represent the set of method calls used for assigning a value for such a variable, if any. If so, this occurrence will cover lifelines (covered by the calls used for variable assignment) by listing their IDs in the 'covered' attribute.

4.4.1.7 Extended Lifelines' Objects

In order to support the differentiation between lifeline types proposed in 3.1.3, we have added new attributes to the classes that are always connected to lifelines in the the XMI representation of SD. These classes are usually created using an element of the type "<packagedElement>", as follows:

```
<packagedElement
   xmi:type="uml:Class"
   xmi:id="class_0001" name="SB"
   xxmi:locality="Local" xxmi:NonLocalType=""
   xxmi:file="C:\tests\cases\Sample.java" xxmi:posStart="1" xxmi:posEnd="18"
/>
```

The attribute 'xxmi:locality' denotes to whether the lifeline corresponds to a locally-created class/object or utilized from language libraries (i.e., the possible values are either 'Local' or 'Lang'). If it is not local, the 'xxmi:NonLocalType' attribute will have a value that defines the general type of this class/object.

Now, to associate this class to a specific lifeline, we should refer to its ID at the corresponding lifeline. Depending on the standard XMI of SDs, each lifeline should correspond to an "<ownedAttribute>", which in turn refers to the ID of its class type, as shown below:

```
<lifeline
   xmi:type="uml:Lifeline"
   xmi:id="lifeline_0001"
   name="s:Sample"
   represents="ownedAttribute_0001" <!-- refers to the ID of the ownedAttribute -->
   coveredBy=""
/>
<ownedAttribute
   xmi:id="ownedAttribute_0001"
   name="Sample"
   type="class_0001" <!-- refers to the ID of the class -->
/>
```

4.4.1.8 Nested Calls (calls inside a call)

To represent nested calls, we have constructed three new XMI elements: "<xxmi:nestedcallfragment "<xxmi:originalCall>", and "<xxmi:innerCalls>". The element "<xxmi:nestedcallfragment>" is used to enclose all the nested interactions executed inside the arguments of the original call, including the original call itself "<xxmi:originalCall>" and its inner messages "<xxmi:innerCalls>". The following XMI representation elaborates this particular case:

```
<xxmi:nestedcallfragment xmi:id="nestedcallfragment_0001"
                    message="message_0001"
                    xxmi:status="start"
/>
```

```
  <xxmi:originalCall covered="lifeline_0001" xxmi:status="start"/>
    <!--MessageOccurrency of the original call-->
  <xxmi:originalCall xxmi:status="end"/>
  <xxmi:innerCalls covered="lifeline_0001" xxmi:status="start"/>
    <!--Sequence of interactions that appear in the arguments-->
  <xxmi:innerCalls xxmi:status="end"/>
<xxmi:nestedcallfragment xmi:id="nestedcallfragment_0001" xxmi:status="end"/>
```

4.4.1.9 Chained Calls

Chained calls are the set of method calls that can be implemented using a single statement where each call depends on an object returned by its preceding one. Actually, while visiting the statements that involve a chain of calls, Polyglot's visitor visits that calls from right to left. For example, if we have the same example discussed in section 3.1.7:

```
A.method1().method2().method3();
```

Polyglot in this particular case will firstly visit the call to `method3`, then `method2`, and finally it finishes by visiting `method1`, while the actual scenario is executed in the opposite direction. This means that this order is also preserved in the XML generated by the information collector. Therefore, to resolve this issue, we have managed a heuristic that can parse the chained calls in their correct order.

The set of chained calls is represented in our extended XMI of sequence diagram using a vertical bar on which all the correlated calls in the chain lie. This bar is represented by a new element called "<xxmi:chaincallbar>" that encompasses all such calls. The order of these call depends on their execution from left to right. The first call returns an anonymous object that is used to call the second call. Then, the second call does the same for its following call, and so on.

```
<xxmi:chaincallbar
   xmi:id="chaincallbar_0001"
   covered="lifeline_0001"
   xxmi:status="start"
/>
  <!--Sequence of interactions that appear along the vertical chain bar-->
<xxmi:chaincallbar xmi:id="chaincallbar_0001" xxmi:status="end"/>
```

For each message appearing inside this bar, we have added an attribute that indicates at which bar this message lie. The name of this attribute is 'xxmi:chainBar' and its value contains the 'ID' of the chain bar where this message appears. If a given message does not relate to any chain bar, then the value of this attribute would be empty.

4.4.1.10 Call-driven Combined-Fragment Specifications

It is known that loops, options, and alternatives are represented in the standard XMI using an element called "<CombinedFragment>". This element is divided into operands, where each operand involves of two sections: one is for the guard specification(s) and the other is for the interactions executed inside the block of the operand. It is common for guards to use text-oriented specifications. However, if the specification is based on method calls, then it is necessary to extend its element representation to contain the messages that reflect the occurrences of such calls. Therefore, we have extended the guard specification element to hold this kind of message-oriented specifications. Now, the following 'for' loop:

```
for(int i=0; isTrue(); increment()) { }
```

is represented as follows:

```
<fragment xmi:id="fragment_0001"
    xmi:type="uml:CombinedFragment"
```

```
     name="loop:For"
     covered="lifeline_0001 lifeline_0002"
     interactionOperator="loop"
     xxmi:file="C:\tests\cases\Sample.java" xxmi:posStart="18" xxmi:posEnd="18"
 >
   <operand xmi:id="fragment_0001.operand_0001"
            name="operand.operand_0001"
            xmi:type="uml:InteractionOperand"
            xxmi:type="Body"
            xxmi:value=""
            covered="lifeline_0001 lifeline_0002"
   >
     <guard xmi:id="fragment_0001.operand_0001.guard_0001"
            name="guard.guard_0001"
            xmi:type="uml:InteractionConstraint"
       >
       <!------ Initialization part ------->
       <specification xmi:id="fragment_0001.operand_0000.guard_0001.specification_0001"
                      xmi:type="Init" value="int i = 0" covered=""
       >
         <fragment xmi:type="xuml:VarOccurrenceSpecification"
           xmi:id="varOccur_0001"
           event="defaultEventOfVarOccurrency"
           var="var_0001"
           covered="lifeline_0001"
           xxmi:status="start"
         />
         <fragment event="defaultEventOfVarOccurrency" var="var_0003" xxmi:status="end"/>
       </specification>
       <!------ Condition part ------->
       <specification xmi:id="fragment_0001.operand_0000.guard_0001.specification_0002"
            xmi:type="uml:LiteralString" value="this.isTrue()" covered="lifeline_0001"
       >
         <fragment xmi:id="messageSendOccur_0001"
                   xmi:type="uml:MessageOccurrenceSpecification"
                   event="defaultEventOfMessageOccurrency"
                   message="message_0001"
                   covered="lifeline_0001"
         />
         <fragment xmi:id="messageReceiveOccur_0001"
                    xmi:type="uml:MessageOccurrenceSpecification"
                    event="defaultEventOfMessageOccurrency"
                    message="message_0001"
                    covered="lifeline_0002"
         />
```

```
      </specification>
      <!------ Iterative part ------->
      <specification xmi:id="fragment_0001.operand_0000.guard_0001.specification_0003"
                     xmi:type="Iters"
                     value="this.increment()"
                     covered=""
      >
        <fragment xmi:id="messageSendOccur_0002"
                  xmi:type="uml:MessageOccurrenceSpecification"
                  event="defaultEventOfMessageOccurrency"
                  message="message_0002"
                  covered="lifeline_0001"
        />
      </specification>
    </guard>
    <!--Inner sequence of Variable and Message Occurrences & CombinedFragments-->
  </operand>
</fragment>
```

We observe that this ‘for’ loop contains one operand consisting of two parts: the guard and interactions. The guard consists of three specifications: the initialization, condition, and iteration. The initialization part has an occurrence of a variable declaration of ‘int i’. The condition has a message representing the call to ‘isTrue()’ that in turn triggers a return message. The last part is the iteration, which is used for inc/decrementing ‘i’ using the method call ‘this.increment()’. Then, messages representing method calls inside the ‘for’ loop are placed between the closing tag of the guard and the closing tag of the operand. Notice that alternatives (e.g., ‘switch’ or ‘if’ with else part(s)) contain more than one operand, where each operand contains a single specification, which is the condition.

4.4.1.11 Recursive Calls

Recursive calls are considered as one of the crucial components of program interactions that need to be effectively handled. Actually, ignoring recursive calls in our approach would lead to infinite loops that would eventually lead to Stack Overflow exceptions. Therefore, capturing the existence of such calls is required for resolving this issue as well as increasing program understandability.

A *Hashmap* was maintained to preserve the signature of each invoked method call that has an execution body (it is removed from the Hashmap once it finishes its execution). During the execution of such a method call, if a method call matches its signature, then the firstly called method is considered as recursive, and we point to its ID in the new call to it. The resulting XMI representation explains this scenario by having an attribute to each message called 'xxmi:isRecursive'. The value of this attribute is set 'Yes' if the message is recursive (i.e., there is another call to it while it is under execution) and 'No' otherwise. For any message that invokes that recursive message, we give it the ID of that message stored in a new attribute called 'xxmi:recursiveMessageId'. We have also created another attribute for message elements called 'xxmi:returnType'. This attribute holds the type of the returned value of that message. Lets have the following example:

```
void main(String args[]){
   direct_recursive(5);
}
void direct_recursive(int i){
   if(i > 0)
      direct_recursive(i-1);
}
```

This particular example represents the case where the recursive call occurs inside the method itself.

```
<message xmi:id="message_0001"
         messageSort="synchCall"
         name="direct_recursive(5)"
```

```
        xxmi:isRecursive="Yes"
        sendEvent="messageSendOccur_0001"
        receiveEvent="messageReceiveOccur_0001"
        xxmi:chainBar=""
        xxmi:recursiveMessageId=""
        xxmi:returnType=""
        xxmi:file="C:\tests\cases\Sample.java" xxmi:posStart="7" xxmi:posEnd="7"
/>
<message xmi:id="message_0002"
        messageSort="synchCall"
        name="direct_recursive(i-1)"
        xxmi:isRecursive="No"
        sendEvent="messageSendOccur_0002"
        receiveEvent="messageReceiveOccur_0002"
        xxmi:chainBar=""
        xxmi:recursiveMessageId="message_0001"
        xxmi:returnType=""
        xxmi:file="C:\tests\cases\Sample.java" xxmi:posStart="26" xxmi:posEnd="26"
/>
```

It can also happen that the invocation to the recursive call occurs from outside the body of the currently executing method. The following example demonstrates this case:

```
void main(String args[]){
    indirect_recursive(5);
}
void indirect_recursive(int i){
    if(i > 0)
        another_method(i);
}
void another_method(int i){
    indirect_recursive(i-1);
}
```

This kind of mutual recursion is also captured in our approach and represented using the same extended XMI notation demonstrated above, which employs a reference to the first invocation of the recursive call in the other calls of it happened while it is executing.

4.4.1.12 Documentation comments (Javadocs)

Documentation comments are represented using a new element called "<xxmi:JavaDoc>". This element is generated for each class or method declaration that has a Javadoc comment. These comments are gathered from the source code without any modification since it is intended to parse them during visualization as HTML code. The only two elements that can have a Javadoc elements are: "<message>" and "<packagedElement>" of the xmi:type 'uml:Class'.

```
<xxmi:JavaDoc>
      This is the main class/method of the program.
</xxmi:JavaDoc>
```

4.5 The Visualizer of XMI-represented Program Interactions

Visualizer is the tool that is used to display the resulting SD that is already represented in an XMI file. The visualizer should take care of several issues such as element positioning, diagram dimensions, and what/how to show/hide.

As a starting point for our visualization work, we have searched for all available closed/open source tools that either reverse engineer programs or design UML sequence diagrams to investigate their ability to visualize standard XMI representations of SDs. We downloaded all well-known tools (listed in table 4.2) in this context.

Table 4.2: SD visualization tools with their attempts towards visualization

Tool	Import	Element Drawing	Correct Visualization
ArgoUML v0.34 [71]	✓	×	×
Altova UModel v.2015 Prof [72]	✓	×	×
Enterprise Architect v12.0 [73]	✓	✓	×
Modelio v3.4 [74]	✓	×	×
StarUML v2.5.1 [75]	✓	×	×
Trace Modeler v1.6.11 [76]	×	×	×
Visual Paradigm v12.2 [77]	✓	×	×

To the best of our knowledge, some of these tools could never import standard XMI representations of sequence diagrams in order to visualize their elements. Other tools could actually import the standard XMIs, identify their elements, and detect errors if any, but the problem is concerned with visualization. None of these tools could visualize the elements. The main reason for this restriction is that these tools always use various UML extensions for representing the visualization of diagram elements, such as coordinates, colors, etc. Consequently, if such extensions are not available, sequence diagrams cannot be visualized in such tools. The only tool that tried to somewhat visualize of the standard XMI representation is Enterprise Architect. However, all elements of the imported diagrams are stacked at the top-left journal of the layout with missing most of the important information and loosing the correct order and direction of the messages/lifelines.

As a result, we have been motivated by these limitations of the existing tools to build a tool that is capable of demonstrating and visualizing XMI representations of sequence diagrams, even if they were generated by other tools. This indeed means that our tool would be capable of visualizing standard XMIs as well as XMIs that are generated and extended using our reverse engineering approach.

Notice that XMI files have been extended and generated by other tools can also be visualized in our visualizer. This is achieved by the visualizer's ability to parse only standard elements in case the file contains other unknown elements (i.e., proprietary extensions generated by other tools are escaped).

4.5.1 Extended Elements to the SD Model

4.5.1.1 Static initialization

In order to distinguish the interaction executed throughout the initialization process, we have created a standalone lifeline that would be responsible for sending/receiving the interaction messages to the other lifelines. It can be observed from Figure 4.2 that the lifeline used for this task is named 'Static@SB', where SB is the name of the main class of the program. It can also be seen that interactions of the Static lifeline have started before the ones of the 'Main@SB' lifeline, which is concerned with the interactions executed in the main method of the program.

We have divided the interactions used for static initialization into two categories: the ones executed inside the static block and the ones executed in response to the initialization of the member variables.

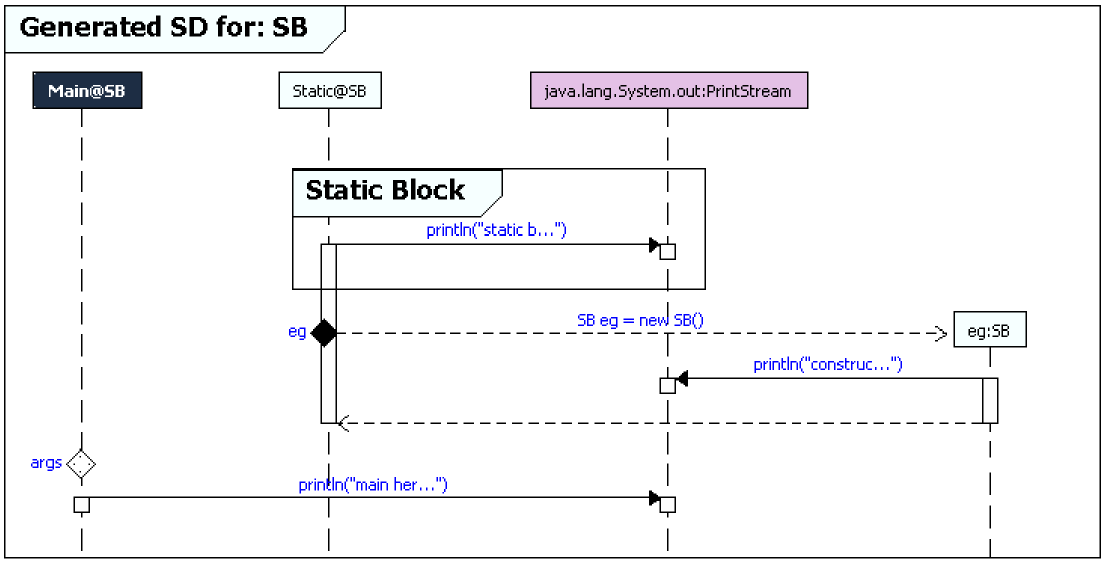


Figure 4.2: Static initialization elements

Static block is shown as an enclosing box containing all interaction messages (or fragments if any) executed within which.

4.5.1.2 Variable declarations

Sometimes, it is very important to show the declarations of the variables that have been used in the resulting SD. Users might see in a message or loop a reference to a variable without being known where this variable is declared and/or has been assigned a value. Therefore, we proposed two new elements to show the declaration of variables in the lifelines where these variables are declared. The first element that appears in a plain diamond as shown in Figure 4.3 and is used to represent the variables that are declared inside the method's arguments. The other element is shown with a black-filled diamond to demonstrate the declaration of local and member variables.

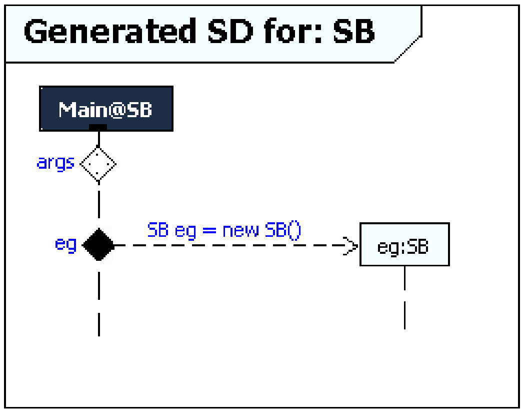


Figure 4.3: Variable declaration elements

Our extended XMI also contains information about the assignment statements used to assign a value for any given variable or change its current value. The demonstration of such information in the current implementation is done through the mouse hovering feature.

4.5.1.3 Extended Lifelines' types

In the standard SD, there are two kinds of lifelines that differentiate between objects and classes. Both of them have the same element style, but the difference is within the label. Object lifelines are represented in the form "<Object_Name>:<Class_Type_Name>", whereas class labels are represented in the form "<Class_Type_Name>". There is a special case where the object is anonymous (or default), which can usually be created using a statement like "`new A().m1()`" that creates single-use objects. In this case, lifeline labels would be of the form ":<Class_Type_Name>", where the object name is omitted.

We aim through our extension to distinguish the interactions of any given program that represent the communications between the program's internal objects and classes themselves as well as the communication between them and other entities of the system. This will allow users to recognize the amount of interactions of the program with the language or system libraries or components. In particular, we have distinguished the program accesses to the system console, file system, and the GUI. This could be identified from the language-supported libraries that allow programs to interact with this external entities.

As shown in Figure 4.4, in the current implementation of the proof-of-concept prototype, different lifeline colors have been applied to the different types of objects. For example, the lifeline that represents the 'Main' method is colored with a black color and uses the symbol '@' to attach the name of the main class that contains this method. Lifelines representing objects that relate to the $FileSystem$, $SystemConsole$, and GUI are represented using certain notations. Here, we just used colors to distinguish the lifelines of different categories. Other types of lifelines are drawn using the 'cyan' color. These colors can actually be changed in the future to different icons that intuitively reflect the main purpose of the corresponding lifeline. This extension is also helpful for enabling users to hide the interactions of a certain type of lifelines if they are not important for his understanding of the program behavior. Likewise, it can be useful for the ones who care about these interactions by providing them the facility to only show the interactions between programs and these lifelines.

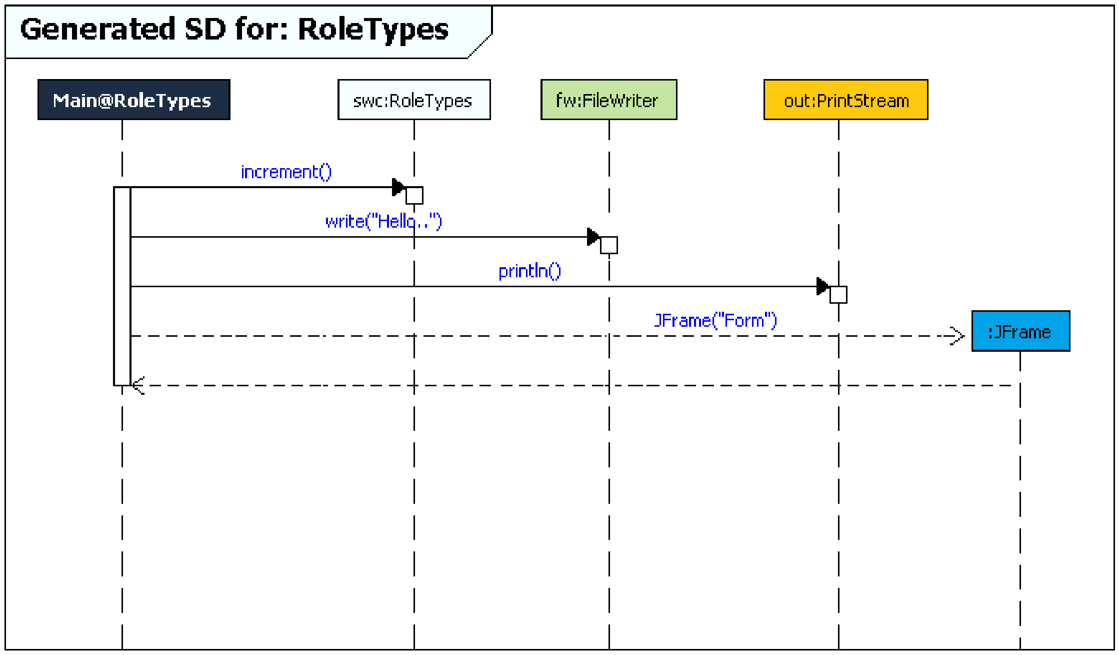


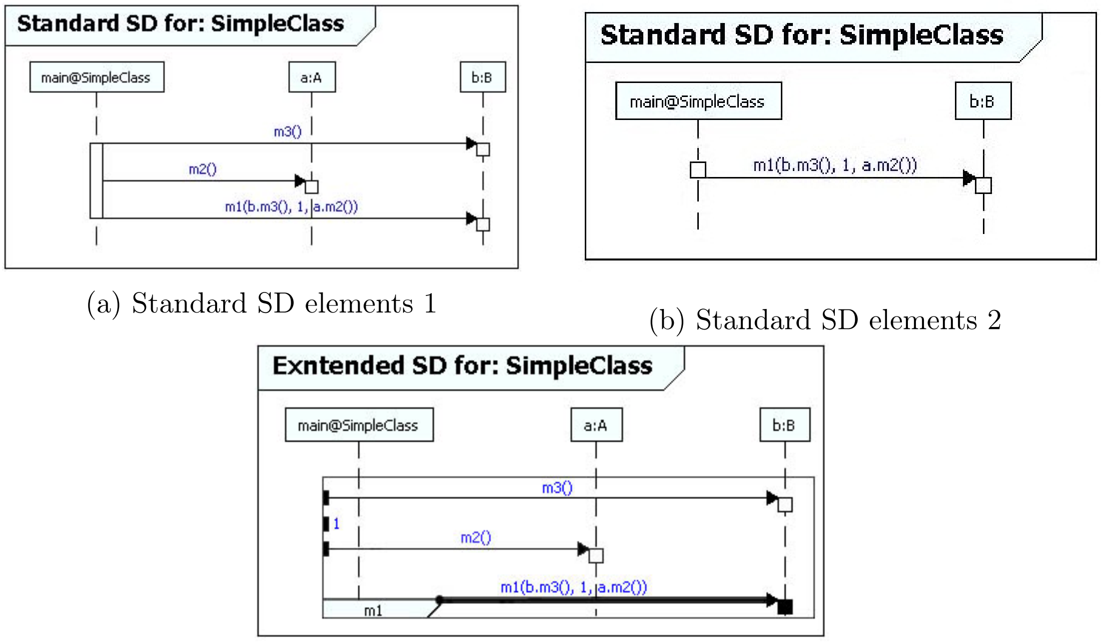
Figure 4.4: Extended Lifelines' Types

4.5.1.4 Nested Calls (calls inside a call)

Visualizing nested call is one of the major extensions in our work. Actually, visualizing such calls can be achieved using the standard SD elements (i.e., messages) in two different ways. The first way is to show the inner calls (i.e., calls inside the parameters) using separate message elements as shown in Figure 4.5a. another way is to only show the original call with a label that shows that it contains some other calls inside its parameters (i.e., without showing separate message lines) as shown in Figure 4.5a. However, both ways do not exactly reflect what is actually represented in the source code, because the execution body of the inner calls might contain lots of important interactions in the program. Therefore, we proposed a new SD element that can represent this kind of calls in an intuitive manner.

Now, to expressively demonstrate the different between the standard visualization and our extension, lets have the following example:

```
b.m1(b.m3(), 1, a.m2());
```



(a) Standard SD elements 1

(b) Standard SD elements 2

(c) Extended SD elements

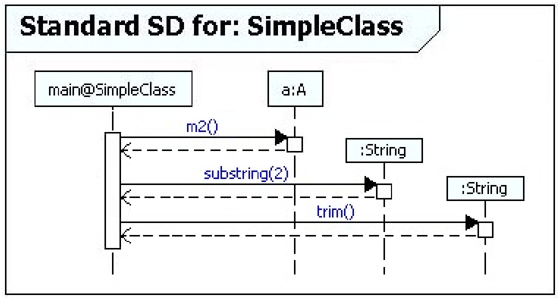
Figure 4.5: Nested calls visualization

As we can see in Figure 4.5c, through the object ‘b’ that is of type ‘B’, we have a call to the method ‘m1’. The first parameter in this method invocation we have another call to the method ‘m3’ from the same object, the second parameter is just a constant values, and the last parameter is also a call to a method called ‘m2’ but from another object called ‘a’ of type ‘A’. Our visualization elements could manage the representation of such a call by having what we have called ’A call box’ that represents the original call by a bold message line while showing all its parameters as small filled rectangles lying at the left side. We show the parameter as if it is a constant value or a variable. In the case of having a parameter with a method call, then thinner message lines will be drawn approaching the corresponding lifeline of owner object of that method. Of course, if the body of that method has plenty of interactions, then all of them will also appear inside the call box.

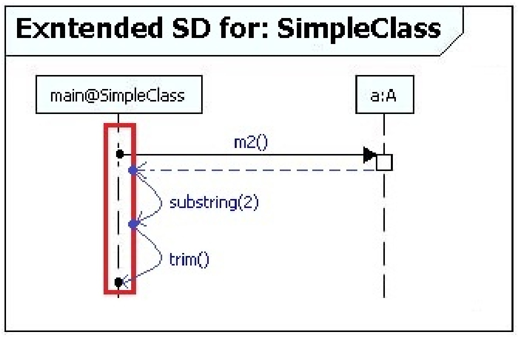
4.5.1.5 Chained Calls

As discussed earlier in this chapter, each method in a nested call statement returns an anonymous object that can be used to call the next method until we reach to the ultimate call that will return the final result. As shown in Figure 4.6a, the statement "`a.m2().substring(2).trim();`" uses three different lifelines for visualizing the three messages invoked through which. This will confuse the user as it might give an indication of the presence of three separate method calls in the program. This what motivated us to come up with our own extended element that can exactly show the statement calls in a way that is identical to their structure in the source code.

Figure 4.6b demonstrates our proposed notation for representing chained calls, which is provided as an extension to the UML SD representation. We can observe the red vertical bar that holds all messages executed within a single statement of chained calls. This bar starts with the message that represents the method that is directly connected to the first object used for initiating this kind of calls (i.e., the method 'm2' of the object 'a'). Then, after executing 'm2', it will return an object to the same vertical bar, which in turn will trigger the next message 'substring' with an integer parameter '2'. Likewise, after executing 'substring', another object will be returned to the bar to eventually be used to call the last method called 'trim' whose value is returned to the lifeline initiated the entire set of calls.



(a) Standard SD elements



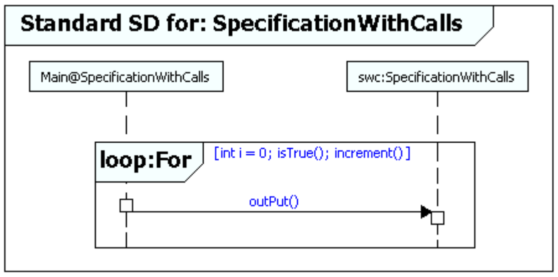
(b) Extended SD elements

Figure 4.6: Chained calls visualization

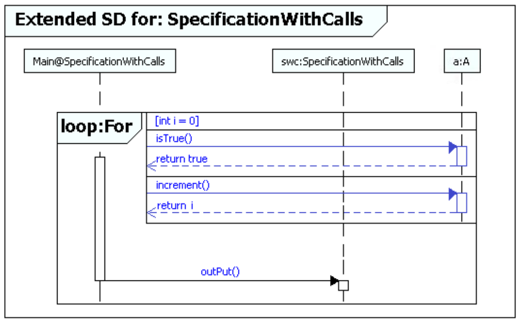
4.5.1.6 Call-driven Combined-Fragment Specifications

For the case of having loops, optional, or alternative clauses where their specification (i.e., their condition or iteration parts) is composed of a method call (and of course this call may trigger other calls inside the body of the used method), standard SD cannot show these internal calls, as shown in Figure 4.7a. Showing such calls may actually complicate the displayed SD but, on the other hand, it opens a window to other interactions that users may be interested in.

Figure 4.7b demonstrate out proposed extension the combined fragments that are usually used in UML 2.0 for representing interactions that are executing within a certain block. We can see that our extended SD notations could show all the interactions executed throughout the execution of the ‘for’ loop whose condition and iteration parts are based on method calls. Although the standard SD is simple and easy to read, lots of information is hidden from the user. In order to support the simplicity of such extension, we allow this kind of information to be hidden in a certain level of zooming.


Standard SD for: SpecificationWithCalls
Main@SpecificationWithCalls
swc:SpecificationWithCalls
loop:For
[int i = 0; isTrue(); increment()]
outPut()

(a) Standard SD elements


Extended SD for: SpecificationWithCalls
Main@SpecificationWithCalls
swc:SpecificationWithCalls
a:A
loop:For
[int i = 0]
isTrue()
return true
increment()
return i
outPut()

(b) Extended SD elements

Figure 4.7: Visualization of call-driven combined-fragment specifications

4.5.1.7 Recursive Calls

With respect to recursive calls, the UML standard of sequence diagrams represents them as self messages. This means that user should track any self message an check if it is still under execution in order to identify whether it is recursive or not. However, this applies only when the recursion happens within the same method. In case we have an indirect recursion, especially with methods of different classes, it will be represented using a normal message line. This urged us to come up with an extension that can expressively represent the case as shown in Figure 4.8. It is clear from the figure that referring to a recursive call is demonstrated using a turnaround line with the new argument passed to that method. Notice that, the label of the first message to the recursive method refers to the parameter of the first invocation but that does not mean that these parameters are always sent to that method.

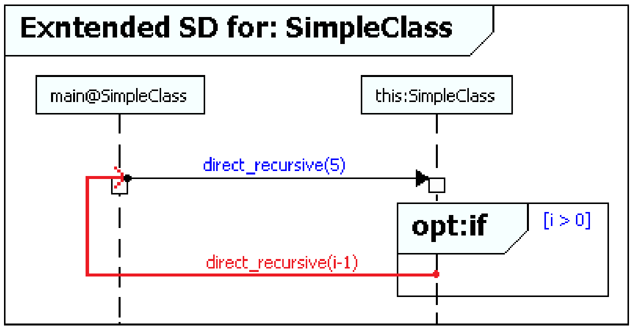


Figure 4.8: Recursive calls visualization

4.5.1.8 Documentation comments (Javadocs)

This extension is used to give more information about lifelines or methods whose actually purpose can not be gained through their names. We refer by this to the explanation of classes and methods that can be provided in the source code as Javadoc comments. Here, we allow users to show the HTML-based Javadoc comments in the SD layout in two different ways. The first way is by hovering among the particular element intended to know more information about. The other way is using the biggest zooming level that will show all information about elements in the layout.

Figure 4.9 demonstrates an example of displaying the Javadoc comments of the class ‘A’ that corresponds to the shown lifeline in the SD layout as a pop-up tooltip when hovering the mouse among the area of that lifeline.

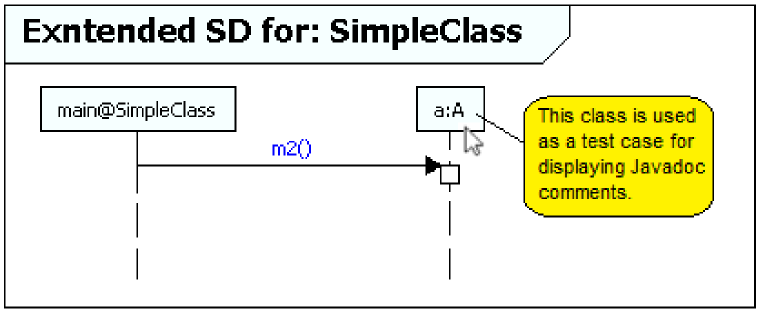

Exntended SD for: SimpleClass
main@SimpleClass
a:A
m2()
This class is used as a test case for displaying Javadoc comments.

Figure 4.9: Javadoc comments visualization

4.5.2 Extended Facilities to the Visualization Tool

4.5.2.1 Lifeline positioning

One challenge of the visualization process is determining the coordinates of the SD lifelines. Lifeline positioning can be addressed from different aspects. a) Objects of the same class can be located nearby each other. b) Classes of the same package can be located nearby each other. c) Classes and objects that have more interactions can be located close to each other regardless where they are located in the source code.

4.5.2.2 Diagram Dimensioning and Zooming (What to show/hide)

Diagram dimension should be relative to different factors. To make the diagram readable, it should be visualized with dimensions relative to the computer screen and resolution. In addition, the number of lifelines, fragments, and messages used in the diagram can also play a role in defining the proper dimension of the resulting diagram. Furthermore, the visualizer provides a zooming facility that enables users to select the desired dimension to their vision.

Zooming in/out depends on a set of predefined templates of levels that regulate the information demonstration in the produced SD. In other words, zooming-in to 150% would show more information than that with 100%, while zooming-out to 50% would reduce the amount of information shown. Therefore, in these templates, we can specify the information intended to be shown/hidden whenever a user attempts to zoom in/out. In addition, the visualizer should provide a facility that allows users to choose certain lifelines to be visualized (i.e., shading in/out). This would help in focusing only on the interactions between those lifelines while the others are disappeared, shaded-out, or decreased in size.

The visualizer also provides extra options that allow users to select the desired elements to be displayed. For instance, a user might need to zoom-in to 150% and, at the same time, he/she does not want to see the interactions with external lifelines. Hence, providing such options would assist in achieving this goal easily.

Actually, determining what to show and what to hide is a complex process and is based on the same factors as the ones used for dimension specification. This means that the amount of information to be displayed participates in deciding what information to show/hide. For example, an SD for a small program can show the self-interactions that show the messages from a certain lifeline to itself, while for larger programs such interactions may be hidden.

4.5.2.3 Mouse hovering

Some hidden information can be displayed on-demand using the mouse hovering facility. This means that the user can point out at the area of a certain element in the SD layout to see more information about it (e.g., Javadoc comments).

4.5.2.4 Source code navigation

To facilitate re-engineering of software projects, the visualizer should support accessing the source code of any item in the displayed SD. This includes declarations, calls, and control structures.

4.5.3 Tool Support

The proof of concept prototype of this work is deployed as a Java executable Jar application that can run under any environment. Since it is a prototype, it is not integrated yet with the other techniques proposed in this work that are responsible for collecting program information and tracing program interactions. Therefore, to visualize a Java project, one needs to run our second tool, specify the project path, and generate the XMI file(s) representing all the trace information. Then through the visualizer tool, the user can open that XMI file and everything will be displayed in the layout of the tool. A snapshot of the tool is shown in Fig. 4.10.

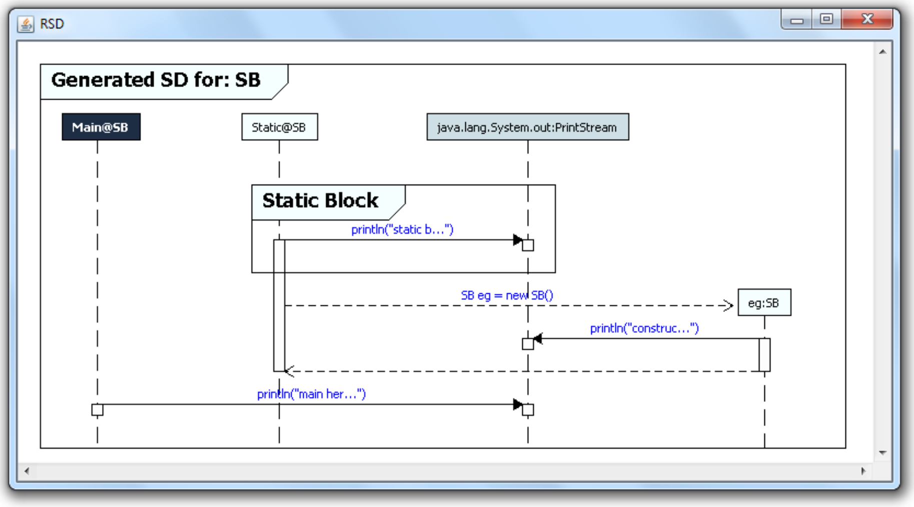


Figure 4.10: The visualizer tool

4.6 Limitations of the Current Implementation

In this section, we discuss some of the limitation in the current implementation of the proposed technique. These limitations may actually lead to producing imprecise diagrams in certain scenarios.

4.6.1 Constructing Objects Inside Constructors

Instantiating a new object inside a constructor can lead to producing one lifeline for that object. The lifeline will then be used by all constructed objects through that constructor. For instance, if we have the following program:

```java
class A{ public void go(){} }
class Example{
   public A a;
   public Example(){
     a = new A();
     a.go();
   }
   public static void main(String args[]){
     Example ex1 = new Example();
     Example ex2 = new Example();
   }
}
```

In our technique, the produced sequence diagram representing this particular program is shown in Fig. 4.12, whereas the anticipated SD should appear as shown in Fig. 4.11.

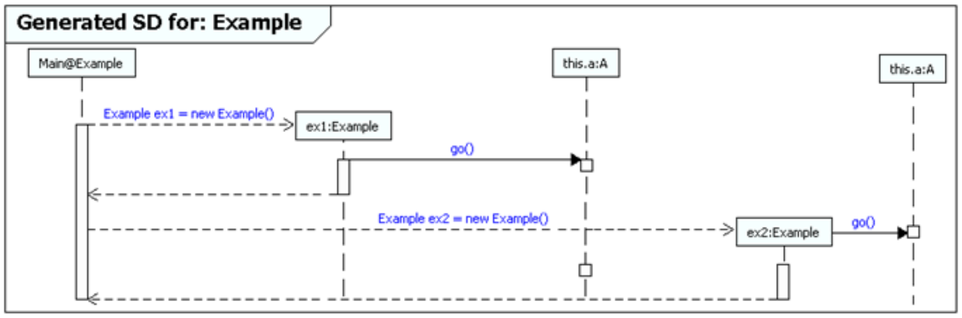

Generated SD for: Example
Main@Example
this.a:A
this.a:A
Example ex1 = new Example()
ex1:Example
go()
Example ex2 = new Example()
ex2:Example
go()

Figure 4.11: The expected SD for the object example provided

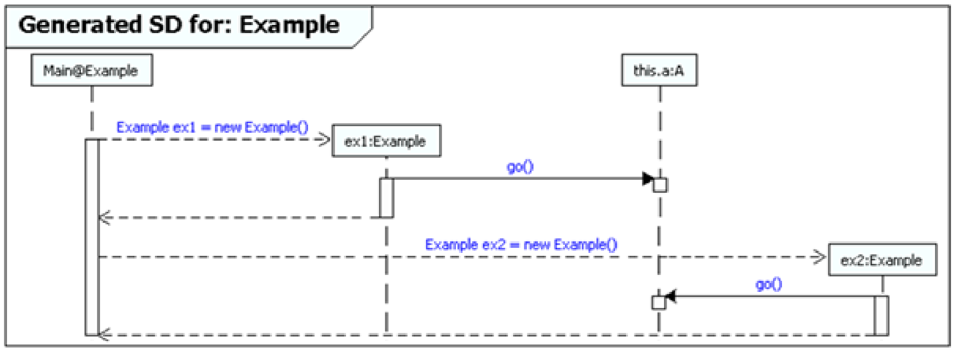

Generated SD for: Example
Main@Example
this.a:A
Example ex1 = new Example()
ex1:Example
go()
Example ex2 = new Example()
ex2:Example
go()

Figure 4.12: The produced SD for the example provided

4.6.2 Default Type Casting

JVM automatically casts the integer values to double ones in case of the argument is sent to a method or constructor that accepts a double parameter. Our approach in this particular case will search for the constructor that accepts integer, which is unlikely to be there. The following example represents the case where the constructor 'Example2' accepts a parameter of the type 'double'. When that constructor is accessed as shown in the example using an integer value, the JVM will use this particular constructor after casting the integer '2' into double, which results in having '2.0'.

```java
class Example2{
  public Example2(double d){
  }
  public static void main(String args[]){
    Example2 ex2 = new Example2(2);
  }
}
```

4.6.3 Event-based Programs

Almost all the interactions that are executed inside the methods that are called using event-driven actions are not captured in our program analysis techniques. This is due to the fact that this kind of interactions are dynamic and can only be captured using a dynamic analysis approach. Therefore, if we have the following program, for example:

```java
class Ex3 extends JFrame implements ActionListener
{
  public Ex3 () {
     JButton jb = new JButton("run");
     jb.addActionListener(this);
     this.add(jb);
  }
  public void actionPerformed(ActionEvent ev) {
     // All program interactions
  }
  public static void main(String args[]) {
     Ex3 ex3 = new Ex3();
     ex3.show();
  }
}
```

The produced sequence diagram for this particular program using our approach is represented as shown in Fig. 4.13, while actual program interactions can be much more than that simple behavior, of course.

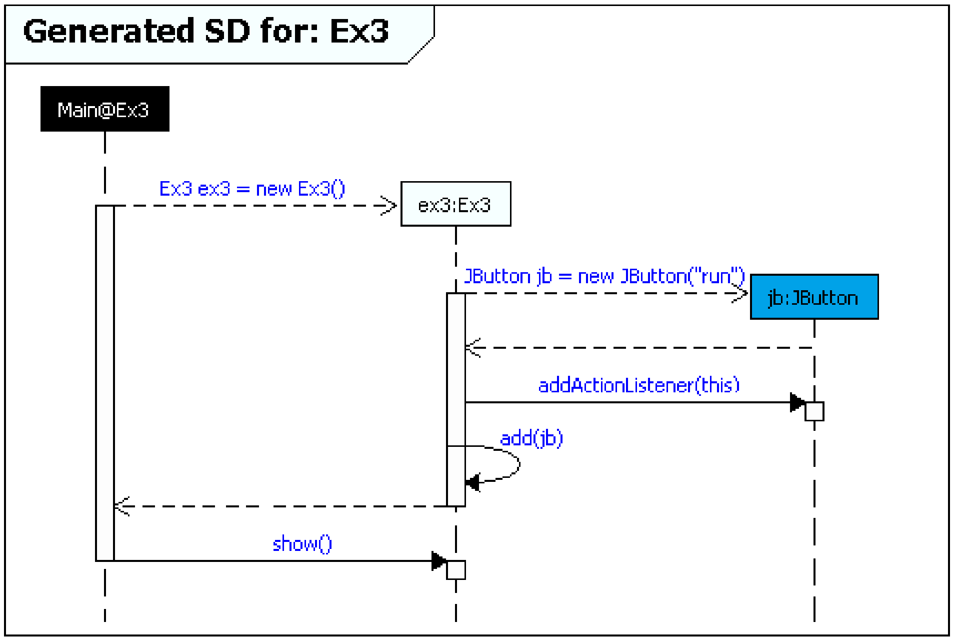


Figure 4.13: The produced SD for the event-based example provided

CHAPTER 5

EVALUATION

The purpose of this chapter is to quantitatively evaluate the validity, usefulness and effectiveness of the proposed Sequence Diagram (SD) extensions for program comprehension. Furthermore, to gain a deeper confidence of their added value, we investigate which aspects of the program control flow benefit the most from the proposed extensions to the sequence diagram. To achieve these goals, we have designed and conducted a controlled experiment in which we measured how these extensions could affect 1) the time that is needed for various kinds of comprehension tasks, and 2) the correctness of the answers provided by the participants during those tasks. Each task was represented as an independent question in the designed questionnaire, where for each question, it is needed to calculate the time spent for answering the question and the points assigned to all its answers. Tasks have been classified into different categories based on the type of answer expected for in each.

5.1 Experimental Design

To satisfy our objective of this chapter, we define a list of comprehension tasks of different kinds and measure the sequence diagram extensions' added value to the standard representation of the UML sequence diagram. We maintain a distinction between the correctness of the responses given and the time spent on the tasks. On the other hand, some tasks have been created for obtaining users' own evaluation of the proposed extensions in terms of usefulness, complexity, and precision in reflecting the actual flow of control. Furthermore, we have identified the different types of tasks to which the use of our extensions is the most advantageous. In addition, we selected Greenfoot [78] as a case study for our experiment as it includes different scenarios that cover most of our extensions.

5.1.1 Research Questions and Hypotheses

Based on our selected case study, we define the following research questions:

1. Does the availability of our proposed extensions to the sequence diagram reduce the time that is needed to achieve the comprehension tasks?

2. Does the availability of our proposed sequence diagram extensions increase the correctness of the answers provided during those tasks?

3. Is representing programs using our proposed sequence diagram extensions less complex and more precise than that with the use of the standard sequence diagram?

4. According to the answers provided to these research questions, which types of tasks benefited most from the availability of our proposed extensions and from control-flow modeling in general?

Then, we associate the first three research questions with three null hypotheses, formulated as follows:

- $H1_0$: The use of our proposed sequence diagram extensions does not affect the time needed to complete each comprehension task.
- $H2_0$: The use of our proposed sequence diagram extensions does not affect the correctness of responses given during those tasks.

After that, we have stated the alternative hypotheses used in the experiment, as follows:

- $H1$: The use of our proposed sequence diagram extensions decreases the time needed to complete each comprehension task.
- $H2$: The use of our proposed sequence diagram extensions improves the correctness of answers given during those tasks.

The first alternative hypothesis is motivated by the fact that the sequence diagram extensions we have introduced provide a detailed and explicit outlook of every particular construct of the subject system, which may help users to recognize the interactions of the system along with the different control structures in the source code more quickly than doing so with the use of the UML standard of sequence diagrams, which require users to implicitly infer about certain behaviors of the system.

On the other hand, the rationale behind the second alternative hypothesis is the inherent precision of our static analysis aligned with the meaningful notations used to differentiate between the various aspects of the expected program behavior. This indeed results in a more deep and detailed understanding and therefore provide more accurate answers.

The third alternative hypothesis is induced by the way and style our extensions are represented. The design of the style of the extended in intended to reflect the actual flow of control in programs by using the least number and size of elements.

In order for the hypotheses $H1_0$ and $H2_0$ to be tested, a set of comprehension tasks have been defined in a way that they can be addressed by both a control group and an experimental group. These two groups are differently treated, where the former group uses a standard UML sequence diagram, whereas the latter group has been given a diagram extended with our proposed notations. A between-subjects design is maintained to allow each subject to be either in the control group or in the experimental group.

5.1.2 The Object of the Experiment

The system that our experiment is based on is Greenfoot, a Java environment that simplifies the development of two-dimensional graphical applications and is meant for educational purposes of programming languages. Generating reverse-engineered sequence diagrams for the overall functionality of Greenfoot will for sure result in obtaining more complex and disappointing diagrams for the subjects to achieve the tasks. Therefore, we have selected only a specific scenario of Greenfoot used for browsing classes. This scenario is based on a class called *ClassBrowser*, which is responsible for drawing and laying out the classes on the user interface. The resulting diagrams contain more than 50 method calls between around 20 objects/classes.

Our choice of Greenfoot, and its selected scenario in particular, as the object of this experiment has been motivated by several factors, stated as follows:

- Greenfoot is open source, which is indeed necessary for our experiment since our technique is based on static analysis, which requires the availability of the source code.
- It is a modular environment, which allows the analysis and modeling of some of its scenarios easily.
- It is written in Java, with which many potential subjects are sufficiently familiar.
- The scenario of Greenfoot that we have chosen for our case study covers most of the our proposed extensions.

Before applying our approach to the case study, a stub class has been created as a starting point of the *ClassBrowser* functionality. After that, the standard and extended reversed-engineered sequence diagrams representing that scenario have been generated. For the standard one, we have used the Visual Paradigm 12.2 tool for producing the UML sequence diagram and used it as a reference. Our extended sequence diagram has been exported to a PDF file allowing users to search for certain terms or to zoom in/out while responding to the tasks assigned. For the evaluation to be completely fair, we have reproduced the standard sequence diagram using Visio and exported to a PDF as well.

The description and analysis of the methodology used to generate these diagrams have been discussed in chapter 3 of this thesis. Our prime objective in this chapter is to analyze whether the availability of our proposed extensions to the sequence diagrams is profitable during the program comprehension activity.

5.1.3 Task Design

With respect to the comprehension tasks that are to be addressed during our experiment, we tried our best to have them representative of real user needs as much as possible. In addition, we designed the tasks so that no task is biased toward either the UML sequence diagram or our extended one. However, since our objective is to evaluate the usefulness of the extended notations over the standard ones, some of the tasks initially appeared to be a bit biased to our extensions. To resolve this and to maintain the balance, we modified some the tasks in a way that made them somewhat easier to be solved using the standard sequence diagram.

In the literature, related controlled experiments for program comprehension applied the comprehension framework proposed by Pacione et al. [16], who classified the comprehension tasks of software visualization into nine primary activities. These activities are intended to represent both general and specific reverse engineering tasks and to cover both static and dynamic information. Again, since we are evaluating specific extensions over a standard representation as a benchmark (i.e., not the tool developed) and we are only interested in the behavioral information (i.e., structural information are not covered), most of such tasks do not apply to our case of evaluation. Therefore, we have designed the tasks to be of different types of questions, where each type requires a different kind of input from the users than the other types.

In particular, tasks in our experiment have been designed based on 4 different evaluation categories (Table 5.1). Categories C1, C2, and C3 represent different comprehension activities in which users are required to provide a different kind of answer to each. In these categories, the time users spent on answering each particular question of them is important since it measures how the information provided by the standard or extended sequence diagram is useful enough to get the answer quickly. In the C4 category, time requirement is not essential since they require users themselves to rate the complexity and precision of the diagrams provided in representing certain scenarios of interaction. Finally, C5 category is composed of only one question that asks users to give their own opinion about whether the proposed extension (and other ones not yet implemented) are useful and effective in understanding programs.

A collection of thirteen tasks has been proposed to cover the aforementioned

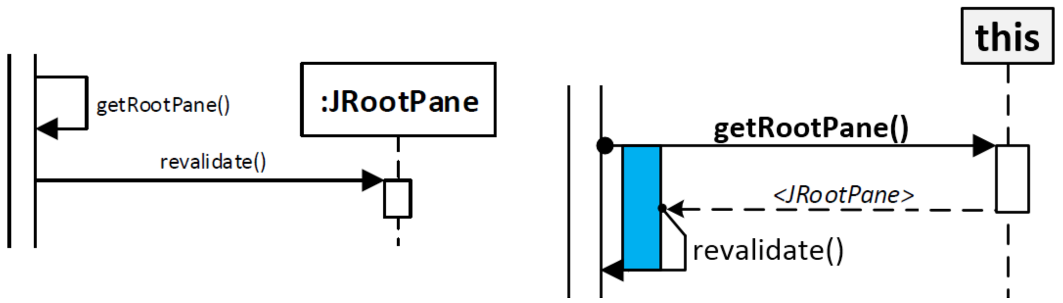
Table 5.1: Categories of evaluation tasks

Category	Description
C1	Searching for the number or names of certain program components
C2	Writing code representing a certain sub-diagram
C3	Snapping the sub-diagram representing some certain code
C4	Comparative rating
C5	General user opinion about all the extensions

comprehension categories. Category C1 contains the tasks T1.1, T1.2, and T1.3. Tasks T2.1, T2.2, T2.3, and T2.4 are of the category C2. Category C3 involves only one task, which is T3. Under category C4, we have four tasks, namely T4.1, T4.2, T4.3, and T4.4 while category C5 has only one task, which is T5.

We aim to highlight all aspects of our extensions in the ClassBrowser case study. Table 5.2 shows descriptions of the tasks by presenting how each of which covers the particular aspects of our proposed extensions. In addition to T5 that covers all the proposed extensions, each particular extension is covered by at least one task. For Example, The extension representing *Chained Calls* is covered by the Tasks T2.1, T2.2, and T4.1 and the *Nested Calls* extension is covered by T2.2, T3, and T4.3. Notice that some tasks may cover more than one extension.

With respect to the tasks of the categories C4 and C5, feedback obtained from users is not graded further. In other words, there was no need to grade responses gathered for these tasks as they are already of a rating kind. Feedback on the tasks of the category C4 represents an evaluation of the users themselves for the complexity and precision of the extended representation compared with the standard one. Similarly, feedback on the task of category C5 refers to how the



(a) Standard SD for the control group (b) Extended SD for the experimental group

Figure 5.1: Figures provided with task T2.1

proposed extensions were useful and effective towards understanding programs, from the point of view of the participants themselves.

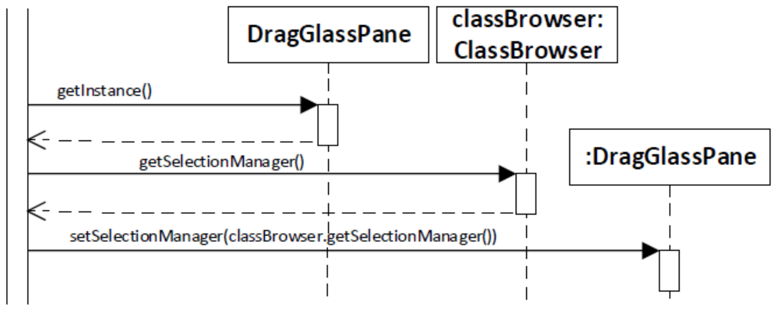
Rather than using multiple-choice question types, we provided our experiment with open questions, which made it harder for participants to guess the answers. This has made the tasks more representative in rendering real comprehension situations. In particular, this only applies to the tasks of the categories C1, C2, and C3. Rating and opinion questions, included in the tasks of category C4 an C5, were designed with multiple choice questions listing all possible ratings between 0 and 4. Each answer can at the end earn a point from 0 to 4. Points were awarded by a sole evaluator, the author of this thesis, to ensure a uniform and fair grading based on a solution model.

5.1.4 Subjects

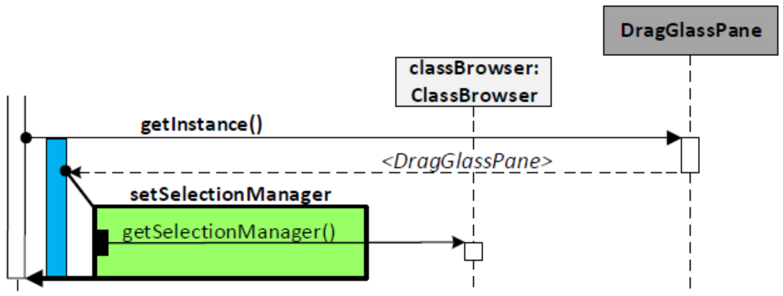
The subjects in this experiment are 2 PhD candidates, 8 MS students, and 12 BS students in the senior stage. The PhD and MS students are actually mixed in their degrees in the computer science department and the resulting group thus

Table 5.2: Descriptions of the comprehension tasks

Tasks	Category	Extensions	Descriptions
T1.1	C1	Type casting	List the names of all casting classes, if any, used for type-casting operations?
T1.2	C1	Recursive Calls	Write the name(s) of all recursive methods, if any?
T1.3	C1	Try-catch Block	How many Interactions originated (i.e., methods called) inside a `try` block?
T2.1	C2	Chained Calls	Write the code that corresponds to the excerpt of SD shown in Fig. 5.1?
T2.2	C2	Chained Calls + Nested Calls	Write the code that corresponds to the excerpt of SD shown in Fig. 5.2?
T2.3	C2	Lifelines of returned objects	Write the code that corresponds to the excerpt of SD shown in Fig. 5.3?
T2.4	C2	Fragment's operands with calls	Write the code that corresponds to the excerpt of SD shown in Fig. 5.4?
T3	C3	Multi-Nested Calls	Identify and screenshot the portion of the sequence diagram that reflects this code: `classBrowser.quickAddClass(newClassView( classBrowser, newGCoreClass(Actor.class,project)));`
T4.1	C4	Chained Calls	Rate the Complexity and Precision of the sub-diagram in Fig. 5.5 in representing this code:? `this.getRootPane().revalidate();`
T4.2	C4	Fragment's operands with calls	Rate the Complexity and Precision of the sub-diagram in Fig. 5.6 representing this code:? `for(Iterator iter = roots.iterator(); iter.hasNext()){}`
T4.3	C4	Nested Calls	Rate the Complexity and Precision of the sub-diagram in Fig. 5.7 representing this code:? `BorderFactory.createTitledBorder(null, Config.getString("browser.border.world"));`
T4.4	C4	Recursive Calls	Rate the Complexity and Precision of the sub-diagram in Fig. 5.8 representing this code:? `Component createClassHierarchyComponent( Collection roots,booleanisRecursiveCall) { createClassHierarchyComponent(children,true); }`
T5	C5	All Extensions	Give your opinion about supplying UML sequence diagrams with the proposed notations shown in Table5.3, in terms of usefulness and effectiveness for program comprehension?

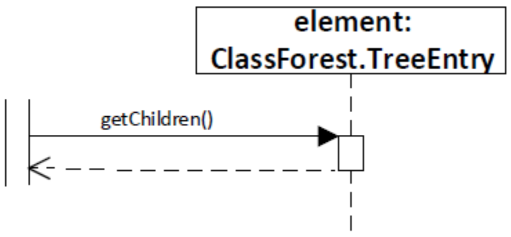

DragGlassPane
classBrowser: ClassBrowser
:DragGlassPane
getInstance()
getSelectionManager()
setSelectionManager(classBrowser.getSelectionManager())

(a) Standard SD for the control group

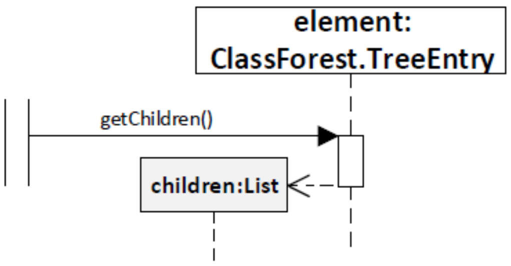


(b) Extended SD for the experimental group

Figure 5.2: Figures provided with task T2.2

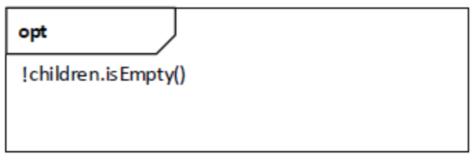


(a) Standard SD for the control group

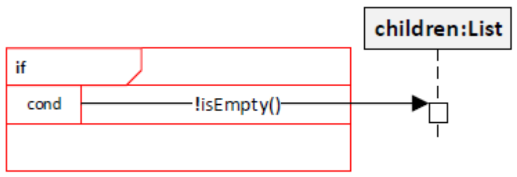


(b) Extended SD for the experimental group

Figure 5.3: Figures provided with task T2.3

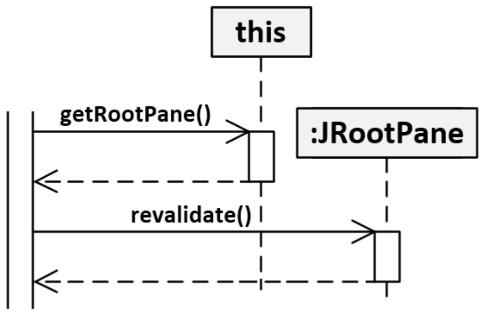


(a) Standard SD for the control group

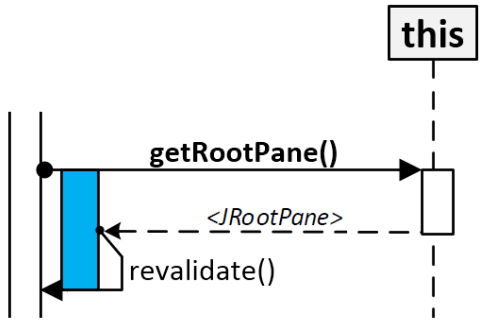


(b) Extended SD for the experimental group

Figure 5.4: Figures provided with task T2.4

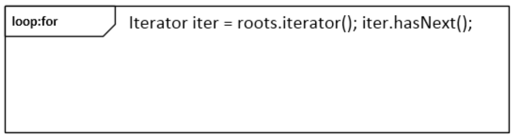


(a) Standard SD for the control group

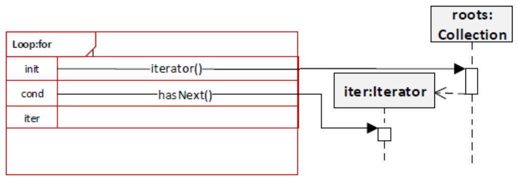


(b) Extended SD for the experimental group

Figure 5.5: Figures provided with task T4.1

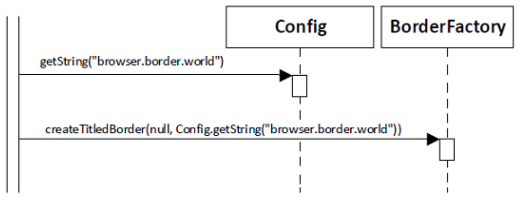


(a) Standard SD for the control group

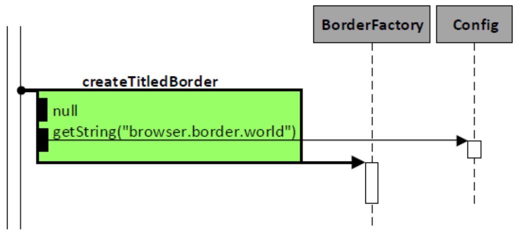


(b) Extended SD for the experimental group

Figure 5.6: Figures provided with task T4.2

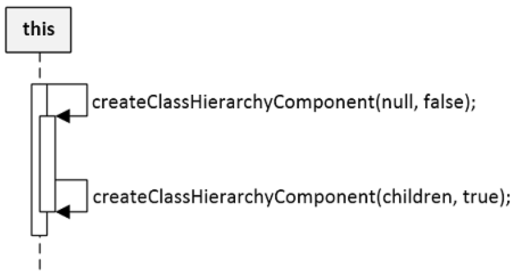

Config
BorderFactory
getString("browser.border.world")
createTitledBorder(null, Config.getString("browser.border.world"))

(a) Standard SD for the control group

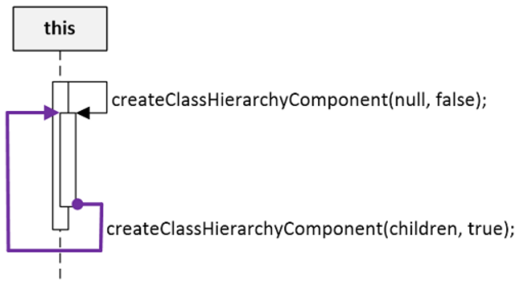


(b) Extended SD for the experimental group

Figure 5.7: Figures provided with task T4.3


this
createClassHierarchyComponent(null, false);
createClassHierarchyComponent(children, true);

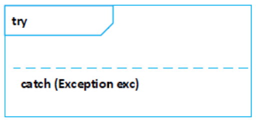
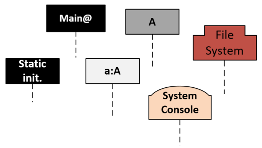
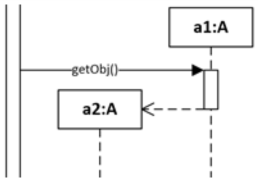
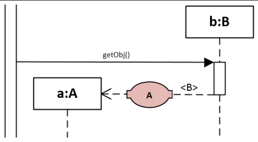
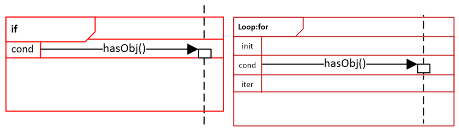
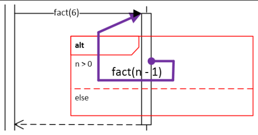
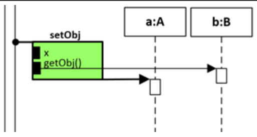
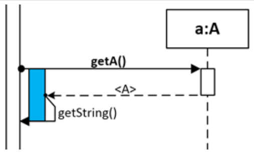
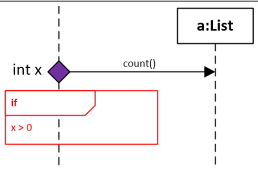
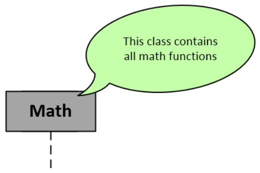
(a) Standard SD for the control group


this
createClassHierarchyComponent(null, false);
createClassHierarchyComponent(children, true);

(b) Extended SD for the experimental group

Figure 5.8: Figures provided with task T4.4

Table 5.3: The list of extensions to be evaluated in task T5

Extension ID	**Extension Name**	**Notation**
Ex1	Try-Catch fragment	
Ex2	Lifeline distinction fragment	 Main@ A File System Static init. a:A System Console
Ex3	Lifelines created from returned objects	
Ex4	Type casting	
Ex5	Calls inside fragmentâĂŹs operands	 if cond hasObj() Loop:for init cond hasObj() iter
Ex6	Recursive calls	
Ex7	Nested calls	
Ex8	Chained calls	
Ex9	Variables names and all their assignments	
Ex10	Javadoc comments of classes and methods	
Ex11	Events and event handling	Take into account that event-driven methods are usually executed when a certain action is fired. Therefore, they cannot appear in the sequence diagram in a specific order. Do you think that showing them will add value to the program understanding?

consists of 22 subjects. The collection of subjects is somewhat heterogeneous in that it represents five different nationalities and all kinds of students are working on different areas of computer science and software engineering.

Although it was recommended by Di Penta et al. [79] that "a subject group made up entirely of students might not adequately represent the intended user population", we were constrained by the time allotted and could not make the subjects more diversified. We hope in the future to extend our subjects to include professors and participants from the industry. Overall, all subjects participated as volunteers, which can thus be assumed that they were appropriately motivated. All of them have prior experience with the UML sequence diagram but none of them has previous knowledge about our extensions.

To partition the subjects in the two groups, we have considered their fields of expertise that can strongly influence the individual performance. They actually represent variables that can really affect the results and should be controlled during the experiment. Therefore, their distribution among the two groups was even and based on their knowledge of Java, software modeling, sequence diagrams and reverse engineering. Since all the undergraduate students were working on senior projects in software engineering, they were just evenly partitioned into two groups of 6 students, one as control and another as experimental. MS and PhD students have been divided based on their experience in software engineering and that was measured by the number and kind of courses they took from the software engineering program. This was actually assessed using informal questions asked to each subject in order to test his experience with sequence diagrams. Questions included asking about the courses they took, even in their master's degree or in their undergraduate studies, and whether they worked on projects related to sequence diagram modeling. Thus, they were equally assigned to the groups (i.e., one PhD and 4 MS students per each group). This ended up with having 11 students in each group, each of which contains 1 PhD, 4 MS and 6 BS students.

5.1.5 Experiment Procedure

The experiment was carried out through two sessions, each of which took place at a computer lab in the department at our university. Both sessions were conducted on workstations with similar specifications, i.e., all of them are of Intel Core i3 - 2.93 GHz CPUs, 4 GB RAM, and screen resolutions of 1440 x 900. In addition, the workstations were using the same Internet connection provided by the university network.

The first session involved the MS and PhD students of both groups while the second session was for the BS students. A 5-minute recall tutorial on sequence diagrams was given to both groups, highlighting the main notations of and how can they reflect Java code. In addition to that, we conducted a 10-minutes presentation showing our proposed extensions to the standard sequence diagram. Both sessions were supervised, allowing the subjects to pose clarification questions and preventing them from communication with each other.

The subjects were given links containing the online questionnaire along with other links for downloading the corresponding diagrams and the tutorial document. The links to the questionnaires were provided as publicly accessible links, meaning that the identity of the subjects cannot be recognized. The assignment was to complete the 13 comprehension tasks within around 35 minutes. During the whole duration of the sessions, we kept requiring the subjects to motivate their answers at all times. The subjects were advised to complete the tasks within the time allocated, recommending them to skip and write 'I could not identify the answer for this' for any question that might take more than 5 minutes. Finally, the questionnaire asks the subjects to write their opinions or any further recommendations regarding the experiment, our extensions, or any other related aspects.

5.1.6 Variables and Analysis

The availability of our extended notations in the experiment is regarded as the independent variable to the UML sequence diagram during all the tasks.

The first dependent variable is the time spent on each task and is measured by having the subjects to record the starting time of any new task. To effectively accomplish this, we have written a small script that can automatically get the system time and show it as text in each task page, allowing them to copy/paste or drag/drop it to a certain field on that web page. In addition, we have disabled the 'Back' button on each page to prevent the subjects from navigating back to earlier tasks.

The second dependent variable is the correctness of the given answers. This has been measured by applying our answer model to the subjects' responses, which specifies the required elements and the associated scores. For the other tasks that require users themselves to rate the complexity and precision of our extensions in representing program code in comparison with the standard representation, there were no answer models for such tasks.

To test our hypotheses, the sample distributions have firstly been tested via the $Kolmogorov-Smirnov$ test [80] to see whether they are normal. In addition, Another test, that is *Levene*'s test [81], was used to check whether sample distributions have equal variances. In general, in case these tests passed successfully, the Student's t-test is used to evaluate the hypotheses.

Following our alternative hypotheses, we employed the one-tailed variant of each statistical test. For the time as well as the correctness variables, a typical confidence level of 95 percent was maintained ($\alpha = 0.05$). The statistical package that we have used for our calculations is R version 3.2.2 (32-bit).

5.1.7 Pilot Studies

Before carrying out the experimental sessions, we conducted two pilot studies to refine several experimental parameters, such as the number and kind of tasks, their feasibility, clarity, and the amount of time would be required. The pilots for the control and experimental groups were performed by two MS students of the computer engineering department. Both did not join in the actual experiment.

The purpose of selecting students from an area that is somewhat outside software engineering is to get a helpful impression about the parameters mentioned above. Both pilots did not have the basic knowledge about sequence diagrams, although they said that they took a related course in their undergraduate studies a long time ago. Therefore, we have given a detailed tutorial, for around 30 minutes and we then got them working on the experiments.

The results of the pilots led to the elimination of three tasks. These tasks were either time consuming or not clear at all, even after live clarification during the experiment. In addition, the studies suggested changing one of the tasks from one category to another. Furthermore, the studies helped us to refine several tasks for the sake of making them clearer and easier to understand. Other than this, the tasks were found to be sufficiently feasible in both the standard and extended sequence diagrams. We have also utilized these studies to create a clear tutorial as a presentation to show 1) the basics of sequence diagrams, our extensions, and the types of tasks and how should they be responded to.

5.2 Results

Table 5.4 displays a set of descriptive statistics of the questionnaire results based on aggregated measurements over the eight tasks, which are basically based on grading users' answers and time spent.

Based on the individual results of each task, we have observed no outliers to be removed from the final results. However, as a key factor for both time and

Table 5.4: Statistics of the questionnaire results

	Time (in minutes)		**Correctness (in points)**	
	Standard SD	*Extended SD*	*Standard SD*	*Extended SD*
mean	23.81	17.81	14.40	26.80
difference		-25.20%		+86.11%
min	17.33	11.26	7	17
max	32.82	24.80	20	30
median	21.45	18.59	14	27
stdev.	5.72	5.10	3.37	3.85
Kolmogorov-Smirnov	0.594	0.597	0.070	0.005
Levene F		0.6405		0.7525
Student's t-test				
df		17.76		17.69
t		2.47		-7.66
p-value		0.0237		0.0001

correctness, we have noticed that two subject (one from each group) were not very interested in conducting the questionnaire as we have noticed that they did not respond to the provided tasks properly. For example, one of them has written some zeros as responses for some of the tasks of the category C2 that required writing code, while the other has entered similar rating values for all both criteria and both diagrams in the tasks of the category C4. Subsequently, we disregarded the entire input provided by these two particular subjects (i.e., we ended up with having responses of 10 subjects from the control group and 10 subjects from the experimental group).

5.2.1 Time Results

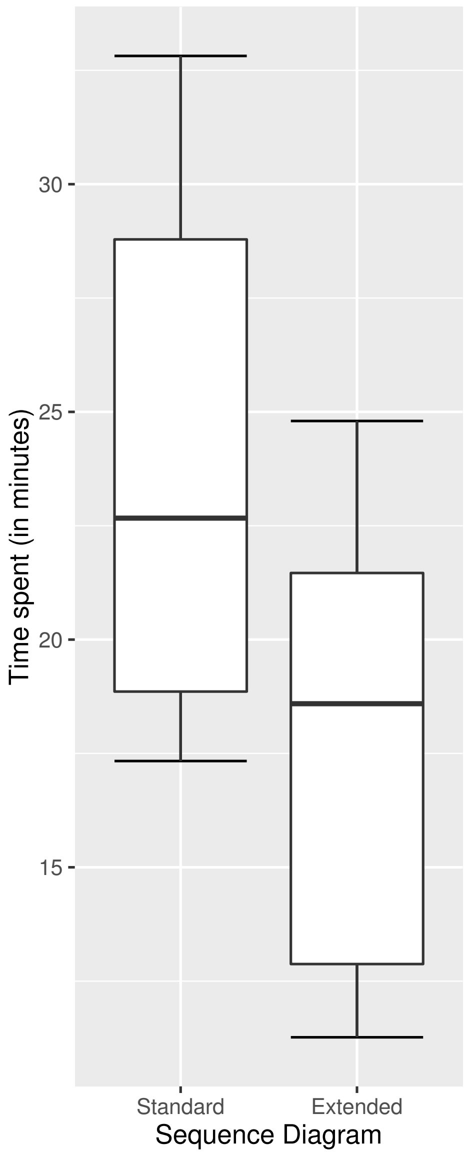
We have started by testing the null hypothesis $H1_0$ described in section 5.1.1 that stated that the time needed to complete comprehension tasks is not impacted by the availability of our proposed sequence diagram extensions. Fig. 5.9a shows the total time spent by the subjects on the first eight tasks using a box plot. It can be also indicated from Table 5.4 that, on average, extended diagram group required 25.20 percent less time.

The distributions of the samples are normal and they have equal variances as well. This has been was proven by the Kolmogorov-Smirnov and Levene tests, which have succeeded for the timing results shown in Table 5.4). This concludes that Student's t-test can be used to test $H1_0$. As presented in Table 5.4, a statistically significant result has been yielded from the t-test, which is represented by the p-value of 0.0237 that is less than 0.05. The average time spent by the extended sequence diagram group was visibly lower, which means that $H1_0$ can be rejected in support of the alternative hypothesis $H1$, implying that the use of our sequence diagram extensions decreases the time needed to achieve different comprehension tasks.

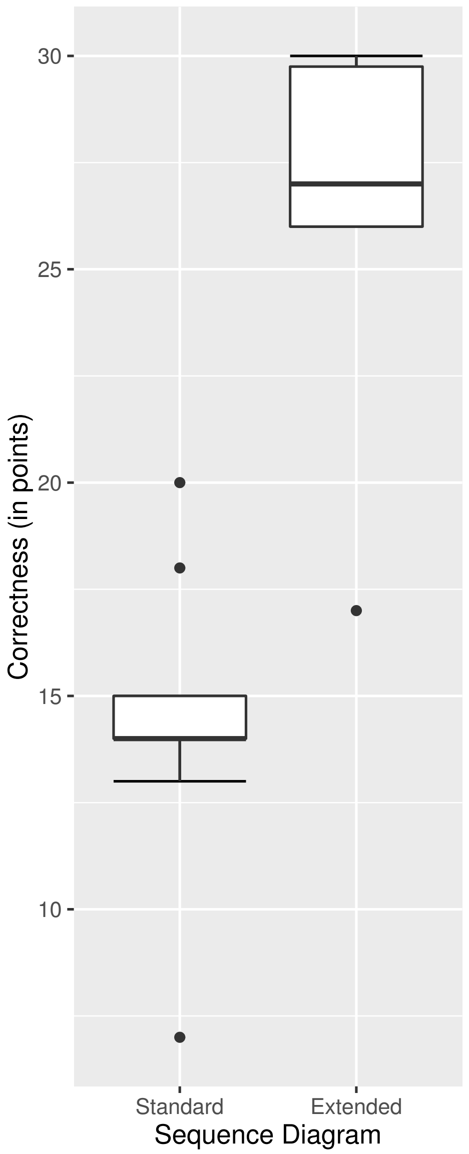
5.2.2 Correctness Results

Now, the null hypothesis $H2_0$ that states that the use of our sequence diagram extensions does not affect the accuracy of answers given during the comprehension tasks is tested.

Fig. 5.9b demonstrates the points obtained by the subjects on the first eight tasks by means of a box plot. Notice that we take into consideration the overall points rather than individual ones (Points per task are discussed in section 5.3.3). The correctness difference is obviously seen from the box plot, and is even more



(a)



(b)

Figure 5.9: Box plots for the overall time spent and correctness

pronounced than that for the timing results. Answers provided by the extended diagram-based subjects were more accurate by 86.11 percent (refer to Table 5.4), that is obtained through averaging 26.8 out of 32 points compared to 14.40 points for the standard diagram group.

Like the timing results, Table 5.4 also shows the results of the Student's t-test for response correctness, in which the requirements for the use of the t-test were met as well. The p-value of 0.0001 implies statistical significance, which means that $H2_0$ can be rejected in support of our alternative hypothesis $H2$, which states that the availability of our sequence diagram extensions enhances the correctness of answers provided throughout the conducted comprehension tasks.

5.3 Analysis and Discussion

This section presents our observations and conclusion of the results obtained from the experiment. It also justifies and elaborates the results of the overall performance, performance per task and performance per subject level.

5.3.1 Reasons for Different Time Requirements

There are several factors that contributed to the lower time requirements for the extended sequence diagram participants. First, most of program interactions and control flow constructs are explicitly represented using special and explicit notations, which helps in finding certain information by just having an outlook to the provided diagram. Users using the standard UML sequence diagram, on the other side, tended to look for certain pointers that might assist them inferring the locations of certain program information. Second, as most of the program information were either not, wrongly or inappropriately presented in the standard sequence diagram, users tended to search for answers to the questions even more than once in some portions of the diagram, which for sure results in having a cognitive load.

On the other hand, there might be several factors that led to having a negative impact on the time requirements of the users who used the extended sequence diagrams. The main important factor is the unfamiliarity of these extensions to the users as it was the first time for users to see such extensions. This has led to having the users requesting a copy of the tutorial presented while they were conducting the questionnaire. Therefore, referring to the tutorial for every particular SD extension in some of the questions contributed to spending a certain amount of time as overhead for recalling its meaning. This could be solved by incorporating the proposed extensions into standard UML as well as the tool that generate it.

5.3.2 Reasons for Response Accuracy Differences

We regard the added value of our proposed extensions for correctness to several factors. The style in which the new sequence diagram extensions have been designed almost talks about the code behind them. This means that, looking up a certain information, writing the representative code or locating the corresponding excerpt of a specific statement was a bit easier compared to the standard one. Second, as the main objective of the proposed extensions is to provide a comprehensive and precise overview of the program control flow using representative notations, users were confident and thus able to capture the correct answer of most of the provided questions. Finally, the debriefing questionnaire results shown in Table 5.4 show that the extended sequence diagram group used their assigned diagram most of the time. In other words, in some of the tasks where users could provide correct answers, they spent a bit more time to get that answer. This has further been reinforced by the users' ratings that indicate that the extensions were precise while they were somewhat complex at a certain degree in some cases.

5.3.3 Individual Task Performance

The main goal of our third research question is to enable us to identify the comprehension tasks that benefited most from using our proposed extensions to sequence diagrams. Therefore, we have examined the performance of the subjects per each task independently in more detail. Fig. 5.10 demonstrates, from a task perspective, the average time spent and points obtained by each group. Although our experiment composes thirteen tasks, only eight of them were considered in this particular evaluation. Others, of the categories C4 and C5, have a different kind of evaluation that is based on the opinion of the users themselves. Based on these results, we can notice that time spent and correctness are negatively correlated. This means that users who needed a relatively little effort could score relatively

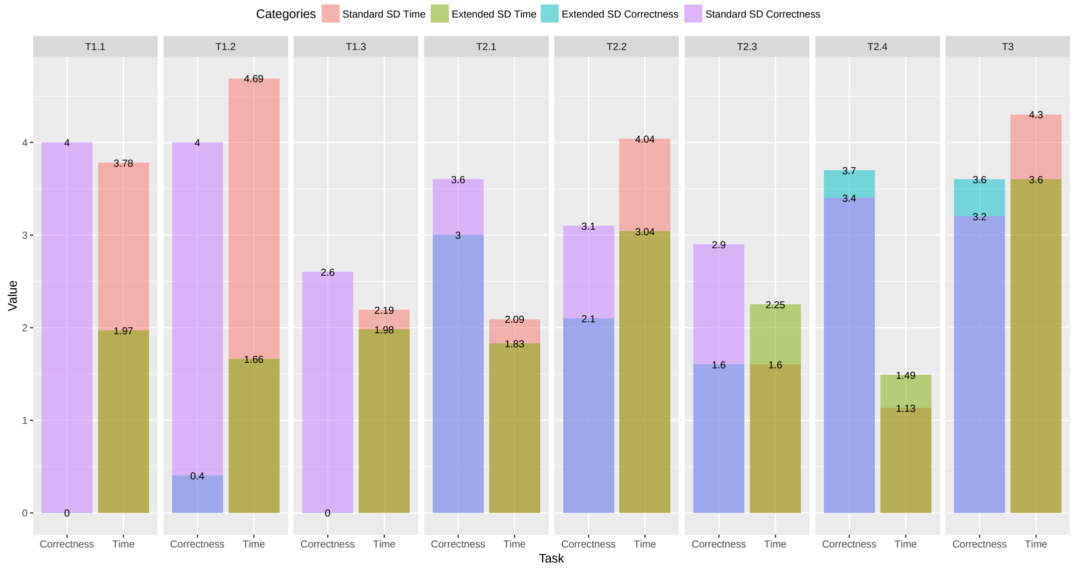


Figure 5.10: Time and correctness averages per task

high points, and vise versa.

5.3.3.1 Task T1.1

The main goal of the firstly introduced task was to identify all type-casting operations in which objects are converted from one type to another. The difficulty that the participants of the control group faced in this task is the fact that the standard UML sequence diagram does not contain a special notation for representing such operations. Therefore, they consumed too much time trying to identify where objects are returned of certain types and then created with lifelines of other types. At the end, almost all of them could not capture such operations in the standard sequence diagram they were given. On the other hand, since the extended sequence diagram has a special notation for such operations, almost all users of the experimental group were able to identify all such operations with a minimal amount of time. The significant difference between the time and points of the answers is obvious.

5.3.3.2 Task T1.2

Task T1.2 concerned a recognition of the methods that are recursively called during the communications between objects. Achieving this task was also easier using our new notations than that with the standard representation since it explicitly exposes all recursive messages using a meaningful notation. In this task, all experimental users could get the full points (i.e., 4 out of 4) and needed less than half the time required by the control group. The main reason of having only two subjects out of eleven who could catch the recursive methods is the fact that self and recursive messages are represented using the same notation in the UML standard of sequence diagrams. Therefore, such users consumed too much time tracking all self-messages for the sake of identifying whether they are being executed or not. Nevertheless, only a few of them could catch the recursive ones.

5.3.3.3 Task T1.3

This task was actually related to counting the number of messages fired inside the *try* block of the $try - catch$ construct. Again, our extended diagram contains a simple extension to the UML sequence diagram fragment. Our extension could utilize the fragment, with a certain color, to enclose all messages originated from the *try*, catch or finally blocks. This means that users of the experimental group could recognize that fragment and started counting the number of messages fired from the *try* block. Although their scores were relatively higher than the one of the control group, there was no significance difference in the time spent. The main cause of that is the shape similarity of the $try - catch$ fragment with other fragments representing other blocks, such as loops, conditions, and alternatives, which required users to investigate each fragment to check whether it relates to a *try* block. Another reason was with the counting strategy; some users were confused whether to only count the messages originated from the *try* block only or they should include the ones originated from the *catch* block. A similar confusion was with the messages created within the nested and chained calls. With respect to the control group, some users were searching for keys that can direct them to find messages inside a *try* block, by searching for an *Exception* class for example. Once they could not find any, they just answered with '0'. Others had the fact that the UML does not provide a specific notation for $try-catch$ and subsequently gave a '0' answer as well. Thus, the time spent in this task by the control group was closer to, but a bit higher than, that of the experimental group, but of course, all the answers provided by the standard diagram participants were not correct.

5.3.3.4 Task T2.1

This kind of tasks was concerned with the ability to recognize the code snippet that caused the generation of a certain excerpt of a sequence diagram. We came up with this kind of question to check whether our extension was simple enough in representing the control flow of programs. Based on the timing results, we can see that users spent less time in writing the code representing an excerpt of a standard sequence diagram compared with the users of the extended diagram, but again answers of the experimental group got higher scores. This is justified by the slight complexity of our proposed extensions compared with the simplicity of the standards. Since the standard representation was simple, users could immediately write the code but in a wrong way since the standard sequence diagram does not reflect the actual flow of the chained calls. Inversely, users used the extended diagram were able to recognize the correct flow of messages with the price of time that was almost spent for recalling what such a notation means by referring to the tutorial provided.

5.3.3.5 Task T2.2

This is another task of writing the code snippet that generated the excerpt of the diagram. As the flow of messages here was relying on chained and nested calls, the diagram excerpts of both the standard and extended diagrams were somewhat complicated. However, users of the experimental group could write the code faster and more correct than those of the control group. The time was 1.8 minutes less while the score was 1.1 points more.

5.3.3.6 Task T2.3

In this task, we clearly observe that the time spent by the experimental group was greater than that spent by the control one. The diagram excerpt used for this task was fairly simple using both: the standard and extended sequence diagrams. This caused the standard users responding faster but, due to the limitation of the UML sequence diagram in creating lifelines for objects once they are returned from a method call, most users could not recognize that the message provided in the excerpt returns an object to a named variable, which as a result led to wrong answers. On the other hand, the extended diagram users were able to identify the returned object and could answer the question better but with the price of time spent.

5.3.3.7 Task T2.4

The question here was also provided with a simple diagram excerpt, representing a conditional fragment with an operand containing a method call. Users of both groups could immediately recognize the representative code, but the one of the control group were a bit faster and got better scores as well. The extension in this context was a bit more complicated that the standard representation as it explicitly demonstrates the call inside the operand with a message connected to a lifeline. The standard diagram was simple because it uses only text for representing the operand with no matter what is inside it.

5.3.3.8 Task T3

This is the only task that represents the category C3. Here, subjects are provided with source code and requested to search for the portion of the supplied diagram representing that code snippet. Again, the diagram excerpt representing that code was relatively simpler and users go catch quickly. However, we can observe the significance of the time spent for this task compared with the other tasks, which actually is caused by having users to search, screenshot, save the snipped image, and then upload that image as a response to this task. However, we can see that it is less than that with the control group. We can see, on the other hand, that the answers provided by the experimental were less accurate in comparison with the standard group users, who could achieve better scores (only 0.3 more than the experimental group). While investigating the root cause of that, we observed that there was another excerpt of the diagram that is somehow identical to the one requested in this task. Everything was similar in those two excerpts except the name of one of the classes used as a parameter to one of the methods called.

5.3.3.9 Task T4.1

In this task and the other tasks of category C4, we allow users to give their evaluation for the complexity and precision of our extensions in representing certain program behavior. To this end, we have defined these two criteria as follows:

Complexity : this criterion refers to the number and shape of the elements used in each notation of the sequence diagram, either the standard or the extended, and the overall composition of such notations.

Precision : this criterion measures how precise is the diagram in representing the associated code snippet. This involves the capability of the diagram of covering all aspects of the given code, such as the types of messages, the composition of different messages, and lifeline interactions.

In all tasks of such a category, users are allowed to rate the provided diagrams against such criteria based on the code given. The rating is done on a scale of 0-4 (i.e., 0, 1, 2, 3 or 4). After collecting the results of all tasks of this kind, we have aggregated them using the median rather than the mean. This is because that the mean will not appropriately represent the overall complexity and criteria as it averages the inputs. Questions of this category are represented similarly to all participant included the control and experimental groups. Overall results are

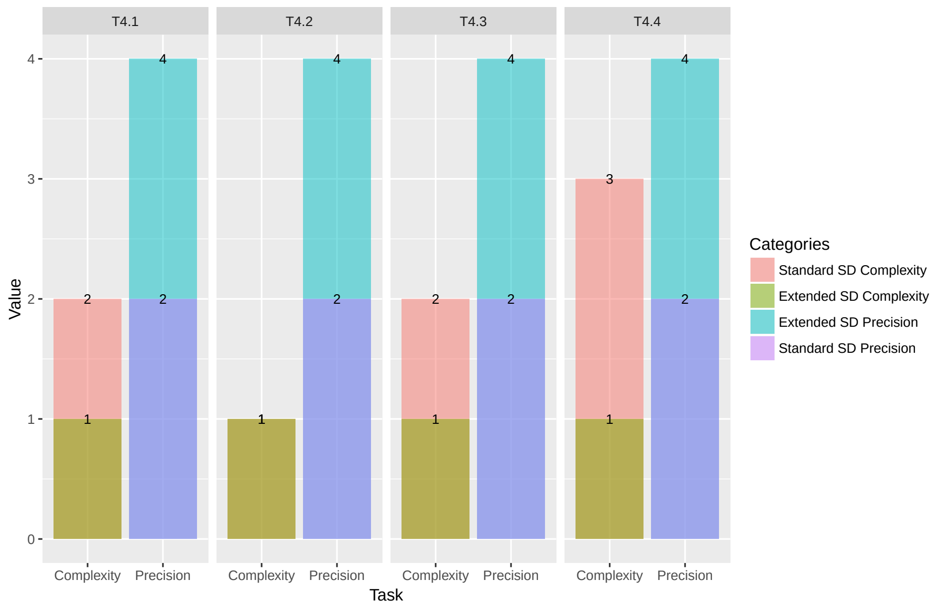


Figure 5.11: Average complexity and precisions per task for both: standard and extended sequence diagrams

also obtained from all participants as one chunk.

In particular, Task T4.1 comparatively measures the complexity and precision of the standard and extended sequence diagrams in representing chained calls. It can be seen from Fig. 5.11 that representing chained call using our proposed notation was twice precise than the notation used by the UML standard of sequence diagrams. In addition, in terms of complexity, the representation of chained calls is half complex than the representation using the standard sequence diagram.

5.3.3.10 Task T4.2

With respect to representing loops that contain method calls inside their specifications, our notation appeared to be similar to the standard representation. However, it was more precise in reflecting the number and kinds of the messages and lifelines would be used during the execution of that for loop.

5.3.3.11 Task T4.3

Regarding nested calls, it is also clear that our representation is more simple and precise at the same time. This is because that the standard sequence diagram complicates the representation by generating lots of separate messages related to the method calls used in the program, which consequently makes it represent the program as it appears in a different flow of control.

5.3.3.12 Task T4.4

Recursive calls notation was also one of the extensions that significantly simplified the representation of recursive methods and their corresponding calls. At the same time, it was precise as it rendered the actual flow of control of the given scenario.

5.3.3.13 Task T5

In this task, we just take the subjects' opinion about the extensions provided in the experiment. Control group users were asked whether the provided extensions would definitely help them answering the questions they already answered in a better way, somewhat or will not help them at all. Similarly, users of the experimental group were asked whether the proposed notations were advantageous while they are carrying out the experiment.

Results in Fig. 5.12 show that almost all proposed extensions to the sequence diagram were useful and effective for understanding the control flow of programs.

5.3.3.14 Summary

After a deep interpretation of the performances obtained per each individual task, we analytically generalize our discussion. From the results of all tasks of the category C1, namely T1.1, T1.2 and T1.3, it has become obvious that our extensions

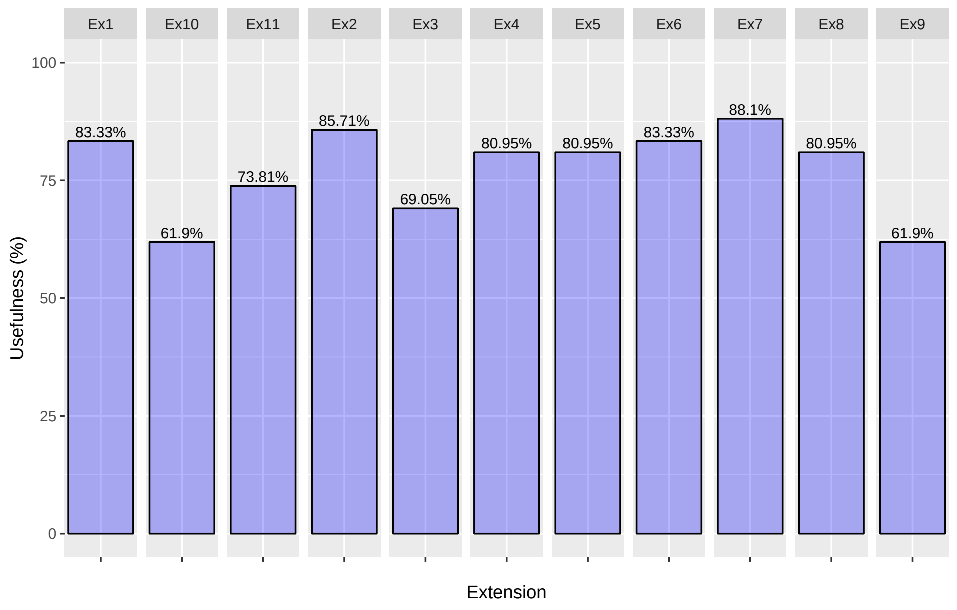


Figure 5.12: Average usefulness of each SD extension

to the standard sequence diagram are of great help in grasping the interactions executed throughout the program or even within a certain kind of control structures. Another conclusion is related to the UML standard of sequence diagram which was basically designed for forward software engineering and is not sufficient for reverse engineering.

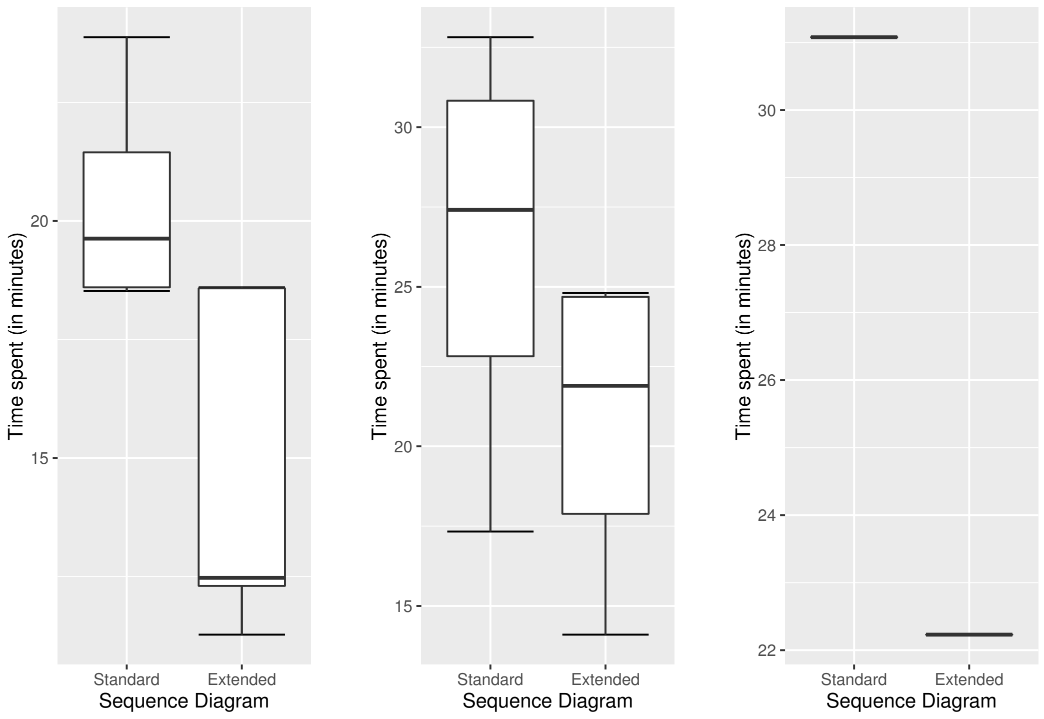
In addition, our sequence diagram extensions provide explicit representations of most of the constructs used to control the flow of programs. It is useful in a way that it represents program interactions such that users can visually distinguish patterns. We refer by patterns to every particular block of interactions, construct, operation or control of the program flow a program may employ. This indeed turns out to expose and derive more accurate information than could be obtained from the exhausting examination of these patterns from the standard UML sequence diagram by the users themselves.

For some of the tasks where the representation of program control flow was simpler using the standard sequence diagram, we can conclude that extensions should focus on reducing the complexity of the resulting diagrams while preserving the precision. This is actually a trade off and we could investigate more about such a trade-off through the use of the tasks of the category C4.

5.3.4 Individual Subject Experience Levels Performance

In order to address the concerns that may arise regarding the effect of the experience of the students used as subjects in this experiment, we have analyzed the performance of each level of students separately. In other words, we have aggregated the results of the PhD students, MS students, and BS students independently so that we can recognize which level of students had a significant impact on the overall results obtained, taking into account that there was only one PhD student per each group.

It can be shown from Fig. 5.13 and Fig. 5.14 that student level of study or experience does not significantly contribute to the change of the overall results. In each level, the mean time spent by students to respond to the given tasks is less for the experimental group than that for the control group. On the other hand, in each of the levels of students' study, scores gained by the participants of the experimental group were much greater than the scores obtained by the control group's participants. Since we have only one PhD student per each group, we cannot comment further on the results provided since the min, max, and mean



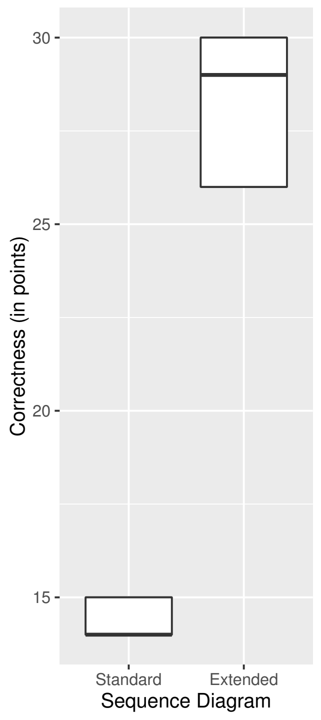
(a) BS students (b) MS students (c) PhD students

Figure 5.13: Box plots for the time spent per subject levels

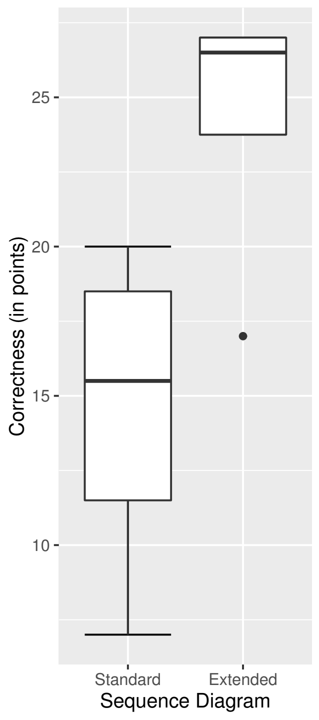
values are the same in this particular case.

From the individual results of the subject levels, we can notice that, in both groups, most of the BS students' results were above the mean in terms of time spent while the majority of MS students were under the mean. From a different perspective, the performance of almost all the BS students of the control group was low in terms of giving correct answers to the tasks questions while it is varied for MS students of the same group. However, both MS and BS students of the experimental groups were performing well in answering the questions based on the extended sequence diagram.

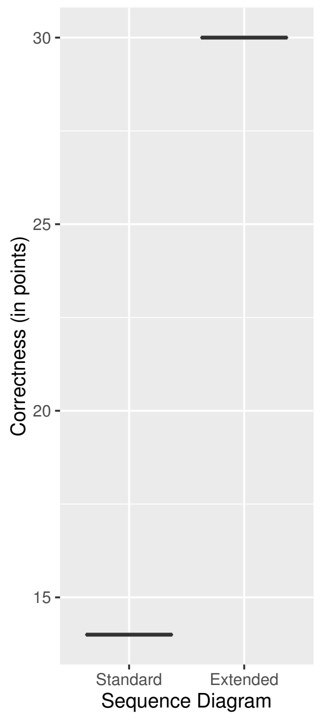
We also demonstrate the student's performance per each level of study in Fig. 5.15. This figure makes it obvious that the performance of the students of the



(a) BS students



(b) MS students



(c) PhD students

Figure 5.14: Box plots for the correctness per subject levels

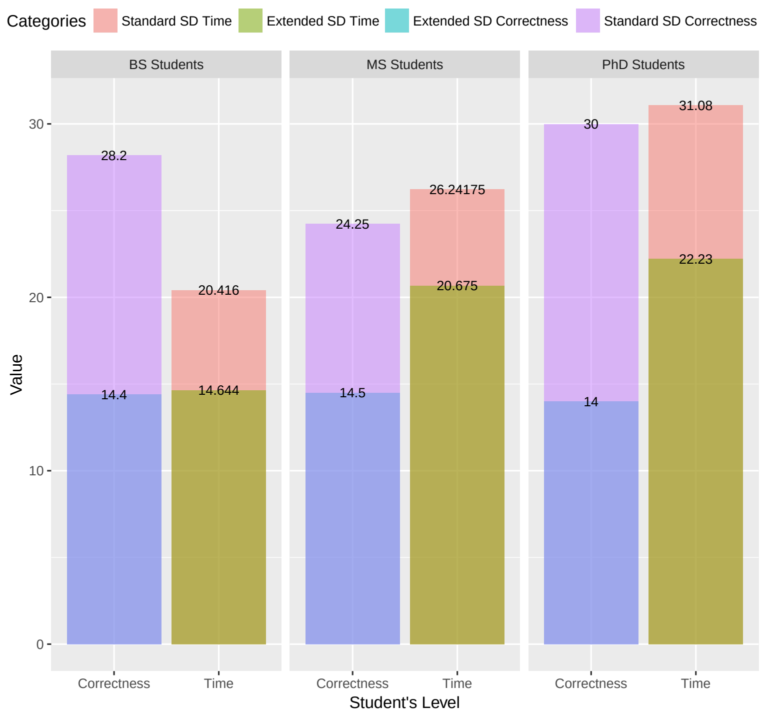


Figure 5.15: Averages per student's level of study or experience

different level of study is about similar in a way that the control group took more time than the experimental with fewer points for the answers given. We conclude that the level of study or experience of the subjects does not have a drastic impact on the overall results as long as they are assigned to both groups of the experiment based on their experience.

5.4 Threats to Validity

In this section, we discuss the validity threats and how we addressed them in our experiment. Such validity threats have been classified into two different categories, namely internal validity and external validity.

5.4.1 Internal Validity

This type of validity refers to the cause-effect inferences made throughout the analysis. It includes the threats related to subjects, tasks, and other variables.

5.4.1.1 Subjects

Since we did not conduct a detailed prior assessment of the subjects competence in the field of software engineering, they may not be adequately competent. To reduce this threat, we ensured that the distribution of the subjects to the two groups of the experiment is fair enough, which was based on their experience in software engineering in general and in sequence diagrams in particular. For this purpose, we firstly asked each subject from the MS and PhD level a set of informal questions in order to infer and measure their comparative experience in the field of the experiment. For undergraduate students, we did our best to select students from the same level (i.e., the senior level) and particularly the ones who are working in similar senior projects with the same supervisor. Second, we have designed a detailed tutorial that could give them an idea about the concept of sequence diagrams as well as our extensions. Then, this tutorial was provided to all of them to allow them to refer to it at any point they feel they might need it.

Another aspect of subject-driven threats is related to the different levels of study as well. A particular level of students can affect the overall results. Therefore, we have analyzed the results of each level to investigate whether there exists any level impacting the overall results.

From a different aspect, subjects may not have been properly motivated to participate in the experiment. This threat is mitigated by the fact that all of them participated as volunteers. Another threat might be related to the knowledge of participants about the objective of the experiment. Actually, most of them, especially the graduate ones, knew that were working on this project. We tried to alleviate this threat by encouraging them to provide fair answers as much as they can. We also told them that we were in the position of enhancing this work and that was why their fair answers, even if they were negative, would definitely help us to improve the quality of the work.

5.4.1.2 Tasks

We have designed the comprehension tasks used in this experiment ourselves. This can indicate that these tasks may have been biased toward our proposed extensions. To avoid this threat, we have designed them in a way that the difficulty of some tasks is distributed over the standard and the extended diagram. In other words, some of the tasks were easier to answer using the standard diagram than the extended one while some others were quite the opposite. Moreover, we have categorized the tasks so that some of the extensions are evaluated using more than one task.

Another threat that is related to task design is that they may have been too tough. This possibility was refuted by the use of pilot studies that enabled us to refine the presentation of the tasks and remove the ones considered very difficult or time-consuming. It could also be possible that subjects' answers were graded wrongly. This threat was mitigated by the use of a model answer that was used as a reference. In addition, the grading process was task-wise, which means that we grade only one task at a time for all the participants.

Another threat that is important to be addressed is the recording of the time spent by users on each particular task. Unfortunately, we could not find an online survey-building tool that provides this feature of keeping track of the time spent for each task (we could only find one tool that can supply timers for questions but it was prohibitively expensive to be used). We tackled this issue by having the online questionnaire (using SoGoSurvey) to be divided into several pages, were on each page, there exist only one question and a field for inserting the starting time of each task That time is automatically generated and shown in front of the users. Then, the task involves a question asking users to copy and paste it in its corresponding field. The 'Back' button has also been disabled. This could lead to another threat in a way that users might tamper with the time or insert incorrect values manually. It can also happen that the user refreshes the web page after spending a certain amount of time on a task and that will lead to generating a new starting time for that task, which will affect the timing of that task and the one that precede it. This should be managed in the future by designing our own survey from scratch and embedding it with hidden timers that can record the time spent for each task in the background.

5.4.1.3 Miscellaneous

Time constraints may have influenced the accuracy of the subject's responses to the tasks. To resolve this, the pilot studies were also useful here as it helped us to estimate the maximum required time for the whole experiment since the participant in such studies were inexperienced with sequence diagrams as they are from the computer engineering department. In addition, we allowed users to slightly exceed (i.e., by a maximum of five minutes) the experiment time that was initially set to be 35 minutes.

Furthermore, our statistical analysis may not have been completely accurate due to having two students with empty-like answers or similar rating points. In order to escape from this threat, we have removed the responses of these two students on all tasks from our analysis.

Another threat to validity could be the way in which the two different diagrams, the standard and the extended, have been deployed to the subjects. This threat was reduced by reproducing such diagrams using the same tool, that is Visio, and exporting them as PDF files. This allowed subjects of both groups to make use of the capabilities provided by the acrobat reader that was installed on the lab machines (of the same version), such as zooming and searching.

5.4.2 External Validity

External threats to validity are concerned with the possibility of generalizing the results to different contexts, and the limited representativeness of the tasks, the subjects and the use of $Greenfoot$ as an object.

With respect to the subjects, the use of a different kind or a large number of participants, such as professionals from the industry or more students of various levels could be a possible threat. Unfortunately, it was quite difficult to invite more than this number of students to the experiment as it was conducted the week before the week of the final exams of the semester. On the other hand, inviting people from industry is also a big issue as they always concerned about their time and how to spend it efficiently. We plan to extend the number of participants in our evaluation of the proposed extensions using another extended controlled experiment.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This work has investigated the possibility of enhancing the comprehension of program interactions using a better visualization approach. After studying the related work in the literature, we observed that most techniques in this context focused on reverse engineering of sequence diagrams using proper program analysis method, such as static analysis, dynamic analysis, or both together. We carried out a deep analysis of the state-of-the-art techniques and identified the limitations and gaps within which, which are concerned with the way sequence diagrams are represented and the amount of information to be presented.

Accordingly, we have defined specific research questions that address the main research problems in program analysis and comprehension that we ultimately aim to address throughout this work. Then, our objectives have been identified along with a detailed articulation of the contributions ad their potential outcomes that can support the fulfillment of these objectives.

The essential contribution of this work is the development of a static program analysis technique that intend to improve the understandability of program interactions and control flow. Our technique composes three major processes: program information extraction, interactions tracing, and trace visualization. The core of our visualization is demonstrated as a set of extensions to the UML notation of sequence diagrams that can carry more information about program interactions and control flow and present them in convenient manner.

The technique has been deployed as tool prototype and applied to a case study, namely Greenfoot, to validate its performance and precision of the produced results. The diagram representing a single scenario of Greenfoot, called $ClassBrowser$, has particularly been selected for the evaluation of the proposed extensions in terms of efficiency and effectiveness towards program comprehension, compared with the standard representation of sequence diagrams as a benchmark. Results obtained were promising and indicated that most of our extensions to the sequence diagram were simple and so useful in comprehending programs in less time with a precise understanding.

6.2 Future work

Several ideas have arisen throughout our work in this thesis, but due to the time limit, we could not go deep and accomplish every particular idea. Therefore, we would like to recommend working on them in the future as their employment would further increase the understandability of programs.

6.2.1 More extensions to the UML sequence diagram

- **Events and event handlers:** GUI-based interactions should be visualized in sequence diagrams to allow users to identify the interactions among the execution of GUI-based applications. Current standard UML-notations do not support the identification of the different events that can be executed at the program runtime. For example, button-pressing, mouse-clicking, window-resizing and many other events can trigger a call for a set of program interactions. Characterizing such events along with their sources in the sequence diagram would increase the comprehensibility of any GUI-based software. Keep in mind that all interactions that can be triggered by GUI events are considered as inactive when static analysis is used. Therefore, it is important to employ dynamic analysis techniques in order to effectively support such an extension.

- **Errors in the source code:** The ability of a reverse engineering technique to extract program information from programs that contain semantic errors would make it more preferable by the users who would like to debug and trace their programs visually. For instance, if a program contains a call to a method that is not defined or from undeclared object, then this call should be, in a way or another, represented in the resulting diagram by using a message line ending with a red cross, for example. We recommend adding such a functionality to current techniques of program visualization as it may add a value to program understandability.

6.2.2 More extensions to the program information collector and interaction tracer

- **Enriching SDs with dynamic information:** As we know, static analysis misses most of the dynamic interactions that can only be obtained through the program at runtime. Therefore, to enrich sequence diagrams produced by our technique, one needs to incorporate dynamic analysis and is best to be accomplished through instrumentation. This additional kind of analysis can capture the interactions information that the static analysis misses. Our suggestion in this matter is to apply aspect-oriented constructs that can facilitate the monitoring of all program method calls and perform proper information logging. In particular, for Java-based programs, AspectJ *pointcuts* and *advices* can achieve this job perfectly.

- **Combining dynamic and static information of SDs:** One of the main challenges of applying hybrid analysis of programs is how to merge the information collected through dynamic analysis with the ones gathered through static analysis. To this end, we hope in the future to address such a challenge using the following procedure:

- Apply the static analysis first, then the dynamic one.
- Each element gathered through the static analysis should have its own signature without any conflicts stored in an XMI representation.
- Once the dynamic analysis starts working, a signature of each element should be captured using the same way and format used in the static analysis.
- For each element collected through the dynamic analysis, apply an XPath query to get its corresponding elements in the static XMI representation. If the query returned an element, attach the new information with the existing ones. Otherwise, the element is considered as it is no longer be collected through the static analysis, which leads to create a new record for this particular element in the XMI representation (with proper location in the file) aligned with its signature and information.

6.2.3 More extensions to the visualization tool

- **Visual Re-engineering:** In addition to the navigation to the source code, the visualization tool may provide a facility that allows users to re-engineer programs visually. This means that they may change method calls, objects, or parameters as well as duplicate call messages and interchange the order of them. All such operations, and maybe more, should be provided through the visualization tool where the changes made are reflected to the corresponding source code. In addition, users may be supplied with the facility that enales them to add normal comments or Javadoc comments in the layout if they are not available or not clear enough in the source code.

- **Live SD enrichment:** With respect to combining dynamic and static analyses, we suggest that the resulting sequence diagram immediately appears after performing the static analysis. After that, the program is executed and the dynamic analysis can launch at that time. Throughout the program runtime, all information collected should be rendered on the already displayed sequence diagram (supposing that the program and SD layout are cascaded in the screen). This means that users will be able to observe the newly added information to the sequence diagram in a live manner.

REFERENCES

[1] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering.* ACM, 2000, pp. 47–60.

[2] E. J. Chikofsky, J. H. Cross *et al.*, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[3] L. C. Briand, "The experimental paradigm in reverse engineering: Role, challenges, and limitations," in *WCRE'06. 13th Working Conference on Reverse Engineering, 2006.* IEEE, 2006, pp. 3–8.

[4] E. Korshunova, M. Petkovic, M. van den Brand, and M. R. Mousavi, "CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code," in *13th Working Conference on Reverse Engineering 2006 (WCRE'06).* IEEE, 2006, pp. 297–298.

[5] 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*, vol. 32, no. 9, pp. 642–663, 2006.

[6] N. Nystrom, M. R. Clarkson, and A. C. Myers, “Polyglot: An extensible compiler framework for Java,” in *Compiler Construction.* Springer, 2003, pp. 138–152.

[7] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, “A systematic survey of program comprehension through dynamic analysis,” *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide.* Pearson Education India, 1999.

[9] A. Rountev and B. H. Connell, “Object naming analysis for reverse-engineered sequence diagrams,” in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 254–263.

[10] “OMG, UML 2.0 Infrastructure Specification, Object Management Group,” http://www.omg.org, accessed on 2015-04-13.

[11] A. Rountev, O. Volgin, and M. Reddoch, “Static control-flow analysis for reverse engineering of UML sequence diagrams,” in *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1. ACM, 2005, pp. 96–102.

[12] A. Von Mayrhauser and A. M. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[13] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *Proceedings of the 13th International Workshop on Program Comprehension, 2005. IWPC 2005.* IEEE, 2005, pp. 181–191.

[14] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 341–355, 2011.

[15] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance, 2001.* IEEE, 2001, pp. 602–611.

[16] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *Proceedings of the 11th Working Conference on Reverse Engineering, 2004.* IEEE, 2004, pp. 70–79.

[17] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 551–560.

[18] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing trace visualizations for program comprehension through controlled experiments," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension.* IEEE Press, 2015, pp. 266–276.

[19] V. Garousi, L. C. Briand, and Y. Labiche, "Control flow analysis of UML 2.0 sequence diagrams," in *Model Driven Architecture–Foundations and Applications.* Springer, 2005, pp. 160–174.

[20] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," in *Proceedings of the 10th Working Conference on Reverse Engineering. IEEE Computer Society.* IEEE, 2003, p. 57.

[21] T. Systä, K. Koskimies, and H. Müller, "Shimba−an environment for reverse engineering Java software systems," *Software: Practice and Experience*, vol. 31, no. 4, pp. 371–394, 2001.

[22] A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014.* Springer, 2015, pp. 113–122.

[23] X. Li and J. Lilius, "Timing analysis of UML sequence diagrams," in *Lecture Notes in Computer Science*, vol. 99. Springer, 1999, pp. 661–674.

[24] H. Gomaa, "Designing concurrent, distributed, and real-time applications with UML," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE).* IEEE Computer Society, 2001, pp. 737–738.

[25] J. Odell, H. V. D. Parunak, and B. Bauer, "Extending UML for agents," *Ann Arbor*, vol. 1001, p. 48103, 2000.

[26] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring, "Kieker: Continuous monitoring and on demand visualization of Java software behavior," in *Proceedings of the IASTED International Conference on Software Engineering.* ACTA Press, 2008.

[27] T. Souder, S. Mancoridis, and M. Salah, "Form: A framework for creating views of program executions," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01).* IEEE Computer Society, 2001, p. 612.

[28] M. Zhu, H. Wang, Y. Cao, Z. Wang, and W. Jin, "The analysis of sequence diagram with time properties in qualitative and quantitative aspects by model transformation," in *2010 17th Asia Pacific Software Engineering Conference (APSEC).* IEEE, 2010, pp. 118–126.

[29] B. Bannour, C. Gaston, and D. Servat, "Eliciting unitary constraints from timed sequence diagram with symbolic techniques: application to testing," in *2011 18th Asia Pacific Software Engineering Conference (APSEC).* IEEE, 2011, pp. 219–226.

[30] "UML Profile for Schedulability, Performance, and Time Specification, SPTP/1.1," OMG Adopted Specification, 2015-10-22.

[31] B. P. Douglass, "Real time UML," in *7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2002,Co-*

sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002. Proceedings, vol. 2469. Springer, 2003, p. 53.

[32] D. B. Petriu and M. Woodside, "A metamodel for generating performance models from UML designs," in *UML 2004– The Unified Modeling Language. Modeling Languages and Applications.* Springer, 2004, pp. 41–53.

[33] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09), 2009.* IEEE, 2009, pp. 287–294.

[34] J. Jürjens, "Towards development of secure systems using UMLsec," in *Fundamental approaches to software engineering.* Springer, 2001, pp. 187–200.

[35] B. Bannour, J. Escobedo, C. Gaston, P. Le Gall, and G. Pedroza, "Designing sequence diagram models for robustness to attacks," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 2014, pp. 26–33.

[36] J. Jurjens, "Secrecy-preserving refinement," in *FME 2001: Formal Methods for Increasing Software Productivity.* Springer, 2001, pp. 135–152.

[37] A. van den Berghe, R. Scandariato, K. Yskout, and W. Joosen, "Design notations for secure software: a systematic literature review," *Software & Systems Modeling*, pp. 1–23, 2015.

[38] M. J. Pacione, M. Roper, and M. Wood, "A comparative evaluation of dynamic visualisation tools," in *20th Working Conference on Reverse Engineering (WCRE), 2013.* IEEE Computer Society, 2003, pp. 80–89.

[39] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research.* IBM Press, 2004, pp. 42–55.

[40] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis.* Springer, 2015.

[41] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications.* Springer, 2015, pp. 581–591.

[42] J. E. Grass, "Object-oriented design archaeology with CIA++," *Computing Systems*, vol. 5, no. 1, pp. 5–67, 1992.

[43] D. Myers, M.-A. Storey, and M. Salois, "Utilizing debug information to compact loops in large program traces," in *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010.* IEEE, 2010, pp. 41–50.

[44] Y.-G. Guéhéneuc and T. Ziadi, "Automated reverse-engineering of UML v2.0 dynamic models," in *proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering.* Citeseer, 2005.

[45] L. Lu and D.-K. Kim, "Required behavior of sequence diagrams: Semantics and conformance," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, p. 15, 2014.

[46] T. J. Grose, G. C. Doney, and S. A. Brodsky, *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, 2002, vol. 21.

[47] T. Ball, "The concept of dynamic analysis," in *Software Engineering - ESEC/FSE'99*. Springer, 1999, pp. 216–234.

[48] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[49] J. S. Carr and B. T. Kachmarck, "Generating module stubs," Aug. 25 2015, uS Patent 9,117,177.

[50] S. Jayaraman, B. Jayaraman *et al.*, "Towards program execution summarization: Deriving state diagrams from sequence diagrams," in *Seventh International Conference on Contemporary Computing (IC3), 2014*. IEEE, 2014, pp. 299–305.

[51] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI)," in *Software Visualization*. Springer, 2002, pp. 176–190.

[52] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework," in *Technical Reports by Department of Computer Science*. Kiel University, Germany, 2009.

[53] Y. Watanabe, T. Ishio, Y. Ito, and K. Inoue, "Visualizing an execution trace as a compact sequence diagram using dominance algorithms," *Program Comprehension through Dynamic Analysis*, p. 1, 2008.

[54] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection," in *Companion of the 30th International Conference on Software Engineering.* ACM, 2008, pp. 969–970.

[55] T. Ziadi, M. A. A. Da Silva, L.-M. Hillah, and M. Ziane, "A fully dynamic approach to the reverse engineering of UML sequence diagrams," in *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2011.* IEEE, 2011, pp. 107–116.

[56] Y. Labiche, B. Kolbah, and H. Mehrfard, "Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams," in *29th IEEE International Conference on Software Maintenance (ICSM), 2013.* IEEE, 2013, pp. 130–139.

[57] S. Lamprier, N. Baskiotis, T. Ziadi, and L.-M. Hillah, "CARE: a platform for reliable Comparison and Analysis of Reverse-Engineering techniques," in *18th International Conference on Engineering of Complex Computer Systems (ICECCS), 2013.* IEEE, 2013, pp. 252–255.

[58] H. Aloulou, "Dérivation de diagrammes de séquence uml compactes à partir de traces d'exécution en se basant des heuristiques." *Theses - FAS - Department*

of Computer Science and Operations Research, University of Montreal, 2016.

[59] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 367–377.

[60] D. Amalfitano, A. R. Fasolino, A. Polcaro, and P. Tramontana, "The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 41–57, 2014.

[61] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," in *Principles of Software Evolution, Eighth International Workshop on.* IEEE, 2005, pp. 148–151.

[62] K. Koskimies and H. Mossenbock, "Scene: Using scenario diagrams and active text for illustrating object-oriented programs," in *Proceedings of the 18th International Conference on Software Engineering, 1996.* IEEE, 1996, pp. 366–375.

[63] H. Mössenböck and N. Wirth, "The programming language Oberon-2," *Structured Programming*, vol. 12, no. 4, pp. 179–196, 1991.

[64] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systa, "Constructing usage scenarios for API redocumentation," in *15th IEEE International Conference on Program Comprehension (ICPC'07), 2007.* IEEE, 2007, pp. 259–264.

[65] E. Mäkinen and T. Systä, "MAS—an interactive synthesizer to support behavioral modelling in UML," in *Proceedings of the 23rd International Conference on Software Engineering.* IEEE Computer Society, 2001, pp. 15–24.

[66] "The Eclipse Foundation, Atlas Transformation Language," https://eclipse.org/atl/, accessed on 2015-04-22.

[67] T. Systa, "On the relationships between static and dynamic models in reverse engineering java software," in *Proceedings. Sixth Working Conference on Reverse Engineering, 1999.* IEEE, 1999, pp. 304–313.

[68] H. Grati, H. Sahraoui, and P. Poulin, "Extracting sequence diagrams from execution traces using interactive visualization," in *17th Working Conference on Reverse Engineering (WCRE), 2010.* IEEE, 2010, pp. 87–96.

[69] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: Elements of reusable object-oriented software.* Reading: Addison-Wesley, 1995.

[70] "XPath Tutorial - W3Schools," www.w3schools.com/xsl/xpath_intro.asp, accessed on 2015-07-4.

[71] "ArgoUML," https://sourceforge.net/projects/argouml, accessed on 2015-10-05.

[72] "Altova UModel," www.altova.com/umodel.html, accessed on 2015-11-30.

[73] "Enterprise Architect," www.sparxsystems.eu, accessed on 2015-09-19.

[74] "Modelio," https://www.modelio.org, accessed on 2015-11-11.

[75] "StarUML," staruml.io, accessed on 2015-10-08.

[76] "Trace Modeler," www.tracemodeler.com, accessed on 2015-09-27.

[77] "Visual Paradigm," https://www.visual-paradigm.com, accessed on 2015-09-19.

[78] "Greenfoot, by the Programming Education Tools Group," www.greenfoot.org, accessed on 2016-02-2y.

[79] M. D. Penta, R. K. Stirewalt, and E. Kraemer, "Designing your next empirical study on program comprehension," in *15th IEEE International Conference on Program Comprehension (ICPC'07), 2007.* IEEE, 2007, pp. 281–285.

[80] F. J. Massey Jr, "The Kolmogorov-Smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[81] H. Levene, "Robust tests for equality of variances," *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, vol. 2, pp. 278–292, 1960.

Vitae

Personal details:

- Name: Taher Ahmed Mohammed Ghaleb
- Nationality: Yemeni
- Date of Birth: 01/01/1984
- Personal Email: *taher.a.ghaleb@gmail.com*
- Permanent Address: Yemen - Taiz - Almoroor, AlManakh neighborhood

Education, Research, and Experience:

- BS degree in Information Technology from Taiz University, Yemen (September 2003 - July 2008).
- Research Interests: Programming languages, extensible compilers, program analysis, program comprehension, software modeling, aspect-oriented programming, and Database systems.
- Academic Experience: Teaching Assistant, Taiz University, Yemen (December 2008 - December 2011).
- Published conference papers and journal articles are listed in this link: `http://orcid.org/0000-0001-9336-7298`
- Mastered Programming Languages: C, C++, Java, AspectJ, HTML, JavaScript, ASP.NET, PHP, VC++, VB.NET, VC#.NET, ORACLE (SQL, PL/SQL, and Developer).