Towards Efficient Instrumentation for Reverse-Engineering Object Oriented Software through Static and Dynamic Analyses

by

Hossein Mehrfard

A dissertation submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

The Ottawa-Carleton Institute for Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

In software engineering, program analysis is usually classified according to static analysis (by analyzing source code) and dynamic analysis (by observing program executions). While static analysis provides inaccurate and imprecise results due to programming language's features (e.g., late binding), dynamic analysis produces more accurate and precise results at runtime at the expense of longer executions to collect traces.

One prime mechanism to observe executions in dynamic analysis is to instrument either the code or the binary/byte code. Instrumentation overhead potentially poses a serious threat to the accuracy of the dynamic analysis, especially for time dependent software systems (e.g., real-time software), since it can cause those software systems to go out of synchronization. For instance, in a typical real-time software, the dynamic analysis result is correct if the instrumentation overhead, which is due to gathering dynamic information, does not add to the response time of real-time behaviour to the extent that deadlines may be missed. If a deadline is missed, the dynamic analysis result and the system's output are not accurate.

There are two ways to increase accuracy of a dynamic analysis: devising more efficient instrumentation and using a hybrid (static plus dynamic) analysis. A hybrid analysis is a favourable approach to cope with the overhead problem over a purely dynamic analysis. Yet, in the context of reverse engineering source code to produce method calls dynamic and hybrid instrumentations typically lead to large execution traces and consequently large execution overhead.

This thesis is a step towards efficient and accurate information collection through a hybrid analysis procedure to reverse engineer source code to produce method calls, with the prime objective to reduce instrumentation overhead. To that aim, the first contribution of this thesis is to systematically analyze the contribution to instrumentation overhead of different elements of an existing and promising hybrid solution. Then, a second contribution of the thesis is to suggest an instrumentation optimization process with a range of different designs for those elements to reduce the overhead and select the best one for each element to optimize that solution. The resulting optimized hybrid technique, our third contribution, which potentially produces more accurate instrumentation compared to that hybrid solution for multi-thread software by reducing execution overhead by three quarters, has a reasonable efficiency to reverse engineer programs to produce method calls for multi-threaded software. A final contribution of this thesis is to suggest a set of recommendations for efficient instrumentation.

Acknowledgements

First and foremost, thanks to the God, for shedding lights throughout my life. The God who resembles to nothing and falls beyond any imagination.

I would like to express my sincere gratitude to my Ph.D. advisor, Prof. Yvan Labiche for the continuous technical and financial support during the course of my Ph.D. Your professionalism and decency have always been exemplary for me. I learned a great deal from you during my Ph.D. study, especially your attention to details finally taught me to how to craft a perfect software engineering experiment. Thank you Yvan for making my Ph.D. an exciting experience.

I would like to thank my Ph.D. committee members, Prof. Guéhéneuc, Prof. Lethbridge, Prof. Franks, and Prof. Baysal for taking their time and reading my thesis. Your insightful comments and different perspectives improved the quality of this manuscript.

My special thanks to my fiancé, Sara. Your love and emotional support motivated me to finish this work. Thank you Sara.

My deep gratitude to my parents, Masoud and Mojgan, for their unconditional love and constant support, I owe it all to you. I am grateful to my brothers, Ali and Ehsan, who have supported me along the way.

I would like to appreciate the SCE staffs, especially Jerry Buburuz and Daren Russ for facilitating my experiments and helping me with the last minute favours.

My appreciation also goes to my friends at Carleton university Ehsan Rahimi and Alireza Sharifian for making my Ph.D. an enjoyable experience. Also, I am grateful to my friend Alireza Salimi for the stimulating discussions and his help on my Ph.D.

Last but not the least, I would like to thank my fellow labmates in SQUALL laboratory for the constructive discussions and their feedback, and for the sleepless nights we were working together before deadlines.

Table of Contents

List of Tables

List of Figures

1 Introduction

In software engineering, program analysis usually takes place either through static analysis or through dynamic analysis. Static analyses refer to analyzing a program to obtain information without executing the program by building an abstracted model of a program based on its source code. On the other hand, dynamic analyses happen during program executions to capture information by intercepting program at runtime.

Static analyses can occur at different program levels (e.g., source code, Java byte-code, binaries) for different purposes (e.g., debugging, program comprehension, reverse engineering, formal verification). Depending on the goal of static analysis, source code is evaluated against a set of properties. These properties are either trivial or non-trivial. A property is nontrivial if there exist a program to satisfy that property and a program that does not. (A property is trivial if it holds true for all programs or holds false for all programs.) For example, the fact that a program does not dereference any null pointer is a nontrivial property, and a specific method name in a program is a trivial property when reverse engineering to produce a class diagram. In general, according to Rice's theorem [1], it is undecidable to statically analyze source code for a nontrivial property: Rice's theorem is a subset of the halting problem. Static analyses usually build an abstract representation (e.g., abstract syntax tree) of source code to analyze properties of interest. In other words, instead of evaluating properties of interest against a program, static analysis techniques usually use abstraction functions to build an abstract model of program states and evaluate properties against this abstract model. Therefore, analyzing properties based on the abstract model can cause approximation in the result of static analysis (over-approximation or under-approximation).

Static analysis is relatively sound and precise regardless of all possible program inputs and behaviour [2]. For a nontrivial property, the soundness ensures that a static analysis technique reports on all conditions that the nontrivial property holds true in the program, but it may also report on conditions that the nontrivial property is not true (i.e., over-approximation which contains false positives). The precision ensures that any condition reported by static analysis holds true for the same nontrivial property, but there is no guarantee that all conditions are reported (i.e., under-approximation which contains false negatives). Usually static analysis tools favour soundness over precision to provide coverage for all execution paths.

To illustrate these notions, we use the case of reverse engineering a class diagram (structural information). The constituents of a UML class diagram are composed of a set of trivial and nontrivial properties. Many tools (e.g., Papyrus[1], MoDisco[2], ArgoUML[3]) rely on static analysis to recover the class diagram. With the static analysis of a program source code, it is possible to recover many constituents of the UML class diagram syntactically or semantically (e.g., class, interface, enumeration, aggregation). All trivial properties (e.g., class name or method name) and some of nontrivial properties (e.g., inheritance) can be recovered soundly and precisely. On the other hand, there are constituents (i.e., nontrivial properties) in the UML class diagram (e.g., multiplicity) that static analysis can capture, though not as accurately as the constituent created during design. Among class diagram constituents, only a few cannot always be recovered (such as composition) or can be recovered with approximation (such as multiplicity) in static analysis. For instance, considering a two-to-four multiplicity in design, a sound static

[1] http://www.eclipse.org/papyrus/
[2] http://www.eclipse.org/MoDisco/
[3] http://argouml.tigris.org/

analysis technique may return a many-to-many multiplicity, while a precise static analysis technique may return a one-to-one multiplicity. We argue that the use of dynamic analysis in those cases can help to recover those constituents more precisely. In fact, there are works taking advantage of combining static and dynamic analyses to recover a class diagram more precisely [3, 4].

Although a static analysis technique can produce a sound and slightly less precise representation of a program structure, it recovers program behaviour even less precisely. For example, it is not possible to recover many constituents of a UML sequence diagram solely based on a static analysis (e.g., the actual object recipient of a message and its type and order of messages sending). Programming language features (such as late binding, generalization, overriding, aliasing) make discovering such constituents from the source code syntax and semantics very difficult or even impossible; in general, as we mentioned earlier, it is an undecidable problem (e.g., [5-9]). On the other hand, depending on the programming language, many of the UML language features (such as `Lifeline`) can be captured accurately during dynamic analysis.

Dynamic analyses capture the required information by observing a program execution generally through instrumentation. Dynamic analyses can take place at different levels (e.g., operating system level, machine (binary) level, application level) to achieve different goals (e.g., reverse engineering, testing, verification). Similar to static analyses, in dynamic analyses the soundness ensures the desired properties are true for the whole program by observing and analyzing program executions (this is also known as test suite coverage in dynamic analysis), while the precision ensures that the observed program properties are the same as actual program properties. Dynamic analysis tools

usually favour precision over soundness. A dynamic analysis technique poses two main challenges when the instrumentation information tends to be large: 1) The instrumentation that captures runtime information (e.g., constituents of UML sequence diagram) during the dynamic analysis adds runtime overhead to the execution of the system under study (SUS), thus increasing the SUS's response time to the extent that a deadline may be missed, resulting in an observed information that may be different from the expected one. As a result, the instrumentation information may not be accurate. This is sometimes referred to as the "probe effect"; 2) A dynamic analysis technique relies on a set of test suites that ensures proper coverage of the program source code and functionalities. If the test suite partially covers the program, the generated behaviour does not represent the complete program behaviour, but the test suite's properties (i.e., not sound).

The problem of runtime overhead is particularly important in real-time and multi-threaded software since such software must generally operate under a specified response time. On the other hand, the soundness problem has been well studied in different areas of dynamic analysis. For example, there are many works to devise proper adequacy criteria to test software based on either source code (e.g., [10]) or sequence diagrams (e.g.,[11, 12]) or state machine diagrams (e.g., [13, 14]).

In this work, we focus on efficient and precise instrumentation in the context of reverse engineering source code to produce method calls, by using both static and dynamic analyses. We choose reverse engineering as a goal of static and dynamic analyses since such reverse engineering requires intensive instrumentation of program. Therefore, instrumentation optimizations for reverse engineering source code to produce

method calls should extend to other types of instrumentation in dynamic analysis (e.g., instrumentation for runtime verification or profiling for performance). One can envision a wide range of applications for reverse engineered method calls such as producing UML diagrams, program comprehension, profiling for performance, security interactions, and verifying software correctness. One main usage of reverse engineered method calls, which is specially noted in this manuscript, is to combine them with the control structures inside methods (i.e., inter procedural interactions) to produce UML sequence diagram. Many tools and techniques reverse engineer source code to produce object interactions to recover software design (e.g., [15]) or understand software features (e.g., [16]). Besides helping comprehension, reverse engineered information can help quality assurance [17-20]. For instance, runtime verification tools (e.g., Java based PathExplorer [19] and C++ based CBuster [20]) verify the correctness of program properties (which are formulated in temporal logic) and concurrency errors (e.g., deadlocks or livelocks) based on messaging events. In addition, reversed engineered interactions can be used in software security, such as uncovering unauthorized interactions between software components [21]. Furthermore, with sequence diagrams, i.e., object interactions, software practitioner can obtain other behavioural diagrams through transformation, such as state machine [22] or activity diagrams [23].

Compared to a purely static analysis or a purely dynamic analysis, a hybrid analysis (e.g., [24]) is a better analysis to solve the overhead problem. A hybrid analysis splits object interactions constituents into two parts, in that it collects one part statically and the other part dynamically. Applying both static and dynamic analyses to recover software behaviour causes lesser information to recover in the dynamic analysis, which

means less interference with the program execution and consequently a lower risk of inaccurate recovered behaviour. A hybrid analysis can recover object interactions soundly and precisely if 1) there are test suites to adequately cover the SUS, 2) a well established mechanism is devised to merge the static information to the dynamic information, 3) the probe effect is avoided, or at least limited to an acceptable level, in the instrumented SUS by a) capturing as much information as possible in the static analysis (i.e., fewer numbers of constituents remain to be captured in dynamic analysis) and b) implementing an efficient instrumentation mechanism for dynamic analysis to capture the remaining amount of information.

Among a very few works to use a hybrid analysis to recover object interactions, Kolbah, in her thesis [25], applied the hybrid analysis to recover object interactions for sequential Java programs. Following a trend[4] on hybrid analysis, she combined execution trace information, resulting from the dynamic analysis, with control flow information, resulting from the static analysis, to generate UML sequence diagrams. More accurately, she reverse engineered one scenario at a time from one execution trace, and then rendered it using the UML sequence diagram notation. Among the three abovementioned requirements for efficient and accurate reverse engineering of sequence diagrams, she addressed the second and third requirements for sequential Java programs: merging static and dynamic information and reducing instrumentation overhead by capturing some constituents of UML sequence diagram statically rather than dynamically. Nevertheless, she neither mentioned that her dynamic model to collect dynamic information is minimal, nor that her instrumentation tool is as efficient as possible. In addition, as mentioned in

[4] Half of the articles examined in a 2009 survey on program comprehension through dynamic analyses also employ static information [26].

the first requirement (i.e., adequate test suite), several scenario diagrams should be merged into a complete sequence diagram for a given use case. This requires triggering as many varied scenarios as possible through multiple executions of the instrumented SUS (e.g., using black-box testing techniques), and merging them into one sequence diagram. However, triggering varied scenarios and merging them were not addressed in Kolbah's work.

This dissertation focuses on recovering method calls from source code accurately and efficiently based on a hybrid analysis. As mentioned earlier, the overhead minimization in dynamic analysis is essential to reverse engineer accurate object interactions, especially in the case of real-time systems. In other words, the objective is to reduce instrumentation as much as possible, to disturb behaviour as little as possible (e.g., to limit the risk of inaccuracies in the reverse-engineered information), and compensate for the missing (dynamic) information by collecting static information. Disturbing behaviour as little as possible will ensure that, for instance object interactions resulting from thread communications in a multi-threaded software system will be correctly reverse-engineered. It is notable that even though we conduct experiments with different test cases based on different case studies, we do not reverse engineer complete sequence diagrams for all the use cases in a SUS; rather we focus on efficient instrumentation of the code to accurately reverse engineer the code and identify the execution of a scenario (based on its test case). Even though the soundness problem (i.e., the first requirement for hybrid approaches: test suite adequacy) is as important as the precision and accuracy problems, the soundness problem has already been investigated by others [11, 12], and it is more a software testing problem than an efficient instrumentation problem.

To achieve our objective, we optimize and extend Kolbah's hybrid approach for Java software. We optimize her hybrid approach to capture a sequence of method calls in Java software in that: a) we capture more pieces of runtime information statically rather than capturing just control flow information as she did, b) we optimize her instrumentation mechanism to reduce the runtime overhead.

1.1 Contributions

The contribution of this dissertation, which should be of interest to anyone who intends to develop a hybrid reverse-engineering technique, is four-fold:

First, looking at Kolbah's work [25], she successfully applied her hybrid approach to capture sequential features of small and large Java programs. Although she showed the effectiveness of her hybrid approach in minimizing the overhead for small software, we apply her hybrid approach on a larger case study (i.e., industry sized case study) to verify its effectiveness at reducing overhead for large software systems. Our experiments show that her hybrid approach reduces overhead at least by half for such systems when compared to a purely dynamic approach, but the instrumented software can still be two times slower than its non-instrumented version.

Second, even though Kolbah's approach improves upon a purely dynamic approach, there is room to further reduce overhead. To do so, we identify, systematically through experiments, how much each element of her instrumentation approach contributes to overhead. This empirical analysis will lead us to outline an optimization process and prioritize optimization activities in the dynamic analysis part of her approach: our objective will be to try to optimize further only when it makes sense to do so, i.e., for elements of her solution that contribute most to overhead.

Third, we suggest general characteristics of instrumentation techniques to study overhead impact. Then, we outline an optimization process to systematically study the contribution to overhead of each characteristic. This process includes a range of optimization activities such as what information to collect and whether it can be collected statically or dynamically. Depending on characteristics of an instrumentation and characteristics of a system under study, one needs to evaluate some trade-offs when selecting an optimization activity. This is a somewhat general result that can be applied to different instrumentation techniques to reduce overhead, in particular, those based on AspectJ instrumentation.

Fourth, we apply the optimization process on Kolbah's instrumentation and discuss plausible ways to optimize her instrumentation. We support this discussion with empirical results as needed. Next, we provide a quantitative analysis of the results of implementing some of these optimizations on her instrumentation for each characteristic, thereby reducing overhead. More specifically, we reduce the overhead in her instrumentation for information, collecting information, encoding information, logging information and study a combination of these characteristics.

1.2 Manuscript Organization

The remainder of this manuscript is organized as follows. We review the context, challenges, and the related work in the next chapter. In Chapter 3, we describe Kolbah's hybrid approach so as to optimize and extend it in later chapters. In Chapter 4, we investigate Kolbah's instrumentation in terms of the following: reducing overhead for industry-sized software systems, revealing the probe effect, and studying the contribution to overhead of each of its components. In Chapter 5, we suggest an optimization process

and apply this process for Kolbah's instrumentation and implement a series of optimizations on that instrumentation, thereby obtaining an optimized instrumentation. Finally, we conclude this dissertation in Chapter 6.

2 Background and related work

In the previous chapter, we mentioned the need to collect less information at runtime and the need for low cost instrumentation to efficiently and accurately reverse engineer object interactions for distributed real-time systems. In this chapter, we define the scope of the problem we seek to solve by describing the required background (Section 2.1) and related works (Section 2.2).

2.1 Background

In this section, we review reverse engineering techniques (Section 2.1.1), execution traces (Section 2.1.2), aspect oriented programming (Section 2.1.3), and the observability challenge when reverse engineering sequence diagrams (SD) from source code for distributed real-time systems (Section 2.1.4). Note that we review aspect oriented programming concepts as we use this methodology to implement our instrumentation strategy.

2.1.1 Reverse engineering

In the context of software engineering, reverse engineering is the process of analyzing a system to create representations of it in another form or at a higher level of abstraction [27-29]. If the created representation of a system is at the same level of abstraction as the original system, it can be used to create alternative views of the system or improve the current documentation. Creating a representation at a higher level of abstraction, the focus of our research, is usually used to recover design or specification from implementation.

The reverse engineering of source code to design can be done through static analysis and/or dynamic analysis. Static analysis refers to analyzing the program without

executing it, while dynamic analysis refers to analyzing the program through monitoring its execution. In static analysis, the source code is analyzed to reason about the program behaviour at runtime regardless of program inputs (e.g., compiler optimizations). More specifically, a static analysis technique entails a process to build a model of the state of a program to later reason about the program behaviour [2]. Static analysis can occur at different program levels (e.g., application level, OS level), for different purposes (e.g., verification, debugging, reverse engineering). However, static analysis techniques are usually sound but imprecise due to programming language features (e.g., late binding). Plenty of tools use static analysis to reverse engineer variations of UML models from source code (such as SA4J, MagicDraw, Rational Software Architect).

On the other hand, dynamic analysis uses various techniques to intercept program executions in order to monitor program behaviour. In dynamic analysis, program interception (i.e., instrumentation) can happen at different levels: at the source code level (e.g., Java and C++ source files), at an intermediate level (i.e., files generated after source code compilation such as Java byte-code files or C++ object files), at the binary level (i.e., binary executable from of source code), or at the operational environment level (e.g., virtual machine or operating system). Typically at the source code and intermediate levels, instrumentations uses either debugging techniques (e.g., [30-33]), profiling techniques (e.g., [34-36]) or injecting code at strategic points (e.g., [37, 38]). Program monitoring in dynamic analysis can take place either during program execution (online or ante-mortem analysis) or after the program execution through execution traces (offline or post-mortem analysis) [39]. We focus on offline analysis in this work.

Traditionally, dynamic analysis has been used for testing (to verify program correctness), profiling (to measure the program performance), debugging (to locate a program fault) and program understanding. Around the millennium, with growing complexity of software systems and rising importance of design models, researchers established new subfields for program comprehension through dynamic analysis: trace analysis, design recovery, feature (or concept) analysis, and visualization [40]. In fact, the growing complexity of software systems was the reason for the renewed interest in software testing through dynamic analysis as well [41, 42]. For instance, an extensive taxonomy has been suggested for runtime fault detection as a subfield of software testing, that classifies different research works and different tools in this area [41].

In the context of our research, we analyze source code using both static and dynamic analyses to efficiently generate method calls to be used for applications such as producing UML diagrams, either to comprehend a program, and/or to verify a program against its specification, and profiling a program for performance and security. In our dynamic analysis, we use aspect oriented programming (in particular, AspectJ and AspectC++ technologies) to instrument a program and then monitor that program offline through execution traces. The aspect oriented programming techniques in our approach instrument the program either at the source code level (i.e., C++ source file) or at the intermediate level (i.e., Java byte-code). For the static analysis, we use customized parsers (i.e., JavaCC, and Bison and Flex) and Soot[5] static analyzer to extract the control flow and call graph information from source code respectively.

[5] https://github.com/Sable/soot

2.1.2 Execution traces

During dynamic analysis, different characteristics of the SUS are monitored and traced. In the case of offline analysis of dynamically collected information, these characteristics (i.e., gathered information) are typically stored into a file as an execution trace. Depending on the goal(s) of the dynamic analysis, these characteristics can range from application level information (such as method/procedure/routine calls, variable values, branches, loops, exceptions, inter-process communications) to system level information (such as external interrupt, memory usage, system calls). Since this research involves recovering object interactions (behaviour) through method calls, we focus on execution traces for storing application level information. Therefore, an execution trace in our study turns into a sequence of events (e.g., method calls) and/or control flows (e.g., loops, branches) that partially or fully reflects the interactions among/within object(s) in the SUS.

In the area of trace analysis, the problem of trace size explosion turns into a big challenge in analyzing software behaviour since execution of even a relatively small program causes large traces [43]. Therefore, many tools and techniques have been developed to tackle this problem. For example, research papers suggest different abstraction techniques (such as [44-49]) or different metrics (such as [50]) for large traces to aid program comprehension or verification. In addition, they suggest many trace visualization techniques (e.g., [51-53]) that factor human cognition into concrete visualization to make a large trace understandable.

Since this work contributes to efficient trace generation, not trace abstraction or visualization, we focus only on those research papers that, fully or partially, address the

trace content problem: what information to collect and how to store the collected information on disk. Several trace formats, which signify different trace contents and ways of representing them, exist for recording object interactions (e.g., [54-56]) or other kinds of data (e.g., OTF focuses on recording performance data [57]). Later, we will look into related work on what to collect for a trace in Section 2.2.2 and what efficient storage mechanism to use in Section 2.2.3.3 when recovering object interactions.

2.1.3 Aspect Oriented Programming

Since the emergence of the object oriented paradigm, many mechanisms have been put in place to increase modularity and reusability of programs such as functional decomposition, use of design patterns (e.g., decorator, proxy, interceptor). Nevertheless, many concerns (e.g., runtime invariants checking, logging, tracing execution) still exist in software that spread over the entire program and it is not possible to identify those concerns (i.e., crosscutting concerns) in a single module or a single class. This discrepancy of concerns results in a decline in modularity and maintainability of a program.

Aspect Oriented Software Development (AOSD) has arisen in response to this problem. The Aspect Oriented Programming (AOP) methodology is a software development and maintenance paradigm that abstracts away crosscutting concerns of interest or adds new concerns in a separate module from the code base (which contains core concerns). The aspects in one program may differ from another depending on the nature of programs, customer requirements, and design constraints. The AOP methodology has been used in different areas of software engineering such as software maintainability, reliability, and security.

In this research, we use the AOP methodology as an instrumentation mechanism to add new concerns for tracing object interactions. More precisely, new concerns collect dynamic information and send the collected information to a Logger without changing the functionality of the SUS. Software practitioners use AOP as an instrumentation mechanism, to trace events (e.g., [58]) or to measure the performance (e.g., [59]) in software systems.

Depending on the base programming language, several AOP languages have been offered to support the AOP methodology. For instance AspectJ[6], AspectWerkz[7], JBoss-AOP[8], Guice[9], the AOP Alliance project[10] and Spring[11] are aspect languages for Java; AspectC++[12] and FeatureC++[13] are aspect languages for C++; and AspectC[14], AspeCt-oriented C[15], and Aspicere[16] are aspect languages for C. XWeaver[17] provides the AOP language support for C, C++, and Java. In this document, we use AspectJ as the instrumentation mechanism to monitor the behaviour of Java. We also use AspectC++ (Appendix D) to show the feasibility of the hybrid technique on other platforms.

Next we discuss some general AOP concepts that will be useful in the rest of the document (Section 2.1.3.1), AOP languages in general (Section 0) and contrasts specifically AspectJ and AspectC++ (Section 2.1.3.3).

[6] http://www.eclipse.org/aspectj/
[7] http://aspectwerkz.codehaus.org/index.html
[8] http://jbossaop.jboss.org/
[9] https://code.google.com/p/google-guice/
[10] http://aopalliance.sourceforge.net/
[11] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html
[12] http://www.aspectc.org/
[13] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/
[14] http://www.cs.ubc.ca/labs/spl/projects/aspectc.html
[15] https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/
[16] http://mcis.polymtl.ca/~bram/aspicere/
[17] http://www.pnp-software.com/XWeaver/

2.1.3.1 AOP concepts

Different AOP languages implement the concepts of the AOP methodology based on their specifications. The degrees to which languages support AOP concepts differ from one another. Although some suggested reference models [60, 61] (or meta-models [62]) for AOP concepts exist, there is no standard conceptual reference model for the AOP methodology. In this section, we explain the AOP concepts based on the generic model suggested by Laddad [60]. Laddad suggested that each AOP language should at least support the concepts in his generic model (Fig. 1) to be considered as an AOP language. In the following, we explain these concepts and their relations in details. Later in Chapters 3 and 5 we will refer to these concepts to describe our instrumentation technique implemented based on AspectJ.

Join point: often in the execution path of an object oriented system, it is possible to identify points where method execution, object creation/deletion, or exceptions happen. These identifiable points, which are known as join points in AOP, are exposed by the aspect language as sites for possible code augmentation. Depending on a join point model of an AOP language, a join point can expose different points (i.e., locations) such as method execution, object creation/deletion, or exception handling. Therefore, an aspect language can define different types of join points based on the exact location in the exposed event. For instance, many AOP languages support "`call`" and "`execution`" join points for "method execution" events. An "`execution`" join point exposes method body, whereas a "`call`" join point exposes the method invocation location.

Pointcut: a set of join points is selected through a pointcut construct. More specifically, during system execution a pointcut selects those method executions, object

creations, and/or exceptions which satisfy a pointcut's predicate (similarly to selecting rows in SQL queries).

Crosscutting element: a crosscutting element is a construct used to build crosscutting functionality. The construct can alter either the static structure or dynamic behaviour of a system. A pointcut, as a separate construct, is used to navigate the alteration of static structure or dynamic behaviour.

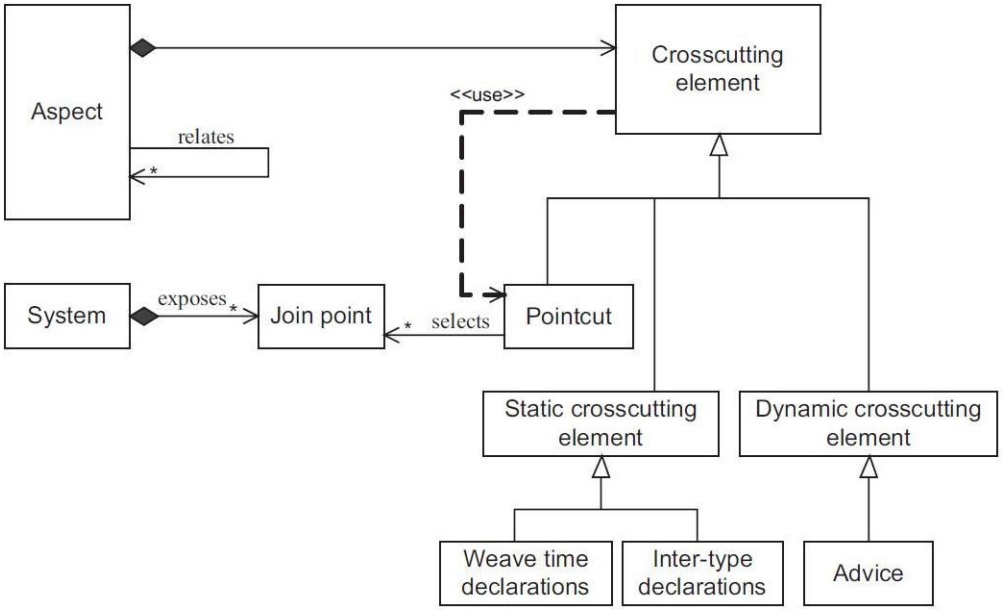


Fig. 1 Generic AOP model [60]

Dynamic crosscutting element (Advice): after a pointcut is used to select a proper set of join points, there should be a mechanism to augment the code at the site of the selected join points in the system. An advice construct provides such a facility in a way that it is being executed `before`, `after`, or `around` the selected join points to alter the system behaviour. A `before` advice executes before each selected join point, an `after` advice executes after each selected join point, and an `around` advice executes over

each selected join point. An around advice which surrounds join points must be declared with a return type to return a value.

Static crosscutting element: aspect languages provide mechanisms to alter the static structure of a system through inter-type declaration and/or weave time declaration. In inter-type declaration, a new static structure (such as a new method or field) is introduced to alter a class, an interface, or an aspect in the system. However, when a new static structure is introduced in the advice, the structural change happens on specific executions. Such structural modifications are called weave time declaration since the declaration happens at weaving time.

Aspect: an aspect is a module that contains the semantics of the new crosscutting functionality. The semantics is expressed through crosscutting constructs (i.e., pointcut, advice, inter-type declaration, weave time declaration). Aspects can relate to one another, similarly to relations between classes in object oriented programming.

To summarize, we briefly review main AOP terms we use throughout this manuscript:

- Join point: an identifiable point in the base source code (e.g., Java code) that is augmented with meta-language source code (i.e., aspect code).
- Pointcut: a mathematical function (based on the aspect language notation) that defines a proper set of join points.
- Advice: a piece of code in the meta-language notation (i.e., aspect language) that augments a join point location.
- Inter-type declaration: a mechanism in the aspect language to alter the static structure of the base language (e.g., Java), such as adding a new method or field.

- Aspect: a module (similar to the concept of class) that contains the semantics of the aspect code.

2.1.3.2 AOP languages

The AOP methodology should be implemented with a language. An AOP language is composed of two parts: language specification and language compiler. The language specification (i.e., the programming syntax) explains the language constructs to implement the core and crosscutting concerns, while the language compiler first verifies the correctness of written code against specification and then translates the code into an executable form [60].

The language specification should provide constructs on how to implement the following: 1) each individual concern (e.g., advices and inter-type declaration in Java); 2) the weaving rules among concerns. The concern is generally implemented in the language of the target programming language, i.e., standard C, C++, or Java. The core concern in our instrumentation approach is the SUS, which remains intact during the instrumentation process. The new concerns (i.e., probes), which are implemented in Java, collect dynamic information in our approach. Then, the new concerns are merged to the SUS based on weaving rules. The AOP language specification can express the weaving rule constructs either through extending the base language (such as Java) or devising new constructs.

The AOP language compiler (i.e., weaver) verifies the code against the AOP language specification, combines concerns according to weaving rules, and generates the executable code. Some AOP compilers (e.g., AC++ and early versions of AspectJ) use a source-to-source transformation by processing aspects and classes (i.e., crosscutting

concerns and core concerns) to produce the woven source code. Then, the weaver supplies the woven source code to the base language compiler to generate executable code. Another type of compilers (e.g., AspectJ), first supplies the base language compiler with core concerns. Then, the aspect code is merged with compiled files to generate the woven compiled files.

For example, in the case of AspectJ, the AspectJ compiler uses the Java compiler to compile Java source code (i.e., core concerns) and then weaves the aspect to Java class files to produce JVM compliant byte-code [60]. Alternatively, the AspectJ compiler can be supplied with Java byte-code of the core concerns to directly weave the aspect to Java class files. We use the former approach since there is no difference in terms of overhead cost between these two approaches (both happen at compile-time). It is notable that the AspectJ compiler performs advice weaving in two phases: lookup and invocation. Lookup selects a set of advices that applies to each join point whereas invocation runs the selected set of advices on that join point. The compiler performs these in two modes: compile-time weaving (a.k.a., static weaving) executes lookup at compile time and invocation at runtime whereas load-time weaving performs both at runtime [63]. In our experimentations, we use compile-time weaving.

2.1.3.3 AspectJ and AspectC++: similarities and differences

The AspectJ and AspectC++ (AC++) languages provide the instrumentation facility for Java and C++, respectively. AspectJ and AC++ extend Java and C++, respectively, to implement the AOP methodology. As mentioned earlier, they use different weaving mechanisms for their compilers. While AspectJ weaves aspects into compiled Java byte-code to generate new class files [60], AC++ weaves aspects to C++

source code through source-to-source transformation to generate C++ source code [64-66]. AC++ uses the C++ template instantiation mechanism to implement this transformation.

Aside from differences in weaving mechanism, AspectJ and AC++ provide different support for AOP concepts. This is, in part, due to differences between Java and C++ (e.g., explicit C++ pointer handling versus implicit Java pointer handling). In addition, AspectJ and AC++ follow different join point models. For instance, AspectJ cannot support `call` and `execution` join points when an object is destroyed. On the other hand, AC++ cannot support a `call` join point when an object is created or destroyed. In addition, AC++ bears more limitations to inter-type declaration compared to AspectJ. For example, AC++ does not allow defining an inter-typed static method. We will refer to these differences later (Chapters 3, 5 and Appendix D) when we explain the implementation of our instrumentations.

2.1.4 Challenges to reverse-engineer Distributed Real Time Systems

Before explaining the challenges one faces when reverse engineering source code to produce object interactions from a Distributed Real-Time Software System (DRTS), it is important to define what we mean by DRTS. A real-time software system (RTS) is a system whose "correctness depends not only on the logical result, but also on the time at which the results are produced" [67]. A hard real-time software system is "a real-time system in which it is absolutely imperative that responses occur within the specified deadline, because otherwise severe consequences will happen" [68]. Otherwise it is a soft real-time software system. A distributed software system (DS) is "a collection of independent computers that appears to its users as a single coherent system" [69]. A

DRTS is a DS where at least one node (i.e., independent computer) is a RTS. In such a software system, each node is interacting with other nodes based on its running process. In the context of operating systems, a process is a program in execution which uses its own resources (e.g., memory or CPU). A thread is a smaller unit in a process that executes its own code independent of other threads, but it shares memory and process context with other threads in that process [69]. Usually RTSs are implemented based on multi-threading concept.

A reverse engineered scenario (i.e., a series of object interactions based on one execution) from a RTS should capture information about timed events such as thread communications or synchronous messages between objects to generate the correct interaction order in the diagram. A reverse engineered scenario from a DS should also address timing problems by identifying the correct order of object interactions among different computers.

There are several architecture styles for a DS, such as layered-based, event-based, etc. In the context of this research, we only focus on those DSs that follow object-based architecture and operate on the Java Remote Method Invocation (RMI) middleware. Our solution in this manuscript can be seamlessly extended for Java RMI based DSs. In DSs, a middleware provides a logical layer between applications (or processes) and the network communication as if applications are running on a single coherent system [69]. Therefore, a RMI based middleware provides direct object interactions for applications by abstracting the network communication details. In this architecture, each node can be a RTS or a multi-threaded application (as a specific type of OO application) that runs a process.

Monitoring the behaviour of a DRTS (either for testing or comprehension purposes) is a complex and expensive activity compared to sequential software systems due to challenges such as observability or reproducibility [70, 71]. For a given test case in a DRTS, the observability challenge represents the problem of monitoring what the software does, and how and when it does this. In other words, an observable DRTS ensures that there is no discrepancy between the observed behaviour (e.g., program variables) and the actual behaviour. The reproducibility problem relates to the difficulty of producing a deterministic expected output based on a given input for the DRTSs. A single input in a DRTS can produce a range of outputs due to the timing and the order of execution in a process (in a RTS) and among processes (in a DS) [72]. In this work, we particularly focus on the observability problem by looking into different monitoring techniques to make sure the chosen monitoring technique has not changed the observed program variables due to the interference by monitoring the execution of a DRTS. Even though the reproducibility problem is more related to the area of testing, we need a deterministic output to judge the correctness of the observed behaviour. We do not contribute to solving the reproducibility problem in DRTSs in this research; since existing solutions (e.g., [72]) provide a deterministic output.

As mentioned before, adding probes to a program or halting a program at predefined breakpoints are two prevalent techniques to capture values of program variables. However, such techniques can potentially introduce execution delays such that some deadline may be missed and an error could happen (e.g., making processes go out of synchronization). This is what we, as well as others (e.g., [71]), refer to as the probe effect. The probe effect not only can change the behaviour in a DRTS (e.g., preventing or

causing errors by changing relative timing among processes), it can also lead to altered temporal properties (i.e., a delay changes the received time of some event).

In general, there are three types of solutions to overcome the probe effect problem in a DRTS: ignoring, minimizing, and avoiding [70]. Depending on the type of DRTS and extent of the probe effect, the effect might happen rarely, and, consequently, may be ignored. Thus, no particular action with regard to the probe effect is required in monitoring such cases. The second class of solutions, probe effect minimization, could be performed through implementing efficient monitoring operations or compensating the probable effect of probing. Techniques such as the use of global breakpoints (e.g., a debugger halts a system in a consistent way) [73], offline instrumentation [74, 75] or instrumentation optimization [76, 77] are suggested for probe effect minimization. The third class of solutions, probe effect avoidance, is necessary to monitor hard real-time systems since ignoring or minimizing the probe effect in such systems may result in catastrophic failure. There are different techniques to avoid interference in a DRTS. One common technique is to use dedicated hardware to passively listen to the traffic between processor, local memory and other parts of the nodes in a DRTS. For example, Tsai et al. use this technique to monitor and replay a RTS behaviour by interfacing the target RTS to the dedicated monitoring hardware [78].

In our research, we apply probe effect minimizations techniques. We use a hybrid analysis technique to monitor the behaviour of sequential programs, thereby reducing the overhead by capturing part of the information during static analysis and the rest of the information in dynamic analysis through efficient instrumentation. Although we try to minimize the probe effect primarily on sequential programs, we also evaluate our

approach on a multi-thread software. Even though we do not collect real-time or distributed information in our hybrid technique, minimizing probe effects for multi-threaded software systems provides enough ground to collect such information efficiently and to easily extend our optimized technique to support RTSs and RMI based DSs (Section 6.3).

2.2 Related works

In Section 2.2.1, we review those works that reverse engineer the behaviour of object oriented systems from source code with respect to the type of reverse engineering technique (i.e., static, dynamic, hybrid) and the kind of target system (i.e., distributed, real-time, sequential). In Section 2.2.2, we look at different trace analysis works to understand what information they collect in their suggested trace formats. In Section 2.2.3, we review the work related to performance optimization of instrumentations.

2.2.1 Related work on hybrid/static/dynamic analysis

In this section, we discuss different types of techniques to reverse-engineer object interactions from source code, specifically, static, dynamic and hybrid ones, in the context of different kinds of target software systems, specifically, sequential, distributed and real-time ones.

Note that we consider reverse engineering of interaction diagrams for web applications, either through dynamic analysis (e.g., [79, 80]) or static analysis (e.g., [81]), less related to our research. Although web applications are a type of distributed systems, reverse engineering a web application does not require collecting the same information as for the kind of DRTS we target. For example, reverse engineering a web application is about interactions between browser sessions, generated pages, and page transitions

instead of object interactions. Therefore, we do not include reverse engineering web applications in this section.

Recall that this work focuses on efficiently recovering object interactions (and method calls in the more specific context) from source code. The correctly reverse engineered scenario, then, can for instance be used to understand a software system or to verify it against its specification during system testing. Hence, we also examine related works aimed for software comprehension and software verification. In addition, we emphasize on precision (or correctness) of reverse engineered scenarios in this work. This distinction is necessary since many works in the area of software comprehension recover software behaviour not as precisely as we would like since partially correct or incomplete information helps maintainer to understand a system due to human resilience to this kind of information [82]. Applying approximation techniques (such as abstraction or filtering) to provide a higher-level view of a system is out of the scope of our research, but it is a logical continuation of this work.

In this section, we discuss the related work to recover object interactions based on static analysis (Section 2.2.1.1), dynamic analysis (Section 2.2.1.2), and hybrid analysis (Section 2.2.1.3). Note that we give special attention to Kolbah's work [25] in this section (specifically in Section 2.2.1.3) since our research is to optimize and extend her hybrid analysis approach.

2.2.1.1 Static analysis

Static analysis of object interactions is generally based on flow analysis (either control flow or data flow) to extract dependencies among objects and method calls. The result of dependency analysis is, then, presented as a variant of an interaction diagram

(e.g., sequence diagram, communication diagram, call graph, control flow graph). For example, a Control Flow Graph (CFG), a representation of the program containing object interactions, is a directed graph where each node shows a basic block and each edge shows a control flow path [83].

Control flow analysis techniques (either intra-procedural or inter-procedural) are generally divided into two groups according to the purpose of the analysis: code-based and model-based. Code-based control flow analysis is mainly used during testing and compiler optimization. For instance, capturing a variant of a CFG from source code is a routine static analysis task for compliers during code optimization [84]. On the other hand, model-based control flow analysis is the analysis of control flow information which is embedded in a design model (such as analyzing loops, conditionals or exceptions in a UML sequence diagram) [85]. In the literature of reverse engineering of sequence diagrams from source code, some static analysis works used model-based control flow analysis to generate object interactions. For instance, Rountev et al. [86] suggested an algorithm to convert a CFG to control flow primitives of a UML sequence diagram (e.g., *loop*, *alt*, *opt*) by analyzing a CFG for each method of a class (i.e., intra-procedural control flow analysis) for Java programs. They turned to data flow analysis techniques to show inter-procedural control flow in sequence diagrams.

In the context of recovering a sequence diagram from source code, data flow analysis techniques are used to identify objects and invocation relationships among methods. Multiple problems require some sort of data flow, static analysis when reverse engineering sequence diagrams from source code, such as identifying lifelines, inheritance, polymorphism, and order and feasibility of different sequences of method

calls. All these problems can be classified as particular instances of one problem in the static analysis area: how to analyze source code to obtain information about data creation (i.e., object creation and deletion in sequence diagrams) and reachability and precision of a generated path (i.e., feasible interactions among objects in sequence diagrams). There is a body of knowledge in the field of static analysis to solve this problem that includes areas such as pointer analysis (including object naming analysis, propagation analysis and reachability analysis, e.g., [87-95]) and call graph construction analysis (including polymorphism and inheritance analysis, e.g., [96-98]). Note that a program call graph, as a form of object interactions, is a directed graph to represent all invocation relationships between a program's methods, where each node shows a method and each edge shows the calling relationships between two methods [96]. There are different algorithms with different levels of approximation to build a call graph. For example, the Class Hierarchy Analysis (CHA) algorithm [99] builds a class hierarchy based on a call site and identifies all possible callees for that call site in the call graph based on reachability analysis. The Rapid Type Analysis (RTA) algorithm [100] optimizes the CHA: instead of including all possible callees based on class hierarchies, it includes only those callees from class hierarchies that their enclosing class have been instantiated somewhere in the program. The Variable Type Analysis (VTA) algorithm optimizes [101] the RTA: after including all possible callees based on instantiated objects in the program, the VTA narrows down possible callees by calculating object reference for each variable in the call site. All these algorithms are sound and the precision increases for each optimization.

With regard to the abovementioned static analysis concepts, there are a couple of works to recover object interactions from source code [102, 103]. These works both

recover object interactions from a call graph based on different point-to analysis techniques to identify objects (i.e., object naming analysis) and different inter-procedural propagation techniques to analyze method calls. Note that point-to analysis techniques approximate an object creation based on object expression (or allocation) sites in the source code. In addition, propagation techniques in inter-procedural data flow analysis generally provide contextual information for a user by propagating information about data creation and data usage among procedures in a program [104]. In the first work, Tonella and Potrich [102] reverse engineer object interactions for C++ programs without recovering the control flow within objects in the form of collaboration diagrams. Similarly, Rountev and Connell [103] present an algorithm, along with their previous works [86, 88, 92], to recover object interactions from a call graph for Java programs in the form of a sequence diagram. Compared to the first work, their algorithm provides a better static approximation of object creation and a better propagation technique for method calls at run time, which results in more precise sequence diagrams. The main difference between Rountev's work and the first work (i.e., [102]) is in object naming analysis: instead of having a new lifeline for each new object expression (i.e., *may-alias* information [102]), their algorithm [103] reduces the number of lifelines by defining the notion of the "singleton call site". Then, they use the identified singleton call sites to develop a more precise propagation technique. A singleton call site guarantees that there is at most one receiver object at that call site at runtime. It is notable that both algorithms are incapable to recover some constituents of the UML sequence diagram precisely. For example, even though static analysis techniques can identify the actual object (i.e., lifeline) in non-singleton call sites, identifying the actual initialized object at such call

sites is generally undecidable. As we mentioned before (Chapter 1), static analysis algorithms capture information from a SUS imprecisely (due to not executing the SUS) [8, 9].

Neither of the abovementioned algorithms targets distributed or real-time software systems to recover the behaviour. Even though there exist static analysis works to analyze either real-time (e.g., [105, 106]), or distributed (e.g., [107-110]) software systems, to the best of our knowledge, there is no work to use such concepts in reverse engineering of source code to produce object interactions.

There is another line of work to perform control flow analysis [111-117], though this analysis is between an object oriented program and a meta-program used within that object oriented program (e.g., AOP languages or Enterprise Java Beans-based[18] (EJB-based) applications). Such works generally propose an extended form of inter-procedural control flow graph to show the meta-language constructs (e.g., advices in AOP or interceptors in EJB). For example, some works [111-114] focus on relationships between an OO program and an aspect oriented program. They recover extended forms of control flow graph (e.g., extended call graph) to show the interaction between objects and aspects. They propose a new representation for the aspect language constructs (advice, pointcut designators) and different types of calls (e.g., method call, advice call, or advice execution) to show interactions. Similarly, some other works [115-117] recover the interaction in enterprise applications between an OO program and its EJB dependencies in the form of a sequence diagram. In Enterprise Java Bean applications, which are used in server-side applications, EJB provides a dependency injection mechanism through the interceptor design pattern which allows an object (or bean) to declare dependency on

[18] http://www.oracle.com/technetwork/java/javaee/ejb/index.html

other objects (or beans) [118]. The mentioned works in this area suggest algorithms for EJB applications to recover the sequence diagram from source code based on a type of EJB method (e.g., business method interceptor, callback method, or timeout method) while abstracting the control flow within methods. Since we only focus on interactions among objects in our research, other OO applications which use meta-programs are out of the scope of our work.

2.2.1.2 Dynamic analysis

In 2009, Cornelissen et al. conducted a systematic survey [40] in the area of program comprehension. The authors systematically analyzed 176 papers (out of the 4,795 initially selected), published between July 1999 and June 2008, that rely on dynamic analysis to conduct program comprehension activities. Twenty of those papers[19] use dynamic analysis techniques (e.g., debugger, source code instrumentation) to reverse engineer object collaborations, rendered under the form of a UML sequence diagram (or a similar diagram).

We focus on those twenty papers as they directly relate to the hybrid approach used in our research. These works collect execution information, specifically constructor, static/non static method calls (or executions), to produce scenario diagrams (i.e., a diagram representing one single execution, or trace). While some of these approaches use both static and dynamic analyses, none of them actually combines both types of analysis to produce dynamic models: the static analysis is only used to generate structural diagrams (e.g., class diagram) and the dynamic analysis is only used to generate object collaborations. In some rare cases, the static analysis is used to guide the user in selecting

[19] References number 19, 21, 22, 23, 24, 27, 29, 30, 33, 40, 90, 103, 116, 121, 123, 126, 129, 135, 141, 147 in [40]

elements of the source code to monitor during the dynamic analysis (e.g., [119]). The majority of those works do not reverse engineer information on alternative executions (i.e., control flow) and generated diagrams do not therefore indicate under which conditions or repetitions objects send messages.

Only two works [56, 120] are closely related to Kolbah's hybrid approach in terms of the generated diagrams, although they are both only dynamic. Leduc and colleagues' work, the closest to Kolbah's work, instruments the source code using aspects and collects method executions and control flow information [56], while the other relies on break points (for method and control statements) being set with a debugger [120]. Other works indicate repetitions in generated diagrams; however, they either use a simplistic heuristic to identify repetitions [121] (specifically, contiguous repeated messages are collapsed into repetitions, which does not produce an accurate diagram in general) or recognize occurrences of known interaction patterns that must be provided by the user [122, 123].

It is notable that the majority of selected papers do not capture real-time or distributed features of a SUS. Only two of those papers capture both real-time and distributed features of the SUS in the form of source and destination timestamp, and node ID [56, 124]. There are also two other works which capture either distributed [125] or real-time [126] features of a SUS. Oechsle and Schmitt built a prototype tool with limited support for multi-threaded Java programs without mentioning how exactly they captured dynamic information [126]. Souder et al. used a Java Profiler to capture dynamic information without identifying threads [125].

We now briefly review Leduc and colleagues' work [56, 127] to better understand the differences with Kolbah's hybrid approach in the next section. Leduc uses a 4-tuple in her approach to uniquely identify an object instance: node ID (unique identifier of a node in a network), thread ID (unique identifier on a node), object ID (unique identifier in a running software), and class name. The 4-tuple is used in four aspects during dynamic analysis to capture information regarding: method calls, threads, control structures, nodes. The first aspect, method call aspect, intercepts method execution (through three `around` advices) to capture caller's and callee's information. The second aspect, thread aspect, captures a thread ID and time stamps by intercepting all calls to a thread (through a `before` advice) and a thread execution itself (through an `around` advice). The third aspect intercepts calls to methods (through `call` advices) to capture control structures (e.g., for, if, while). Since AspectJ did not offer any mechanism (i.e., join point) to capture control structure, Leduc marked the beginning and the end of each control structure in the source code by adding calls to custom methods with an empty body to be able to capture (aspects targeting calls to these custom methods) conditions and loops with AspectJ, similar to how method calls are captured. These calls to the empty methods, which provide interception points to capture control structures, are added to the SUS through a customized parser to modify the abstract syntax tree (AST) before execution. The fourth aspect, node aspect, intercepts execution of each node on RMI middleware to capture node ID in a distributed SUS. Thanks to the use of RMI and the recording of client and server local time stamps for the sending of RMI calls and receiving of RMI calls, no additional time stamps for nodes are collected since recorded thread time stamps and the order of logging on log server would be enough for

synchronization. Recall that RMI calls are synchronous, that is a client thread has to wait until the remote call execution finishes at server side. Therefore, if a client thread makes multiple remote calls, it is possible to reconcile caller's information (client) with callee's information by looking at the order remote calls happen (based on local clock) in both client and server traces. Once the execution has started, by feeding test cases to the SUS, aspects capture behaviour in the distributed real-time SUS.

Since the 2009 systematic survey, additional related work has been published. Once again, we focus on reverse engineering object collaborations through dynamic analysis. Some approaches attempt to generate sequence diagrams by relying on execution traces through a dynamic only analysis and without (necessarily) recovering alternatives or loops [128-133]. None of those works captures distributed or real-time features of a SUS, except one which visualizes multi-thread object interactions [128]. In that work, Ishio et al. [128] develop a Java profiler, AMIDA[20], to capture the method signature, object ID, thread ID and timestamp on program events (i.e., method call, method return and exceptional exit) and visualize those events in a sequence diagram. In essence they collect as much information as Leduc, and are therefore not interested in reducing the amount of information they collect at runtime. This is however one of our objectives. Another different work [134] uses the sequence diagram notation to show interactions among distributed components (i.e., nodes) of a Java RMI application, but authors do not recover object interactions on each node.

Similarly to Malloy and Power [119], static analysis is sometimes suggested, but not actually used, as a way to guide instrumentation [131, 132]. Ka-Yee Ng et al. [131] use static analysis to identify where to instrument in the program by locating all potential

[20] http://sel.ist.osaka-u.ac.jp/people/ishio/amida/

design motifs in the source code. Taïani et al. [132] narrow down the scope of instrumentation in their work by using a static analysis algorithm to convert a program stack-trace to a call-tree. Nevertheless, they differ from one another in terms of target system since for instance, the latter paper (i.e., [132]) supports multi-threaded environment on distributed middleware (CORBA). One purely dynamic technique [15] recognizes loops and alternatives when reverse engineering network communications. This technique complements Kolbah's hybrid approach [25]: it looks at the boundaries of interacting networked applications while the hybrid approach of Kolbah looks at the inside of the interacting applications. They instrument each interacting node in the network to monitor packets and capture timestamp of each message (either sent to or received from), node ID and packet ID. Since they render node interaction in the form of a sequence diagram, they do not rely on a distribution middleware: for instance, Leduc and colleagues rely on RMI [56]. Another purely dynamic technique [130] identifies loops and alternatives. Ziadi et al. focus on how multiple execution traces can be merged to generate a sequence diagram whereas we focus on how such traces can be obtained without disturbing too much the observed behaviour. Yet another work [135] records temporal constraints of object interactions on trace for the distributed SUS to be later verified against a specification. However, they do not use any distribution middleware to monitor object interactions; instead they use an architecture in which each distributed process (on each node) is connected to a specific process, the so called central data collection process (a relational database), through a socket connection. The central data collection processes logs information of different processes and synchronizes them by determining a clock offset for each process.

Many tools are capable of reverse engineering software to generate sequence diagrams. (We omit tools that only reverse-engineer software to generate class diagrams, such as Topcased[21], Poseidon[22], ModelMaker[23], Together[24], or MoDisco[25].) They either rely on a purely static analysis of the source code (e.g., MagicDraw[26], RSA[27]), or trace method executions/calls without collecting control flow information (e.g., MaintainJ[28], reverseJava[29], JSonde[30], javaCallTracer[31], J2U, TPTP's UML2 trace interaction View[32], CodeLogic[33], PragmaDev Tracer[34]). Note that Fujaba[35] and related projects do not reverse-engineer sequence diagrams. However, some Fujaba projects do manipulate traces, but for the purpose of detecting design patterns. With respect to tool support for reverse-engineering sequence diagrams, some authors discuss the right features such a tool should provide, especially when dealing with large traces/diagrams [136].

Another recent dynamic analysis work [137] gathers similar dynamic information as Kolbah does with a similar instrumentation technique. They collect information about the number of method invocations, the number of object creations, the name and signature of caller and callee methods. However, they use this information through visualization techniques to guide the software maintainer to better understand and analyze the system instead of reverse engineering sequence diagram. They use AspectJ to

[21] http://www.topcased.org/
[22] http://www.gentleware.com/
[23] http://www.modelmakertools.com/
[24] https://www.borland.com/products/together/
[25] http://www.eclipse.org/MoDisco/
[26] http://www.nomagic.com/products/magicdraw.html
[27] http://www.ibm.com/developerworks/downloads/r/architect/
[28] http://maintainj.com/
[29] http://www.reversejava.com/reversejavahome.htm
[30] http://jsonde.sourceforge.net/
[31] http://javacalltracer.sourceforge.net/
[32] http://www.eclipse.org/tptp/
[33] http://www.logicexplorers.com/CodeLogicOverview.html
[34] http://www.pragmadev.com/product/tracing.html
[35] http://www.fujaba.de/

intercept method calls and object creations. Specifically, they use `before execution` and `after execution` advices to capture the number of method calls and `after call` advices to capture the number of object creations. Because one can capture only callee's information through an `execution` join point, they rely on the call stack (i.e., call tree) to capture the caller's information of each call. Kolbah, on the other hand, used `before call` advices to capture both caller's and callee's information when a method call is made.

2.2.1.3 Hybrid analysis

In this section, we briefly review Kolbah's work first, since we build on it, and then discuss other related works. Her hybrid approach is summarized as a UML activity diagram in Fig. 2 [25]. The approach minimizes instrumentation by combining a static analysis technique with a dynamic analysis technique. Kolbah uses aspects (activity `a1` in Fig. 2), and executes the instrumented version of the software using test cases (activity `a2`). Offline, the hybrid technique reverse-engineers the control flow graph of methods (activity `a3`). It then combines the trace and control flow information in a model transformation activity (`a4`) to generate scenario diagrams. Kolbah created tool support to automate activities `a1`, `a3` and `a4`. Activity `a2` can be automated, for instance using a framework like JUnit.

Leduc's work [56], the closest dynamic approach to Kolbah's hybrid approach as we mentioned earlier (Kolbah's solution builds on Leduc's solution), has a similar process, the main difference being the absence of activity `a3` (and the generated graphs). As a consequence, activity `a4` is different. The two approaches are equally easy to set up and use, as the same activities are automated and the aspects are generic (i.e., can be automatically tailored to any case study software). In addition, since Kolbah's hybrid

approach does not capture any information about distributed and real-time behavioural characteristics of the SUS (e.g., node and thread identifiers), contrary to the two aspects capturing distributed and real-time behaviour of the SUS in Leduc's approach, there is no code in the instrumentation to capture such behaviour. Therefore, Leduc's solution for distribution and multi-threading can theoretically be adapted to Kolbah's solution. Last, although Leduc's instrumentation strategy collects the same amount of information as Kolbah's instrumentation strategy, the former involves two calls to the trace logger for every method execution and two additional calls for each control flow structure (i.e., condition or loop) whereas the latter involves only calls to the trace logger for method calls; Kolbah's instrumentation is therefore lighter in terms of impact on the SUS execution than Leduc's.

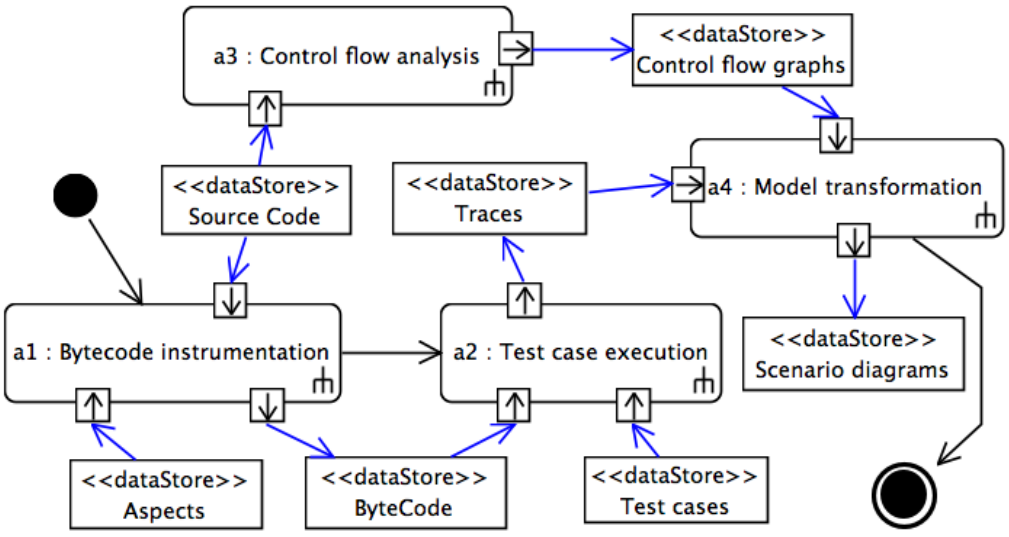


Fig. 2 Hybrid approach (UML activity diagram)

The main contributions of Kolbah's work are: (1) a hybrid approach combining static and dynamic data for reverse-engineering behaviour from source code with the intent to reduce execution overhead; (2) a precise modeling of the approach (with models

and OCL mapping rules), allowing model transformation; (3) the reverse engineering of software for alternative and iterative executions, including test conditions; (4) case studies, though limited in size, enough for verifying the accuracy and correctness of generated scenario diagrams against the ones, previously deemed correct in Leduc's work [56], accounting for the different versions of UML (1.x in Leduc's, 2.x in Kolbah's); (5) investigating the performance improvement of her hybrid technique against Leduc's dynamic technique based on limited size case studies.

Tonella et al. [138] used both dynamic and static analyses to recover an object diagram, another form of diagram that can show interactions, from source code, though limited to showing possible links between objects without showing how these objects actually interact, from C++ source code. They created a conservative form of the object diagram based on a CFG constructed through a static analysis technique and performed a number of executions on the SUS (based on a test suite) to create a less conservative object diagram (e.g., understand the correct multiplicity of relationships). However, unlike Kolbah's approach, they didn't attempt to collect less information during dynamic analysis. In other words, Kolbah's hybrid approach generates accurate scenarios for the sequence diagram, whereas the technique presented by Tonella et al. provides more complete scenarios (depending on the coverage of the test suite) for the object diagram.

One recent hybrid technique [139] is to collect the same information as Kolbah's hybrid approach [25], specifically the line number and signature of invocations (static information) and invocations objects make to one another (dynamic information). It uses an algorithm to merge static information (i.e., AST) with dynamic information (i.e., call graph), while Kolbah's hybrid approach merges static and dynamic information (as

instances of models) through OCL transformation rules. In addition, this recent hybrid technique relies on debug and source code analysis to collect static information, while Kolbah only relies on source code analysis. Moreover, it has a different objective than Kolbah's hybrid approach: showing how static and dynamic information allow a tool to recognize loops and report them in a condensed way in order to better visualize object interactions for locating program features (the authors do not report on the execution overhead caused by their instrumentation strategy); instead, Kolbah studied how combining both techniques reduces instrumentation overhead, while also recognizing control flows.

They implemented their technique as an Eclipse plug-in, named Diver[36], to instrument programs and visualize sequence diagrams to identify the SUS features. Experimenting with their technique and Diver on different case studies showed other limitations of this technique/tool such as its difficulty to uniquely identify all object instances or identifying nested constructor calls. Note that we cannot comment on the instrumentation performance of these two approaches without conducting an experiment to compare them. However, such comparison would not be accurate in this case since Kolbah's instrumentation tool operates as a standalone application whereas the debugger-based instrumentation operates as an Eclipse plug-in. It is likely that Eclipse affects the debugger performance since many of Eclipse libraries are called during instrumentation. Therefore, we can only experimentally compare Kolbah's work to Leduc's work to judge the effectiveness of her hybrid approach. It is worth noting that all of the mentioned hybrid techniques can only capture sequential features of the SUS. To the best of our

[36] http://eclipsediver.wordpress.com/

knowledge, there is no hybrid technique or tool that can capture real-time or distributed features of the SUS.

2.2.2 Related works on collecting information in traces

As mentioned in Section 2.1.2, many trace formats exist for collecting object interactions information, each of which allows one to represent different amounts of information. In fact, many tools, that have been developed to recover object interaction, specify their own trace format (e.g., Shimaba, Scene, Program Explorer, Jinsight formats) [43]. There is no single standard trace format regarding the content of trace when recording object interactions. Nevertheless, there are popular trace formats among software practitioners to capture system information (e.g., Common Trace Format (CTF) [140] and Perf data[37]). These formats are mainly used to capture event's information (e.g., PID or stack) and system's information (e.g., CPUID or Hostname). For example, CTF is an open and efficient format that uses metadata (based on Trace Stream Description Language) to produce a binary trace.

There are dynamic analysis works that suggest what elements to collect (such as [54]) as well as the relationship among the collected elements in the form of a meta-model (such as [55, 141, 142]). Similarly, some static analysis works suggest what information to collect in the form of a meta-model (such as [85, 143]). Among the abovementioned works, several (i.e., [141], [55], and [85]) suggest model elements for capturing distributed features of a SUS (e.g., `CallNode` and `ReplyNode` [85], `timeStampSource` and `timeStampDestination` [141], `name` attribute of the `Thread` class and `timeStamp` [55]). Only one work [141] suggests what elements and their relationships to collect for distributed features of a SUS (e.g., `nodeID`). Kolbah, in her

[37] https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/perf.data-file-format.txt

hybrid approach, suggested a model (called the trace model, Fig. 3 on page 54) to capture sequential features of a SUS for the trace content.

2.2.3 Related work on instrumentation optimization

Looking closer at the dynamic analysis, when instrumenting a program at the source code level (or at an intermediate or at the binary level), each phase of the instrumentation can contribute to overhead. These phases are: a) selecting the probe insertion points and the proper probes for each point in a SUS, b) executing probes at specified points. The probe selection phase identifies interception points and the probes to be executed at each point. The probe execution phase includes executing probes, fetching data, and analyzing data either online (e.g., verifying temporal logic properties or contract) or offline (e.g., storing data in the form of a trace). Depending on the instrumentation language, each of those phases can fully or partially happen at compile-time or runtime. Regardless, executing probes should happen at least partially at runtime. Therefore, an efficient instrumentation means minimizing the number of probes and probe insertion points as well as executing probes and collecting data (including data analysis) as fast as possible. Achieving such efficiency to develop an instrumentation strategy requires one to have a good understanding of the instrumentation language and compiler, because each instrumentation language provides different constructs with different overhead impact and each compiler provides different degrees of automation at compile time or runtime for each phase, thereby directly impacting the instrumentation strategy.

In this section, first we discuss the performance of different instrumentation mechanisms other than AspectJ interception mechanism (Section 2.2.3.1). Next, we

discuss some related work on improving the performance of AspectJ instrumentations (Section 2.2.3.2). Finally, we look into some related work for efficient trace generation (Section 2.2.3.3).

2.2.3.1 Optimizing instrumentation for performance

Discussing overhead reduction of an instrumentation (or probing) technique generally includes reducing the number of probe insertion points, reducing the number of probes, reducing the cost of probe execution and reducing the cost of data collection (e.g., [76, 144-150]). The number of probes and the cost of data collection directly depend on the intent of instrumentation: the more one wants to collect, the larger the number of probes and the more data collected by each probe. Reverse-engineering software to produce behaviour is among the most probe-intensive instrumentations compared to other forms of instrumentations (e.g., performance tuning [146] and, runtime verification [147]). Reducing overhead due to those characteristics is exactly what triggered our use of a hybrid technology [151].

At the operating system level, instrumentation tools take advantage of OS kernel to trace program execution. For instance, System Tap[38] and Perf Events[39] instrument programs based on the Linux kernel, while DTtrace[40] does program instrumentations based on the BSD or Mac kernel. These tools are usually efficient to trace program execution due to running at a very low level, yet it is not possible to capture specific application behaviour (e.g., caller's class and method name) with these tools since they do not provide proper programming constructs.

[38] https://sourceware.org/systemtap/wiki
[39] https://perf.wiki.kernel.org
[40] http://dtrace.org

In this section, we rather discuss overhead reduction of probing, which is about weaving, and also a very context dependent issue: one tries to optimize weaving for C++ very differently from Java simply because of the characteristics of the target languages. Some general weaving optimization principles may apply regardless of the technology and programming language and one may get inspired by AspectC++ weaving optimization solutions (e.g., [66]) when trying to optimize AspectJ weaving. We do not contribute to AspectJ and AspectC++ weaving mechanisms or to the AspectJ and AspectC++ languages: we consider this outside the scope of this work.

In the realm of Java programs, different technologies generally perform instrumentation at the byte-code level, yet the level of abstraction in these technologies when adding probes differ from one to another: either by adding probes to the Java source code and then transforming them to byte-code or by adding probes directly to the Java byte-code. Technologies such as AspectJ [60] , LTTng tracer [152] and DiSL [153] offer high level languages to facilitate probing to the Java source code whereas libraries such as ASM [154] and BCEL [155] provide APIs to directly change the byte-code. In the following, we take a deeper look at these technologies.

Byte-code manipulation technologies (e.g., ASM [154], BCEL [155], Javassist [156]) provide extensive libraries to make a wide range of changes in the byte-code such as changing basic blocks or loops. These technologies generally require knowledge of byte-code instructions. Compared to high level instrumentation languages, byte-code manipulation technologies are more efficient and allow developer more flexibilities in changing byte-code. Yet developing an efficient instrumentation (based on byte-code manipulation technologies) requires more efforts due to the lower level of programming.

Thus, we turn to higher level instrumentation languages in our solution due to lower programming efforts and the extensive knowledge of byte-code methods, which reduces applicability this type of instrumentation for other developers.

High level Java instrumentation languages such as AspectJ add probes to the source code. There are only a few such technologies other than AspectJ to implement probes at the Java programming level: LTTng and DiSL.

LTTng is a technology originally started with tracing Linux kernel events [157, 158]. Probes are inserted to trace kernel events based on tracepoint APIs (marker is also used in old versions of LTTng). After successfully tracing kernel events, tracepoint APIs have evolved to support userspace tracing [159, 160]. Userspace tracing provided enough ground to trace user applications such as C++ or Java applications. LTTng uses the Common Trace Format to store trace information, and trace storage can take place locally or remotely with TCP channels. Recent versions of LTTng [152] (since 2.7 onward) provide direct support to trace different context fields in an executing object (e.g., method name, time stamp) in Java, yet LTTng falls short to provide any API to trace the callee object in the executing object's context (there is no probe insertion point for "call", but only for "execution"). Looking deeper into LTTng based on our requirements, we can change our trace model to use "execution" instead of "call" to capture method calls, yet it is not possible to trace object identifiers with LTTng. In addition, there is no performance comparison in the literature between AspectJ and LTTng instrumentation for Java applications since LTTng is originally meant for low level instrumentation.

DiSL [153] is a very appealing, recent solution since the authors argue that it is equally expressive as AspectJ and as efficient as ASM (and therefore more efficient than

AspectJ), thereby having the advantages of both kinds of solutions without their drawbacks. This technology is built on top of the ASM library to increase abstraction level of ASM while keeping its efficiency. The overhead reduction reported by the authors on several case studies, when comparing DiSL to AspectJ is not precise enough to make a decision as to use one technology or the other in a specific context. Specifically, we know (Section 2.2.3.2) that several AspectJ constructs are very expensive (high overhead) and that several attempts have been made to remedy the situation; some authors also suggest efficient usages of some AspectJ constructs to reduce overhead. Unfortunately, the comparison between DiSL and AspectJ [153] does not disclose which of those constructs were used. This is an important piece of information that is missing since, as discussed later, we do not use those expensive AspectJ constructs.

Our own overhead study (see Section 4.9.2) shows that AspectJ itself is a very small contributor to overhead in our case. We conclude that, at the time of writing, there is no compelling argument showing that, in our context, we should use DiSL or LTTng rather than AspectJ.

2.2.3.2 Optimizing the performance of AspectJ programs

There are two general approaches to improve the performance of an AspectJ program: making an efficient use of the AspectJ language, and improving the AspectJ compiler or the JVM. Since we intend to use AspectJ as a toolbox, we only focus on the former and not contribute to the latter: we want to devise an efficient use of the AspectJ language for the purpose of reverse-engineering object interactions.

An efficient use of the AspectJ language requires both efficient weaving rules (pointcuts) and efficient concerns (advices and inter-type declarations). We report on the few works we have found that discuss efficient practices for AspectJ programming. Dufour et al. [161] suggest AspectJ programming guidelines for reducing overhead, noticing programmers impose considerable overhead to a base program (i.e., the SUS) when they use loose pointcuts, i.e., pointcuts that match too many join points, generic advices (in particular generic around advice), the cflow pointcut, or when they introduce too many new constructors through inter-type declarations. Similarly, the AspectJ reference books [60, 162] provide recommendations on how to improve the performance of AspectJ programs, such as an efficient use of APIs for dynamic context collection (join point APIs versus Java reflection APIs).

Programming and refactoring of Java programs for high performance computing (HPC) [163-171] is also related to our work since AspectJ declarations are written in Java and we want high performance Java code in AspectJ declarations to limit execution overhead. For instance, practices such as in-lining (fewer subroutines), constant propagation, dead code elimination, proper choice of data types, and using standard library routines are suggested to improve performance. We simply mention them here to indicate that we account for refactoring for performance improvement opportunities in our work.

An efficient implementation of byte-code weaving, i.e., improving the AspectJ compiler, involves optimizing either the AspectJ compiler or the JVM. A number of works suggest optimizations to the original AspectJ weaving mechanisms [58, 63, 172-179], such as the around advice, the cflow pointcut, or advice dispatch. Although we do

not follow this path of instrumentation optimization, we note we do not use computationally expensive AspectJ constructs for which optimizations have been proposed (around advices or cflow pointcuts). In our experiments, we optimize the implementation of AspectJ concerns, and use the standard AspectJ compiler and the standard JVM.

Since adding new concerns to core concerns adds overhead, an instrumentation alternative could be to manually hard code the new concerns into the SUS in Java instead of using (AspectJ) aspects. Studies [172, 180] reported the performance of an aspect program is equal to or better than the equivalent non-aspect program due to better encapsulation of advices.

Finally, one may argue that AspectJ performs faster with load-time weaving compared to compile-time weaving. On the one hand, compile time weaving reduces the runtime overhead by executing lookup at compile time. On the other hand, opposite to load-time weaving, compile-time weaving is unable to take advantage of runtime data and the JVM internal structure to implement optimizations for the inserted aspects [63]. However, empirical studies show that the AspectJ compiler causes less runtime overhead with compile-time weaving than with load-time weaving when there is a large number of classes loaded or join points executed [181]. Thus, we opted for compile-time weaving as this thesis targets industrial-sized systems.

2.2.3.3 Optimizing the performance of information collection and storage

As mentioned in Section 2.1.2, a dynamic analysis technique monitors and gathers different types of data from a SUS, and typically stores this data in a file as an execution trace using a specific format for offline consumption [39]. Capturing less data or

condensing the gathered data while preserving semantics and using an efficient storage mechanism can potentially reduce the overhead of dynamic analysis. Prominent works on the trace format for storing object interactions discuss encoding, condensing, and compacting data (by removing redundant data) in a trace file since such files tend to be huge [54, 55, 182]. For example, there are algorithms for lossless trace compression based on string compression [54], bytecode compression [183], and converting a call tree to a directed acyclic graph (DAG) [55, 182]. In a DAG similar subtrees occur only once. In our context, although these are valid objectives, we are more interested in reducing the overhead due to producing the data than the format of storage of the data in a file. A trade off needs to be found between the overhead due to producing the data, possibly condensed/compacted, and the amount of data to store. In our case, since we prioritized overhead reduction and we collect as few data as possible, which we believe is already condensed enough, we decided to not use any of the existing condensing/compacting solutions.

In another line of work, Baca minimizes the overhead of producing execution traces within procedures thanks to a hybrid solution [184]. He suggests a static analysis algorithm to identify and inject fewer probes into basic blocks based on the basic block's frequency of execution (compared to injecting into all basic blocks) to build a CFG. The CFG is then captured at runtime with a lesser overhead. We rather focus on optimizing function calls as we capture CFGs in the static analysis. In addition, path profiling techniques (e.g., [185]) measure the frequencies of path executions. Again, we rather focus on object interactions.

Despite all our efforts to minimize the size of traces, our instrumentation can produce large traces. For such large traces, storing trace on magnetic disk or transferring through network can delay CPU operations and this logging aspect of the instrumentation can become a candidate for performance optimization, because CPU and memory speeds outpace network and disk throughputs by a large margin. Related works, especially in the HPC area (e.g., [168, 171, 186, 187]), suggest practices to boost the performance of logging or transferring large data, such as: buffering in memory, decoupling the application process from I/O operations, pre-allocating files, using multiple disks for parallel data transfer (e.g., using RAID technology [188]), eliminating electromechanical nature of magnetic disk technology by using solid state technology (if commercially viable), avoiding serialization, and establishing non-blocking and multiple connections. Using logging frameworks (e.g., Log4J [189]) is another alternative for transferring data on a network; however, there is not enough credible information about the performance of existing logging frameworks and one may apply different frameworks to achieve a proper performance. Along the same line as HPC, data intensive applications (e.g., climate modeling) use expensive high speed networks (so called data grid) [190-193] with specific architectures (e.g., InfiniBand [194]) to transfer large quantity of data. Due to lack of access to such expensive infrastructure, we do not use such high-speed networks in our work.

3 An Existing Hybrid Reverse Engineering Approach

In this chapter, we describe Kolbah's hybrid approach in detail since we build on it and improve its performance. To formalize this approach and specify it from a logical standpoint so it can be analyzed in, and compared with, future work, Kolbah defined two models (class diagrams). The first model is for traces and another model is for control flow graphs; and she defined mapping rules between them using the OCL [195]. These rules are used as specifications to implement a tool to instrument code so as to generate traces, to analyze source code to create control flow graphs, and then transform (model transformation) an instance of the trace model and instances of control flow graphs (for several methods) into a UML scenario diagram. Recall that a UML scenario diagram is a specific case of a sequence diagram where a series of object interactions has shown based on one execution (or one scenario in a use case). Kolbah in her solution and we in our solution do not merge multiple scenario diagrams into a sequence diagram.

The main issues that drove her design of a solution were:

- Keeping instrumentation to a minimum required for collecting the necessary information. Kolbah only collects method calls: caller and callee objects, method signature, class name and line number of call;
- Uniquely identifying every object with a pair (class name, instance counter for that class), similarly to Leduc's technique [141];
- Collecting the right information from the source code to match trace information, specifically line number of method calls (including complete signature), and to identify control flow structures;

- Devising models to facilitate model transformations: i.e., the trace model is close to the UML 2 meta-model.

The remainder of this chapter first (Section 3.1) discusses the control flow (static) information and the trace (dynamic) information, that is, the models for `<<dataStore>>` `Control flow graphs` and `Traces` (Fig. 2), and then (Section 3.2) the tools to automate activities `a1`, `a3`, and `a4` (Fig. 2). The chapter is then concluded in Section 3.3.

3.1 Control flow and trace information

We refer the reader to the UML 2 standard meta-model description [196] for more details on the sequence diagram meta-model, and only discuss the trace model (Section 3.1.1) and control flow model (Section 3.1.2).

3.1.1 Trace model

The trace model (Fig. 3) models execution trace data and is very close in structure to the UML 2 Superstructure's `Message` components: elements `Log`, `MessageLog`, `MessageLogOccurrenceSpecification` and `MessageSort` map to the UML's `Interaction`, `Message`, `MessageOccurrenceSpecification` and `MessageSort`, respectively.

`Log` represents a single program execution and contains a sequence of `MessageLog`s. A `MessageLog` represents a message sent to the logger to signal the start of an execution between a sending object and a receiving object (the two associations to `MessageLogOccurenceSpecification`). In class `MessageLogOccurenceSpecification`, attribute `covered` is a `String` containing the identification of an object (a unique identifier representing an object of a class), to be eventually translated into a lifeline in the sequence diagram. `MessageLog`'s attributes specify the kind of message

(`messageSort` attribute), that is, a synchronous call or an object creation, the message's signature[41] (in the form `returntype package.class.calledMethodName(formal parameter types)`, i.e., the signature of the method being called), and the name of the class whose instance executes the called method (`bindToClass`). For a `MessageLog` instance, using `bindToClass` and `signature` attribute values, we know exactly which method in a hierarchy of classes actually executed, i.e., the data allow us to account for overriding. In the case of a static call, `bindToClass` contains the class defining this static method. This way, the transformation algorithm can determine the specific class and method invoked by the method call. `SourceLocation` (in `MessageLog`) specifies the location (`name` of the class and `lineNumber`) in the source code from where the logged method call has been made; this is the call site.

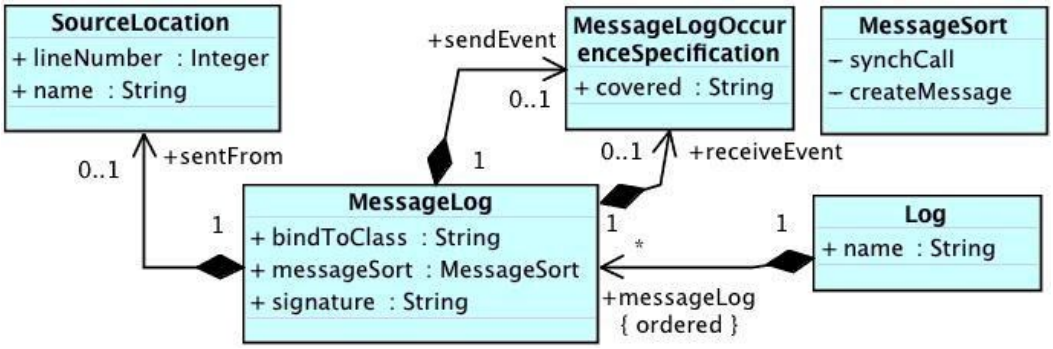


Fig. 3 Trace model [25]

3.1.2 Control flow model

The control flow model (Fig. 4) captures a method's code structure in terms of method calls (identified by line numbers), possibly performed under conditions (alternatives, loops). This model allows one to accurately locate method calls from the source code, based on matching `MessageLog`s from the trace model, and then place them

[41] Although collected at runtime, the signature includes the types of formal parameters, not the actual parameters' types. One can also collect dynamic types and parameter values but this was not necessary for Kolbah's purpose.

into the UML sequence diagram structure. Knowledge of a method call's host method and, if the method call is inside a condition, its control flow structures will allow one to accurately construct the scenario diagram with reduced dynamic (trace) data. Specifically, the value of attribute `covered` of a `MessageLog`'s `sendEvent` gives the class name from where the call captured by the log originated; the value of attribute `lineNumber` of the `MessageLog`'s `SourceLocation` then gives the exact statement of the calling class source code that executed. Looking for this line number of that class in the corresponding control flow model instance shows whether the statement (and therefore the call) was performed in a conditional statement and which method of the caller class performed the call.

In Fig. 4, a `Class` whose behaviour is monitored contains `Method`s, which, in turn, contain sequences of `CodeSection`s. A `CodeSection` can be a `MethodCall` (we do not distinguish constructors) or a `ConditionalSection`, possibly nested (a `ConditionalSection` contains a sequence of `CodeSection`s). A `ConditionalSection` is either an `Opt`, an `Alt` or a `Loop`. A `Loop` has a `LoopType` set to either `for`, `do` or `while`. Attribute `conditionDescription` (class `ConditionalSection`) specifies the actual condition. A `MethodCall` is performed at a specific `lineNumber` in a specific caller method (`isInMethod`) in a specific class (`isInClass`). `isInMethod` contains the signature of the method this method call is in. For `MethodCall`s outside a `ConditionalSection`, `method.signature` (navigating the association between `MethodCall` and `Method`, inherited from `CodeSection`, and accessing attribute `signature` of the calling method) equals `isInMethod`. However, `MethodCall`s inside a `ConditionalSection` do not have direct access to this association and, therefore, need

to carry the `isInMethod` attribute. Attribute `isInClass` contains the name of the `Class` the `MethodCall` is in, and is needed for similar reasons. An `Alt ConditionalSection` does not contain information about true/false branches because they are not handled as such in Kolbah's work: each branch (either true or false) is an `Alt` instance.

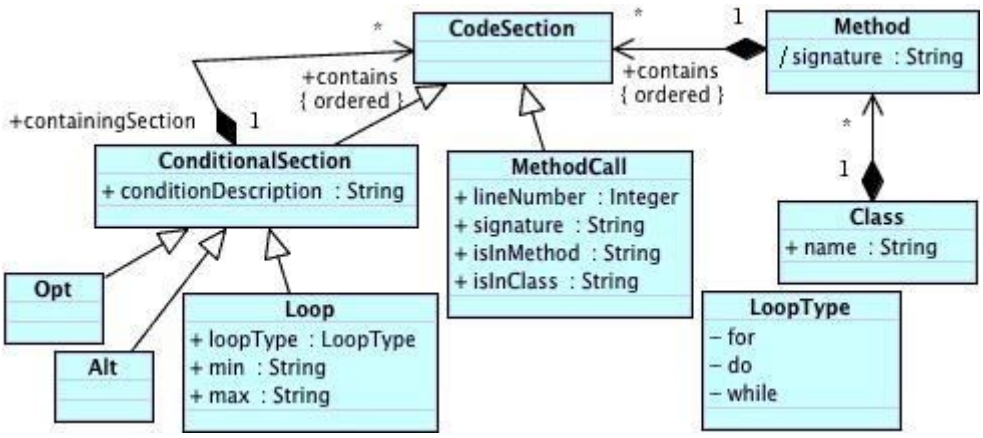


Fig. 4 Control flow model [25]

3.2 Tool support

Section 3.2.1 discusses the aspects that Kolbah used to collect runtime information (traces). Section 3.2.2 discusses the control flow model construction Kolbah designed. Section 3.2.3 discusses Kolbah's model transformation. An example illustrating the models and the model transformation is discussed in Section 3.2.4.

3.2.1 Aspects

The premise of the hybrid approach, in Kolbah's work, was to provide a lighter instrumentation strategy than Leduc's (purely) dynamic analysis technique. Kolbah tried to achieve a lighter instrumentation strategy by (1) avoiding instrumenting control flow structures in the source code [141] or the byte code [120] and (2) limiting the impact of aspects, that is, reducing the number of recorded logs. In implementing this lighter instrumentation, Kolbah followed good programming practices to improve maintenance,

modularity or usability for instance (e.g., use of Java standard data structures), which can adversely hurt the performance of her instrumentation. To avoid instrumenting the source code, since AspectJ [162] did not provide pointcuts for control flow structures (even `cflow` and `cflowbelow` pointcut designators do not help), Kolbah turned to static analysis (a control flow graph created by parsing the source code). It should be noted that, even if AspectJ were providing such pointcuts, combining static analysis with dynamic analysis would still be preferable to limit the probe effect as fewer aspects and pointcuts would be needed.

With a hybrid analysis, there needs to be a way to match static information to dynamic trace information. Collecting pairs (class name, method signature) for executing methods is not sufficient as we know nothing about the caller. Instead of instrumenting method executions [120, 141, 197], Kolbah instrumented method calls, as this allows one to collect information about both the caller (line number and the source file name from where the call was made) and the callee (class name, object identity). Combined with a unique identification of executing objects, the hybrid technique can correctly link dynamic and static data, as information on class names and line numbers is also in the control flow graphs.

Once Kolbah can associate a method call from the trace (`SourceLocation`'s attributes) with the location in the source code where that call is made (`MethodCall`'s attributes), the static analysis allowed her to determine from which method in which class the call was made and whether this call is inside a condition or a loop.

Furthermore, when using a `call` join point, AspectJ can provide information about both the source and destination methods, as opposed to an `execution` join point,

which only provides information about the callee. Since control flow is obtained statically, there is no need to detect the start and end of executing methods, that is, the hybrid approach does not need an `around` advice: it only needs a `before` advice, which further reduces the number of log statements compared to Leduc's work [141] (Recall that Leduc's dynamic approach uses `around` advices to capture method call information to later merge this information with control flow information). This improvement was expected to translate into significantly lower overhead and faster executions. One drawback, however, is that `call` join points cannot catch calls made with `super()` or `this()` while `execution` join points can (as per the AspectJ specification). So, such calls would not be captured in the Kolbah's hybrid solution as well as our solution.

We now review the implementation of the dynamic analysis tool, i.e., the AspectJ aspects Kolbah designed, as we will refer to this implementation when we attempt to optimize this hybrid approach in Chapter 5. The dynamic analysis tool adds a generic `instrument` package[42] to the software source code, and then compiles the source code with the AspectJ compiler. The `instrument` package includes: two aspect classes (`MethodAspect`, `IdentifierAspect`), a Java interface (`ObjectID`), and a `Logger` class, which we discuss next in Sections 3.2.1.1, 3.2.1.2 and 3.2.1.3, respectively.

3.2.1.1 The MethodAspect class

Kolbah designed three advices and used seven methods within those advices in the `MethodAspect` class. Fig. 5 (excerpt) highlights the general structure of the

[42] Note that another name could be selected if the default (`instrument`) name conflicts with a package name of the reverse-engineered software. Such a conflict is easy to fix automatically, and using the default name does not reduce the generality of the solution. Similarly, it is important to note that the aspects are defined in an abstract way (i.e., definitions are not specific to any Java class to instrument), and provide templates that can be automatically tailored to instrument any Java class.

`MethodAspect` class, showing pointcuts, advices, and method definitions. Next, we will further discuss each advice in detail.

Kolbah defined pointcuts `callMethod`, `callStaticMethod`, and `callConstructor` (lines 3 to 7 in Fig. 5) intercept all calls to methods (either static or not) and constructors in the System Under Study (SUS), respectively. The pointcuts ensure that they only intercept classes in the SUS by excluding calls to `objectIDgenerator` and methods in the `instrument` package. Then, before any call to static methods (line 46), non-static methods (line 9), or constructors (line 74) happens during the execution of the instrumented SUS, the advice corresponding to that specific pointcut is executed. Kolbah opted for `before` advices instead of `after` advices since `before` advices ensure that the right order of events (calls) is captured.

```
1    package instrument;
2    public aspect MethodAspect {
3       pointcut callMethod() : call (!static * PackageName..*(..));
4       pointcut callStaticMethod():call(static * PackageName..*(..))
5    && !call (*PackageName..objectIDgenerator(..));
6       pointcut callConstructor() : call (PackageName..new(..)) &&
7    !within (instrument..*);
8
9       before(): callMethod () {
…          //Content of advice: see Fig. 6
44      }
45
46      before(): callStaticMethod () {
…          //Content of advice: see Fig. 7
72      }
73
74      before(): callConstructor () {
…          //Content of advice: see Fig. 8
102     }
103
104        //Helper methods: see Fig. 9
…             .
129           .
130  }
```

Fig. 5 Excerpt of the MethodAspect aspect class

Fig. 6, Fig. 7, and Fig. 8 illustrate code excerpts for advices `before():callMethod()`, `before():callStaticMethod()`, and `before():callConstructor()`, respectively. We discuss these advices in sequence.

The `before():callMethod()` relies on the capability of the instrumented code to count classes' instances (see Section 3.2.1.2), and reports on the unique identifiers of those instances: classes implement interface `ObjectID`, which defines method `getObjectID()`. In addition, the `before():callMethod()` and `before():callStaticMethod()` advices intercept calls to methods (either static or not) and collect information before they are made: `call` join point, `before` advice (Fig. 6, and Fig. 7). The advice `before():callMethod()` collects information about both the caller and the callee of a method call while the `before():callStaticMethod()` advice collects information about the caller only: the callee is the class receiving the call.

```
9    before(): callMethod ()    {
10      String thisID = new String();
11      LinkedList log = new LinkedList();
12      if (thisJoinPoint.getThis() != null) {
13         try {
14         thisID = ((ObjectID) thisJoinPoint.getThis()).getObjectID();
15         }
16         catch (ClassCastException e) {
17            thisID = "Caught NonInstrumentedCaller";
18         }
19      } else {
20         thisID = getStaticClassName(thisJoinPointStaticPart.
21   getSourceLocation().toString());
22         }
23      String targetID = new String();
24      try {
25         targetID = ((ObjectID) thisJoinPoint.getTarget()).get ObjectID();
26      } catch (ClassCastException e) {
27         targetID = "Caught NonInstrumentedCallee";
28      }
29      log.add("Dynamic<messageLog bindToClass=\""
30         + MethodAspect.getBindToClassName(thisJoinPoint
31   .getTarget().toString())
32         + "\" messageSort=\"synchCall\" signature=\""
33         + MethodAspect.getMethodSignature(thisJoinPoint.toString()) +
34   "\">");
35      log.add(" <sendEvent covered=\"" + thisID + "\"/>");
36      log.add(" <receiveEvent covered=\"" + targetID + "\"/>");
37      log.add(" <sentFrom lineNumber=\""
38         + MethodAspect.getLineNumber(thisJoinPointStaticPart
39   .getSourceLocation().toString() + "\" name=\""
40         + MethodAspect.getFileName(thisJoinPointStaticPart
41   .getSourceLocation().toString()) + "\"/>");
42      log.add("</messageLog>");
43      Logger.getLoggingClient().instrument(log);
44   }
```

Fig. 6 before():callMethod implementation in MethodAspect

The collected information in advices `before():callMethod()` (Fig. 6), and `before(): callStaticMethod()`(Fig. 7) includes: unique identifiers of interacting objects (or classes), the signature of the method being called (including formal parameter information) in lines 33 and 62, as well as the name of the class whose method signature is captured in lines 30 and 60, the line number and the file name from where the call is made, i.e., in the caller (lines 38-41, 66-69). Both advices collect the unique identifier of the caller object in lines 14 and 51 (or class name in the case of static methods in lines 20-21 and 56-57). While there is no need to write additional code to get the callee's information when a static method is called (the name of the class is collected in line 60),

the `before():callMethod()` advice collects the callee's unique identifier in line 25. Note that the `instrument` package should necessarily handle calls between the SUS code and code outside of the SUS code itself (e.g., third party library, JVM/JRE); therefore, every time an advice collects information regarding unique identifiers of interacting objects, it accounts for the possibility that the caller or the callee is not instrumented and therefore does not have a unique identifier (lines 17, 27, 53).

```
46   before(): callStaticMethod () {
47      String thisID = new String();
48      LinkedList log = new LinkedList();
49      if (thisJoinPoint.getThis() != null) {
50         try {
51         thisID = ((ObjectID) thisJoinPoint.getThis()) .getObjectID();
52         } catch (ClassCastException e) {
53            thisID = "Caught NonInstrumentedCaller";
54         }
55      } else {
56         thisID = getStaticClassName(thisJoinPointStaticPart
57   .getSourceLocation().toString());
58      }
59      log.add("Static<messageLog bindToClass=\""
60         + MethodAspect.getStaticBindToClassName(thisJoinPoint.toString())
61         + "\" messageSort=\"synchCall\" signature=\""
62         + MethodAspect.getMethodSignature(thisJoinPoint.toString())+ "\">");
63      log.add(" <sendEvent covered=\"" + thisID + "\"/>");
64      log.add(" <receiveEvent covered=\"" + thisID + "\"/>");
65      log.add(" <sentFrom lineNumber=\""
66         + MethodAspect.getLineNumber(thisJoinPointStaticPart
67   .getSourceLocation().toString())  + "\" name=\""
68         + MethodAspect.getFileName(thisJoinPointStaticPart
69   .getSourceLocation().toString()) +  "\"/>");
70      log.add("</messageLog>");
71      Logger.getLoggingClient().instrument(log);
72   }
```

Fig. 7 before():callStaticMethod implementation in MethodAspect

A constructor of an object instance is intercepted with the advice `before():callConstructor` to collect the same information as in the `before():callMethod()` advice except for one piece of information: the unique identifier of the called constructor's object (Fig. 8). Because the `before():callConstructor()` advice executes before object creation has completed, it would not be possible to get the `objectID` (implemented by the `getObjectID()` method) to uniquely identify the object

being created by the called constructor. Koblah's solution, however, compensates for this missing information by reporting on each object's unique identifier as soon as the object is created (Section 3.2.1.2). The information reported in the `before():callConstructor()` log statement and the object identification log statement (Section 3.2.1.2) are then merged during post-processing of a trace.

```
74   before(): callConstructor () {
75      String thisID = new String();
76      LinkedList log = new LinkedList();
77      if (thisJoinPoint.getThis() != null) {
78         try{
79            thisID = ((ObjectID) thisJoinPoint.getThis()) .getObjectID();
80         }
81         catch (ClassCastException e) {
82            thisID = "Cought NonInstrumentedCaller";
83         }
84      } else {
85         thisID = getStaticClassName(thisJoinPointStaticPart
86   .getSourceLocation().toString());
87      }
88      log.add("<messageLog bindToClass=\""
89         + MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
90         + "\" messageSort=\"createMessage\" signature = \"new "
91         + MethodAspect.getMethodSignature(thisJoinPoint.toString())+ "\">");
92      log.add(" <sendEvent covered=\"" + thisID + "\"/>");
93      log.add(" <receiveEvent covered=\"nothing\"/>");
94      log.add(" <sentFrom lineNumber=\""
95         + MethodAspect.getLineNumber(thisJoinPointStaticPart
96   .getSourceLocation().toString())
97         + "\" name=\""
98         + MethodAspect.getFileName(thisJoinPointStaticPart
99   .getSourceLocation().toString ()) + "\"/>");
100     log.add("</messageLog>");
101     Logger.getLoggingClient().instrument(log);
102     }
```

Fig. 8 before(): callConstructor implementation in MethodAspect

Advices use private static methods (Fig. 9) to extract required information from join point such as: class name (line 111) or line number (line 105).

```
104         //Helper methods
105   private static String getLineNumber(String s) {
106      return s.substring(s.indexOf(":") + 1);
107   }
108   private static String getFileName(String s) {
109      return s.substring(0, s.indexOf(":"));
110   }
111   private static String getBindToClassName(String s) {
112      if (s.contains("@")) {
113         s = s.substring(0, s.indexOf("@"));
114      }
115      return s;
116   }
117   private static String getStaticClassName(String s){
118      s = s.substring(0, s.indexOf("."));
119      return s + "_Static";
120   }
121   private static String getStaticBindToClassName(String s) {
122      return s.substring(s.indexOf(" ") + 1, s.lastIndexOf("."));
123   }
124   private static String getNewBindToClassName(String s) {
125      return s.substring(s.indexOf("(") + 1, s.lastIndexOf("("));
126   }
127   private static String getMethodSignature(String s) {
128      return s.substring(s.indexOf("(") + 1, s.lastIndexOf(")"));
129   }
```

Fig. 9 helper methods

3.2.1.2 Object unique identification

Kolbah uses the inter-type declaration mechanism in AspectJ in `IdentifierAspect` to implement an object identification mechanism. The `instrument` package contains the `ObjectID` interface, which simply defines a `getObjectID()` method that returns a `String` supposed to uniquely identify an instance of a class in the set of instances of that class (Fig. 10).

```
package instrument;

public interface ObjectID {
   public String getObjectID();
}
```

Fig. 10 ObjectID interface

The `IdentifierAspect` adds the `ObjectID` interface to the set of interfaces implemented by all the classes in the SUS (Fig. 11). For each class in the SUS, a static method `objectIDgenerator()` is defined to keep track of the number of instances that

are created from the class: this method implements an extension of the singleton design pattern to count the number of instances of a class and assigns a new unique number to each new instance of the class: line 1 adds a private attribute holding that unique number to each object of the class. It is also here where the class name and the new instance unique number is logged in the trace file to be later merged with the `before():callConstructor()` log (line 8).

```
1    private int ClassName.objectID = ClassName.objectIDgenerator(objectID);
2    private static int ClassName.currentObjectID = 1;
3    private static int ClassName.objectIDgenerator(int i) {
4       int id = i;
5       if(i<1){
6          LinkedList log = new LinkedList();
7          id = ClassName.currentObjectID++;
8          log.add("<lifeline className=\"ClassName\" name=\"className_"
9    + id + "\"/>");
10         Logger.getLoggingClient().instrument(log);
11      }
12      return id;
13   }
14   declare parents : ClassName implements ObjectID;
15   public String ClassName.getObjectID() {
16      if (objectID < 1){
17         objectID = ClassName.objectIDgenerator(objectID);
18      }
19      return "ClassName_" + objectID;
20   }
```

Fig. 11 Identifying the last object instance

In the case of the creation of an instance of a class that belongs to an inheritance hierarchy, this object identification mechanism generates a log from the class being instantiated but also from each parent class: as if not only an instance of the class was created but also an instance of each parent class. This does not impact advices since `getObjectID()` returns the unique identifier (value of attribute `objectID`) of the object being created, not the parents'. The only drawback is spurious logs: the ones for the parents. Therefore, all parents' logs have to be removed from the trace during post-processing. Although during the process of calling the constructor of an inherited class the constructors of all parents got executed in Java, the UML sequence diagram only

shows the instance of inherited class in the form of "lifeline" [198]. Another side effect is that not all consecutive integer values are necessarily used to represent instances of a given class. For instance, if class A is instantiated (generating id value 1), its child class B is instantiated (generating value 1 for B, and incrementing the count for A by 1), and a new instance of A is created, this last instance will have a unique identifier of 3; value 2 is not used. This is not a problem since what matter is that each instance be uniquely identified.

3.2.1.3 Logging mechanism

Kolbah's solution uses a generic logger that logs the information passed from the advices through calls to `Logger.getLoggingClient().instrument(log)` to a file (Fig. 12, lines 22-33). The `Logger` class implements the singleton design pattern.

Based on this logger implementation, the logger records the log information to a file each time an advice needs to record some log information; therefore, we can expect a large of number accesses to the disk for any typical program.

```
1    package instrument;
2
3    public class Logger {
4       public static Logger instance = new Logger();
5       private FileWriter filewriter = null;
6
7       public static Logger getLoggingClient() {
8          return instance;
9       }
10
11      public FileWriter getFileWriter() {
12         try {
13            if (filewriter == null){
14               filewriter = new FileWriter(new File("Trace.txt"));
15            }
16         } catch (IOException e) {
17            System.out.println("Error in Logging Client: " + e);
18         }
19         return filewriter;
20      }
21
22      public void instrument(List <String> record) {
23         FileWriter writer = getFileWriter();
24         try{
25            for (int count = 0; count < record.size(); count++) {
26               writer.write((String) record.get(count));
27               writer.write("\r\n");
28            }
29            writer.flush();
30         } catch (IOException e) {
31            System.out.println("Error in Logging Client: " + e);
32         }
33      }
34   }
```

Fig. 12 Logger implementation

3.2.2 Control flow

Kolbah created a JavaCC (with JJTree) parser [199] to generate instances of the control flow model, using a simplified Java grammar for Java version 1.5. Only a simplified Java grammar was necessary since not all characteristics of the program needed to be identified: only the control flow. The generated instance of the parser returns class definitions (including inner class definitions), method/constructor definitions, method and constructor calls (including those passed as arguments of other calls or those used in control flow structures), and control flow structures. The parser can handle if, else if and else, while loop, for loop (including for-each) but not ?: and the do-while loop (doing so is not a technical challenge). It does not handle exceptions, which

Kolbah said would be considered in the future. Note that when a condition contains a method call, the method call appears in the control flow graph right ahead of the condition (outside of the conditional control flow construct). This is to better match the trace information and the UML sequence diagram notation.

3.2.3 Model transformation

Kolbah formalized the different steps of the transformation of instances of the trace and control flow models into an instance of the UML (sequence diagram) meta-model in terms of mapping rules between these models, using the OCL [195]. She created eleven such rules, which range from four to 50 lines of OCL to: (1) match every single message log to a message in a sequence diagram, (2) match control flow structures to combined fragments, and (3) ensure the order of messages matches the order of logs.

The transformation was then performed with MDWorkbench[43], a commercial Eclipse-based IDE for model driven development that provides a model transformation capability. Transformations are rule-based, specified in a proprietary, imperative model transformation language, and can transform any number of source models into any number of target models.

The OCL rules can be seen as specifications for the transformation and were used to identify whether the trace and control flow models had the required information to accurately perform transformations. They were also used to ensure partial correctness of the generated diagrams.

Kolbah specified models using XML Schema. MDWorkbench uses these rules to manipulate instances of the trace and control flow models (XMI input files), producing an

[43] www.mdworkbench.com

instance of the UML sequence diagram meta-model (XMI output file). This XMI output file is ready to be used by any UML CASE tool.

3.2.4 Illustrating example

We now illustrate Kolbah's approach on a synthetic example of our own. Fig. 13 shows (excerpt) the source code of class A (part I), the control flow model of classes A and B (parts II and III), and the trace model instance for one execution. We only focus on the important aspects of Kolbah's work that allow us to illustrate the approach. As a result, some information is simply not available in the figure; for example, attribute values of the `SourceLocation` and `sendEvent` `MLOS` objects linked to `ML5` refer to source code information of the call to `m()` on an instance of `A`, which is not in part I of the figure. Although not used in the example, the hybrid approach handles method parameters and return values as discussed in Sections 3.1.1 and 3.1.2 and illustrated in Fig. 3 and Fig. 4.

We assume reference `b` (part I) is an instance of class `B`. The `lineNumber` attribute values in parts II and IV correspond to the line numbers in part I. In part IV, `MLOS` simply refers to `MessageLogOccurenceSpecification`. In part II, the control flow recognizes that `m()` contains an `Opt` alternative, which is, itself, made of a sequence of a `Loop` (performing a method call to `n()` on `b`) followed by a method call (to `m()` on `b`). The numbers `1` and `2` on the `Loop` and `MethodCall` sides of the links simply indicate that the links between the `Opt` object, and the `Loop` and `MethodCall` objects are ordered (`CodeSection`s are ordered in a method, Fig. 4). In part IV, we assume that the execution of the program resulted in four initial log messages, followed by log messages `ML5`, `ML6`, …, `ML15`. Since the loop is executed nine times, there are nine structures similar to `ML6` and its linked instances in the sequence of `MessageLog` instances linked to the `Log`

instance (Fig. 3). Instances of objects executing methods are uniquely identified by the aspects, which is represented in part IV as strings "A.1" and "B.10", suggesting that the instance of A executing m() is the first instance of A ever created in the program, and that calls to n() and m() are performed on the tenth instance of B created.

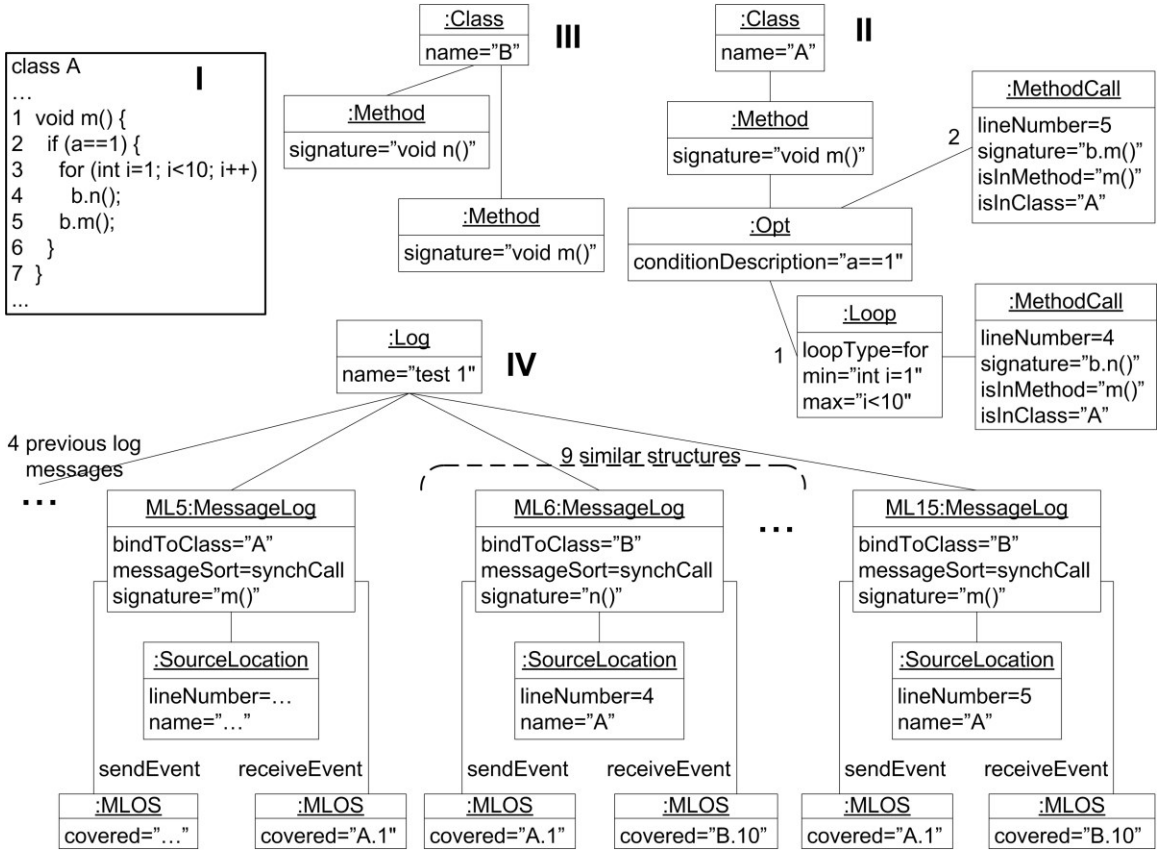


Fig. 13 Illustrating example (excerpt) for Kolbah's solution: (I) source code for class A; (II) control flow model for class A; (III) control flow model for class B; (IV) trace model for one execution

Let us now illustrate the essence of the model transformation in Kolbah's hybrid approach. The trace model instance shows two executing objects: instance 1 of A and instance 10 of B. This allows the transformation to create two lifelines, one for each of these instances. Instance ML5 shows the call to m() on instance 1 of class A: the receiveEvent MLOS linked to ML5. The following MessageLog in the sequence from the Log instance, specifically ML6, shows a call to n() on the tenth instance of B (the

receiveEvent MLOS linked to ML6) performed by the first instance of A (the sendEvent MLOS linked to ML6). Since there is no other MessageLog between ML5 and ML6 in the sequence, and the receiveEvent of ML5 and the sendEvent of ML6 have the same covered attribute value (A.1), we can conclude that the call to n() in ML6 is performed by m() which has been called in ML5. This allows the transformation to create an execution specification on the lifeline for A.1, showing the execution of m(), another execution specification on the lifeline for B.10, showing the execution of n(), as well as a message from the m() execution specification to the beginning of the n()'s execution specification. The same principle applies to the eight other MessageLog instances similar to ML6, as well as ML15, resulting in the sequence diagram of Fig. 14 (a).

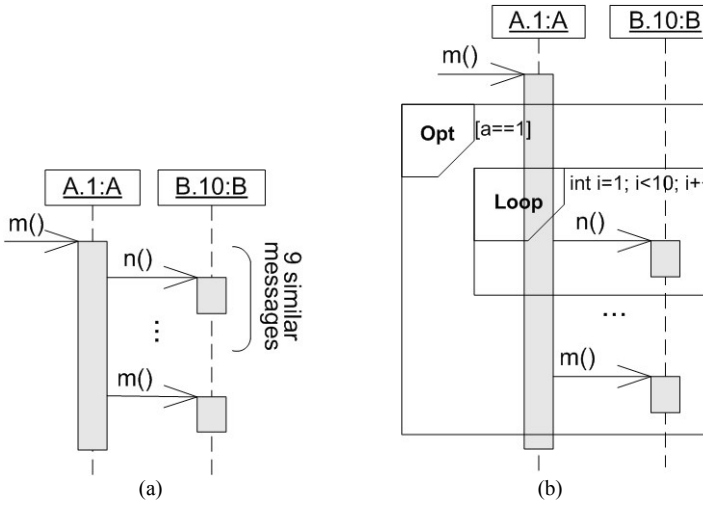


Fig. 14 Illustrating the model transformation

The transformation also recognizes that the call to n() on an instance of B recorded by ML6 occurs at line 4 (attribute lineNumber of the ML6's SourceLocation instance), that this call is performed in method A.m() (which we already discovered), and that, from the control flow model of method A.m(), the call to n() on an instance of B at line 4 happens in a loop, which, itself, happens in an alternative. This applies to all the

`ML6` to `ML14` objects: we therefore obtain nine `Loop` combined fragments each containing a message labeled `n()`: Fig. 14 (b). Last, since `ML15` is a call that happens in `A.m()` at line 5 and that this call happens (control flow model) in the alternative, after the loop, the hybrid approach can correctly place message `m()`: Fig. 14 (b).

Collapsing the nine `Loop` combined fragments and their message labeled `n()` into one message in one `Loop` combined fragment, itself inside an `Opt` combined fragment, is not an easy task in the general case and this was mentioned as future work by Kolbah. In the simple example discussed here, it would be easy to do that. However, in general, the contents of the loop instances may not be the same (as different control flows may be triggered in the loop), making the merging difficult.

3.3 Conclusion

In this chapter, we presented Kolbah's hybrid technique, which relies on both static (control flow) and dynamic (execution trace) information, to automatically reverse engineer scenario diagrams. We focused on important aspects of her hybrid approach, specifically how to reduce the amount of instrumentation of the bytecode and avoiding instrumenting the source code. This is particularly important as Kolbah tried to reduce the risk of not observing the right behaviour when executing the instrumented program.

We reviewed how aspects in Kolbah's hybrid approach could limit the overhead of trace generation (e.g., tracing calls rather than executions, with a before rather than an around advice). In parallel, we looked at Kolbah's generated control flow graphs of methods in which she was only interested in method definitions, the method calls they trigger, and the conditions under which those calls are triggered. We showed how Kolbah represented both sets of information using UML class diagrams. Kolbah devised a model

transformation that transforms an instance of the trace model and instances of the control flow model into an instance of the UML sequence diagram meta-model. In the next chapter, we will investigate in detail how Kolbah's solution behaves with respect to overhead, which will drive our search for an even better (i.e., with less overhead) solution.

4 Investigating the Overhead of Kolbah's Hybrid Approach

In Chapter 3, we explained Kolbah's hybrid approach. In this chapter, we will investigate the instrumentation overhead of this hybrid approach by conducting various experiments, with different case studies. We show how much the instrumentation overhead is reduced in the hybrid approach compared to the original purely dynamic approach of Leduc (Section 4.7), how the instrumentation overhead in the hybrid approach can change the behaviour of a SUS (Section 4.8), and what are the potential areas we can consider to reduce overhead in Kolbah's hybrid approach (Section 4.9).

During this chapter and the next one, we mainly follow the software engineering experimentation outline proposed by Wohlin et al. to describe experiments [200]. First we define the specific <*objective*> of the experiment (i.e., what is the question we want to answer and what we don't want to answer). Then, we explain the <*experiment design*> to make the experiment reproducible by someone else. We answer questions such as: What is the context of the experiment and what are the hypotheses? These can include: defining variables necessary to measure, explaining the strategy for measurement and data analysis; How is the experiment designed? This includes: the required preparations, the execution; What is the validation procedure of data collection? How the data is presented? This includes: analysis model, sample size, outliers. Finally, we interpret <*results*> to give an understanding of the presented data and conclude the experiments.

There are commonalities among the experiments we will conduct, especially in terms of *experiment design*. For instance, we use the same case studies to answer research questions (RQs): this is true for this chapter as well as the following one (Chapter 5). Therefore, we exclude these commonalities from the *experiment design* of each

experiment and discuss them in separate subsections: the description of case study systems (Section 4.2), the executions we will be using for each case study (i.e., test cases) (Section 4.3), the measurement framework used within each experiment (Section 4.4), and the criteria we use to select proper case studies for an experiment (4.5).

The reminder of this chapter is organized as follows: we first define the Research Questions (RQ) in Section 4.1, and then explain the case studies used to answer RQs in Section 4.2. In Sections 4.3 and 4.4, we present the experiment set-up and measurement framework used for all experiments. In Section 4.5, we define a set of criteria to select proper case studies based on experiment requirements. In Section 4.6 we present the experiment design we used for studying execution overhead. In Sections 4.7, 4.8 and 4.9, we report on case studies to answer RQs. The experiments are conducted based on Kolbah's hybrid approach and Leduc's dynamic approach. We conclude this chapter in Section 4.10.

4.1 Research Questions

We used a number of case study systems with three research questions in mind:

(RQ1) Is the execution overhead, measured as execution time, reduced when the hybrid approach is used compared to a purely dynamic approach and if so what is the amount of reduction?

(RQ2) In case of overhead reduction in the hybrid approach, can this instrumentation approach, because of overhead, change the software behaviour?

(RQ3) How much does each component of the hybrid approach contribute to overhead?

To answer RQ1, we only compare to one purely dynamic approach, Leduc's, since we identified in Section 2.2.1 this is the closest approach to Kolbah's hybrid approach, which we build upon. Furthermore, Kolbah already investigated RQ1 with limited size case studies [25]. While we re-do Kolbah's experiments with larger numbers of executions, we additionally focus more on larger case studies to answer RQ1. In addition, before further reducing the overhead of the hybrid approach, we believe it is important to show the potential of inaccurate software behaviour observation due to the probe effect by asking RQ2. Although the existence of the probe effect has been illustrated in peer-reviewed journal and conference papers [41, 70, 78], studying the probe effect of the hybrid technique in a multi-threaded system is necessary to justify any future efforts towards further reducing overhead. Finally, we answer RQ3 before starting the optimization process, and identify where to optimize, in Chapter 5. Before trying to optimize, we want to identify where it is worth trying to optimize.

For all RQs, we are only interested in measuring the dynamic characteristics of the hybrid approach (i.e., instrumentation); therefore, we are not actually reverse-engineering and visualizing scenario diagrams. Kolbah already showed that her traces and Leduc's traces were similar enough[44] to produce identical sequence diagrams; therefore we can perform comparisons without looking at sequence diagrams; we nevertheless ensure, though this is not systematically reported, that traces are indeed the same.

[44] They are similar enough in the sense that information in a trace produced by Kolbah, merged with static information she collects, is equivalent to the information in a trace produced by Leduc.

4.2 Case studies

To answer these research questions, we relied on five different case study systems[45]. The first system, *Overhead Emulator (OE)*, is a short program (40 lines of Java code) Kolbah built specifically to exercise and control many different situations (e.g., nested loops and numerous iterations of loops). Through a command line argument, we can control the number of times method calls are performed (including calls to different objects), loops are executed, thereby simulating larger program executions. Typically, a command line argument value of n approximately leads to n loop iterations, 5*n-3 if-then or if-else executions, and 5*n method executions. The code for OE does not contain any computation, any GUI, any interaction with IO devices (e.g., reading a file). This should allow us to evaluate to what extent the hybrid technique reduces the overhead as the size of the software (simulated by increasing the number of loops and calls) increases, without actually using larger software. Specifically, we use OE as a synthetic case study to mimic large number of method calls; given we instrument to intercept method calls, varying the number of such calls will allow us to study scalability and overhead as such a number increases. Note, however, that we expect execution overhead results to be worse with OE than with a real system exercising a similar pattern of calls and control flows, since we do not have any computation in OE. In other words, the percentage of increased execution time would be smaller than what we report with OE.

The second system, *PCC Prover*, implements the Proof Carrying Code (PCC) technique [201], a technique for safe execution of untrusted code, and was implemented by Kolbah.

[45] The source code for case studies and experiments is available at: http://sce.carleton.ca/~mehrfard/repository/

The third system is a simple *Calculator*, partly generated by JavaCC, that implements the Visitor design pattern.

The fourth system, *Weka*[46], is a well-known machine learning and data mining software. It is a much bigger, industry-size software system to experiment with.

Finally, we developed a simple *Producer-Consumer* based on the standard producer-consumer model [202] to investigate the possible change in software behaviour due to the probe effect (instrumentation overhead). The producer-consumer system implements a producer thread, a consumer thread, and a FIFO queue. Once the main method transfers execution to the producer thread, the producer thread creates an empty object, puts the object in the FIFO queue, and then pauses its execution (delay caused by calling the Java `Thread.sleep()` method) for a specified time period (deadline); Simultaneously, the consumer thread checks the queue constantly to take out any object it may contain, consumes it, and triggers a delay. The delay mechanism in the consumer corresponds to what a consumer would do to process the contents of the queue. We simulate, and control this delay by calling OE within the consumer. If the FIFO queue is full when the producer wants to deposit an element, the producer throws the `QueueFull` exception. This design gives us the opportunity to set a constant delay in the producer, vary the consumer delay (different OE input values), and study the impact of instrumentation (the computation is traced). It should be noted that the capacity of the FIFO queue is 16 objects.

Table 1 summarizes characteristics of the five case studies we used. The third and fourth columns in the table count the number of classes (accounting for inner classes) and lines of code (without counting blank lines and comment lines), which we obtained

[46] http://www.cs.waikato.ac.nz/~ml/weka/, Weka version: 3.7.7

thanks to the Eclipse Metrics[47] plugin. The "NMC_{SUS}" (Number of Method Calls in the SUS) column reports on the total number of calls to constructors, static and non-static methods we observed, thanks to a dedicated simple AspectJ aspect (Appendix B), when executing the different case studies with specific test suites (column "Test case"). OE_TC2 triggers a similar number of method calls as Weka_TC4 execution, illustrating the possibility to mimic, with OE, numbers of method calls similar to what is triggered in large, real software. Thus, if observations we make when studying overhead with Weka are similar to those we make when studying overhead with OE_TC2, we will have confirmation, or at least will be more confidant that OE can indeed help to mimic real software of various sizes (in terms of triggered method calls), which will strengthen the observation we make with OE. The "Max DIT" column indicates the maximum Depth of Inheritance Tree (DIT) among the classes in each case study, which we obtained again thanks to the Eclipse Metrics plugin. Recall that the Depth of Inheritance Tree metric is the maximum length of a path from a class to a root class in the inheritance hierarchy of a system [203]. We use DIT, among other code complexity metrics, as it will help us to explain some results. Fig. 15 shows the number of classes in Weka and OE with various DIT values. Among 1,025 classes in Weka (not accounting for inner classes[48]) thirty percent (308 classes) have one ancestor (DIT=2). On the other hand, among four classes in OE, there is only one class with one ancestor and the DIT for other classes is equal to one. Zero and one in this metric indicate the existence of interface and a class with no inheritance respectively.

[47] http://metrics.sourceforge.net/
[48] There are 155 inner classes ; 155+1025=1180 as indicated in Table 1.

Case study	Test case	Classes	LOC	NMC_{SUS}	Max DIT	Question(s)
OE	OE_TC1	4	40	1,000	2	1
	OE_TC2			4,000,000	2	1,3
PCC Prover	PCC Prover	8	1,280	1,138	1	1
Calculator	Calculator	16	1,175	113	3	1
Weka	Weka_TestSuite	1,180	238,556	4,497,261	5	1
	Weka_TC4			3,993,699	5	1, 3
Producer-Consumer	TC1	9	237	50,002	3	2
	TC2			225,002	3	2
	TC3			500,002	3	2
	TC4			5,000,002	3	2

Table 1 Characteristics of the five case study systems

The last column in Table 1 indicates which case study system is used to answer which research question.

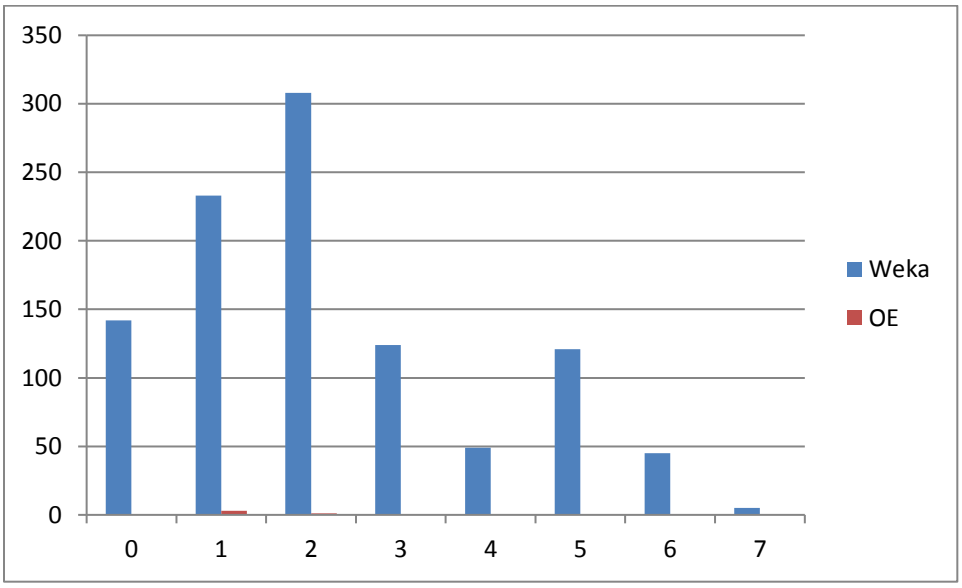


Fig. 15 Depth of Inheritance Tree for Weka and OE

4.3 Experiment set up

We describe the experiment set up in terms of the test cases we executed on the case studies in order to generate traces (Section 4.3.1), the specific setup of Leduc's

approach to allow comparisons (Section 4.3.2), and the execution platform we used (Section 4.3.3).

4.3.1 Executions (test cases)

To answer RQs and quantify overhead, each case study necessarily needs executions, i.e., test cases. We used a set of test cases for each case study during this chapter and the next one. Depending on the experiment, we use different inputs for OE to simulate a large or small software system (200, 10^2, 10^3, 10^7, etc.). We primarily used inputs 200, which we refer to as OE_TC1, and $8*10^5$, which we refer to as OE_TC2, in this chapter and the next one. We selected this specific input for OE in test case OE_TC2 because, as indicated in Table 1, it results in a similar number of method calls as what we obtain when executing Weka with test Weka_TC4, thereby allowing us to perform some comparisons between the two experiments. Calculator uses a simple one plus one test case (1+1), while PCC Prover uses a set of user rules to prove addition operation between sentential forms of two decimal digits [201]. For Weka, we collected test cases from the Weka user manual [204] and executed them, by-passing the GUI (Weka can be executed from the command line): use of *ZeroR* and *J48* classifier models to map from all-but-one dataset attributes to the class attribute (two examples); use of the *meta-classifiers* command to apply multiple classifiers' models in parallel for a dataset (stacking) (one example); supervised filtering of datasets based on class information to discretize numeric attributes into nominal attributes and create a stratified subsample of a given dataset (resample) (two examples); and list information for the specific package from the Weka server, which was locally cached as the network was off at the time of experiment (one example). More details about these test cases are available in Appendix C. We used

these test cases in two situations. We created a test suite made of all those test cases executed in sequence, which we refer to as Weka_TestSuite (Table 1); we studied the longest, in terms of execution time, test case separately (Weka_TC4 in Table 1).

For Producer-Consumer, we tried different test cases and initial settings to observe changes in the rate of production and consumption when the SUS was instrumented. Among those cases, we chose four of them to conduct the experiment. They all vary in terms of OE inputs: TC1 with input 10^4, TC2 with input $4.5*10^4$, TC3 with input 10^5, and TC4 with input 10^6.

4.3.2 Setup of Leduc's approach

Leduc's approach can reverse engineer object interactions when those objects execute in different threads or on different nodes on the network, assuming the Java Remote Method Invocation (RMI) mechanisms are used. To do this, the approach not only assigns a unique identifier to objects but it also assigns a unique identifier to threads. It also assigns unique identifiers to nodes in the network and collects timestamps to order logs collected on different nodes in the network. In other words, Leduc's approach collects more information than Kolbah's, and this may have a specific overhead cost.

To avoid a bias in favour of the Light (Kolbah) instrumentation, we removed the recording of node and thread IDs and timestamps from the Original (Leduc) instrumentation, which were necessary in the Original technique to trace RMI and thread communications, thereby making the two instrumentation techniques comparable. Otherwise, the Original technique would be collecting more data than the hybrid (Light) one and would, therefore, be put at a disadvantage.

4.3.3 Execution platform

Executions were performed on a Asus laptop with an Intel(R) i7-3610QM (at 2.3 Ghz * 8), 16 GB of memory and 250 GB Solid-State drive (Samsung 840 Series), running Ubuntu 12.04 64x OS, Open JDK 1.6.0_30, Apache Ant 1.8.2, Apache Ivy 2.3.0, Apache Log4J 1.2.17 and AspectJ 1.6.7.

When possible, all other applications and services running on the computer were turned off, and the network was disconnected (except when we use a remote logger in the next chapter).

4.4 Measurement framework

The experiments we conducted to answer RQ1 and RQ3, rely on two types of measurements to collect data: the number of method calls (Section 4.4.1), and the execution time (Section 4.4.2).

4.4.1 Counting method calls

We measure the Number of Method Calls (NMC) for two purposes in this chapter: reporting characteristic of a case study, and evaluating overhead as it is a well-accepted measure in the HPC field [168]. NMC is the number of places (callers) in the SUS (or `instrument` package) where an invocation (or a series of invocations) is made to another method in the SUS (or in the `instrument` package). In other words, we count places where methods (non-static/static methods and constructors) are called.

As we reported earlier, the dedicated AspectJ program, which calculates NMC_{SUS}, intercepts all methods and constructors within the SUS packages to count method calls to the SUS, not to Java libraries (Table 1). NMC_{SUS} is different from NMC since NMC also includes calls due to the instrumentation (NMC_{Ins}): $NMC = NMC_{SUS} + NMC_{Ins}$. To

capture NMC_{SUS}, the aspect in the dedicated AspectJ program uses `call` join points to intercept all method calls in the program: see Appendix B. More precisely, we used a `before():call` join point to count method calls since the AspectJ compiler weaves the "method call counter advice" into all method invocation locations and therefore counts the calls (or a series of calls), that leads to execution of counted method. Every time a method call is intercepted, a counter is incremented to keep track of the number of each type of method call (by type we mean static or non-static methods and constructors) in the dedicated AspectJ program.

In addition, later in this chapter (Table 5 in Section 4.9.1), we will precisely report on the number of method calls due to the `instrument` package, which we refer to as NMC_{Ins}. We slightly altered the aspects of the Light instrumentation so that, in addition to reporting on object interactions in the SUS, they report on calls that take place within the instrumentation package, i.e., the aspects and helper classes. The information we collect for each call includes: the call type (static/non-static/constructor), the caller's name and location, and the callee's name. This heavier (from an execution time point of view) version of the Light instrumentation has the same instrumentation behavior as the Light instrumentation (measurement is accurate) and is only used for measuring NMC_{Ins}; it is not used to measure execution time.

4.4.2 Measuring time

In addition, we measured execution time. The execution time indicates the time spent executing the (instrumented) SUS. Any instrumentation directly adds to the execution time of a SUS since it requires additional computations. We relied on three versions of OE, PCC Prover, Calculator, Weka and two versions of Producer-Consumer:

one with no instrumentation (referred to as the *Base* version), one with Kolbah's hybrid instrumentation technique [151] (the *Light* version), and one with Leduc's dynamic instrumentation technique [141] (the *Original* version).

The execution time is measured in two ways in our experiments. First, we measured the time difference between the start and end of the program execution using the `time` command in Linux. It is an open source C program (under GNU license[49]) that calculates time based on the kernel information (the kernel keeps track of CPU usage statistics for all processes). We also completed this measure by calling the Java `currentTimeInMillis()` method at the start and end of the SUS and computed the difference.

In addition, we deemed our number of executions (100 executions of each test case) sufficient to average out unexpected behaviours: standard deviations in our data sets account for less than %5 of average execution time.

It is worth noting that unlike the captured time by the Java method, the captured time with the `time` command accounts for the start of the JVM as well as other JVM bookkeeping activities (e.g., garbage collector), and not only the time spent executing the (instrumented) SUS. However, since we compare the execution time for different instrumentations with the same time measurement in our experiments, this should not have any adverse impact on our conclusions. In addition, we deemed our number of executions (typically 100 executions of each test case) sufficient to average out such unexpected behaviours. A sufficient number of executions allows us to examine the validity of a data set through different statistical test, which will be reported in our

[49] http://www.gnu.org/software/time/

experiments (e.g., standard deviations are less than %5 of average execution times in our data sets). Last, as shown and discussed later in this document, we did not observe many outliers, suggesting that the environment of execution was reasonably stable, therefore allowing us to compare data, even if time values include JVM activities.

4.5 Criteria to select case study

In this section, we present a set of criteria to distinguish between case studies and select proper ones based on each experiment's requirements. We borrow these criteria from the benchmark engineering field [205]. We suggest six criteria (denoted as C) to analyze case studies, a case study may/may not satisfy these criteria.

1. C1: Representative / Relevant: the case study should be representative enough that experiment results can be extended to a real system. Therefore, a case study needs to: a) select a representative workload to simulate the intended real system, b) use relevant software features with respect to the intended system. For example, to test a supermarket software based on the message oriented middleware, the same kind of messages should be simulated under the message oriented middleware.
2. C2: Repeatable: the case study should be repeatable; i.e., different executions of the same case study should result in the same results. Therefore, the case study and its workload should: a) produce deterministic results, b) be designed for repeatability (e.g., a case study should execute in different environments).
3. C3: Robust: the case study results should not be influenced by factors outside the control of the case study. Therefore, the case study needs a) deterministic results b) robust measurements to resist its own perturbation (e.g., always report on a correct execution time).

4. C4: Simple: a case study should be defined in a simple and clear way. That is, it should be easy to configure and execute, it should be easy to understand what is exactly measured and how to interpret results. Therefore, simple measures would result in a better understanding in an experiment.
5. C5: Scalable: the workload (i.e., test cases in our context) of the case study can be increased or decreased to allow performance evaluation on wide ranges of target systems.
6. C6: Comprehensive: beside being relevant, a case study and its workload need to cover all important parts of intended system and environment.

Now we analyze the case studies we selected based on these criteria. Table 2 shows the list of criteria each case study satisfies. The PCC Prover and Calculator case studies do not satisfy the representativeness criterion. For experiments on overhead, we target large systems containing long sequences of method calls. PCC prover and calculator cannot produce such large traces. In addition, the design of the Producer-Consumer case study, even though it is quite a simple multi-threaded program, is good enough for our purpose to study the probe effect.

Criteria / Case study	C1	C2	C3	C4	C5	C6
OE	X	X	X	X	X	X
PCC Prover		X	X	X		
Calculator		X	X	X		
Weka	X	X	X		X	X
Producer-Consumer	X	X	X	X	X	X

Table 2 Satisfied criteria for each case study

All case studies are repeatable and robust since they produce deterministic results and they are used with well-defined measures. Also, Table 2 shows that, except for Weka, all the other case studies satisfy the simplicity criterion. In the case of Weka, depending on a type of experiment, it is not always easy to configure and execute it. Weka, OE and Producer-Consumer to some extent, can simulate large or small systems based on their test cases. As mentioned before, OE is designed to be very scalable by easily controlling the number of loop iterations through input values. In addition, Weka, OE and Producer-Consumer can illustrate the overhead and probe effects in different conditions. Of course, Weka is more comprehensive than OE and Producer-Consumer.

We use PCC Prover, Calculator, and OE (with short execution) to simulate small systems in overhead study. To simulate large systems, the intended type of system in this work, we use Weka and OE with long executions. Finally, we use Producer-Consumer to study the probe effect in multi-threaded programs.

4.6 Experimental design for overhead study

Our objective was to study the probe effect of the Original and Light instrumentations on execution time in three steps: (1) by looking at the instrumentation impact on limited size software systems; (2) by simulating the instrumentation impact on execution time for larger systems; (3) by looking at the instrumentation impact on a real, large system. We first discuss below the experimental design that is specific to these experiments, above what we already summarized earlier, and then discuss results (Section 4.7).

First, we repeated Kolbah's experiment with three case studies of limited sizes (first objective mentioned earlier): OE_TC1, i.e., OE with command line argument value

200, PCC Prover, and Calculator. For each case study system, we executed each test case 500 times in an attempt to control for the possible impact of the operating system or other aspects of the execution environment on execution time. Kolbah selected input 200 for OE as this leads to a number of method calls similar to the execution of PCC Prover (Table 1) and would therefore allow comparisons. Each time we executed the Base, the Original and the Light instrumentation versions.

In a second step (second objective mentioned earlier), and contrary to what Kolbah did, OE was executed with varying command line argument values to trigger large to very large amounts of method calls and loop executions. Specifically, we executed OE with command line argument values of 10^2, 10^3, 10^4, 10^5, 10^6, and 10^7, 100 times each. (Recall from Section 4.2 that this resulted in as many loop iterations and even more if-then or if-else and method executions.) This necessarily resulted in large traces. Each time we executed the Base, the Original and the Light instrumentation versions.

In a third step, as opposed to simulating long executions and, therefore, large traces, we actually used a much larger system: Weka. Each test case was executed 100 times. Each time we executed the Base, the Original and the Light instrumentation versions.

Before discussing *results*, recall that we expect variations from case study system to case study system because of differences in instrumentation techniques (e.g., Kolbah used a less demanding before advice than the around advice of Leduc) and the characteristics of the case study systems (e.g., amount of loops or if statements, amounts of calls to monitor). For instance, if we only consider the number of calls to the logger, which is one of the main sources of overhead, the differences between Light and Original

are due to the fact that for the Light instrumentation, the number of calls to the logger is the sum of the number of method calls and twice the number of constructor calls in the SUS whereas for the Original instrumentation, the number of calls to the logger is the sum of twice (because of the around advice) the number of method and constructor calls and twice the number of conditions and loops encountered in the SUS.

Execution times are calculated based on the Unix `time` command. (The other time measurement led to similar conclusions and we therefore do not report those results.) We use box plots to show all observed execution time values (all in seconds) along the y-axis, including the minimal and maximal ones, as well as first and third quartiles encompassing a time range achieved by half of the total executions (recall that each instrumentation version was executed 500 times in the first step and 100 times in the second and third steps). Figures also provide tables with mean execution time values of the different instrumentations.

4.7 Results of overhead study

We discuss results for the three objectives in sequence as they correspond to the three experimental steps we discussed earlier: the impact of overhead on limited size programs (Section 4.7.1), the trend of overhead as the size of SUS grows (Section 4.7.2), and the impact of overhead on an industrial size system (Section 4.7.3).

4.7.1 Overhead impact on limited size software

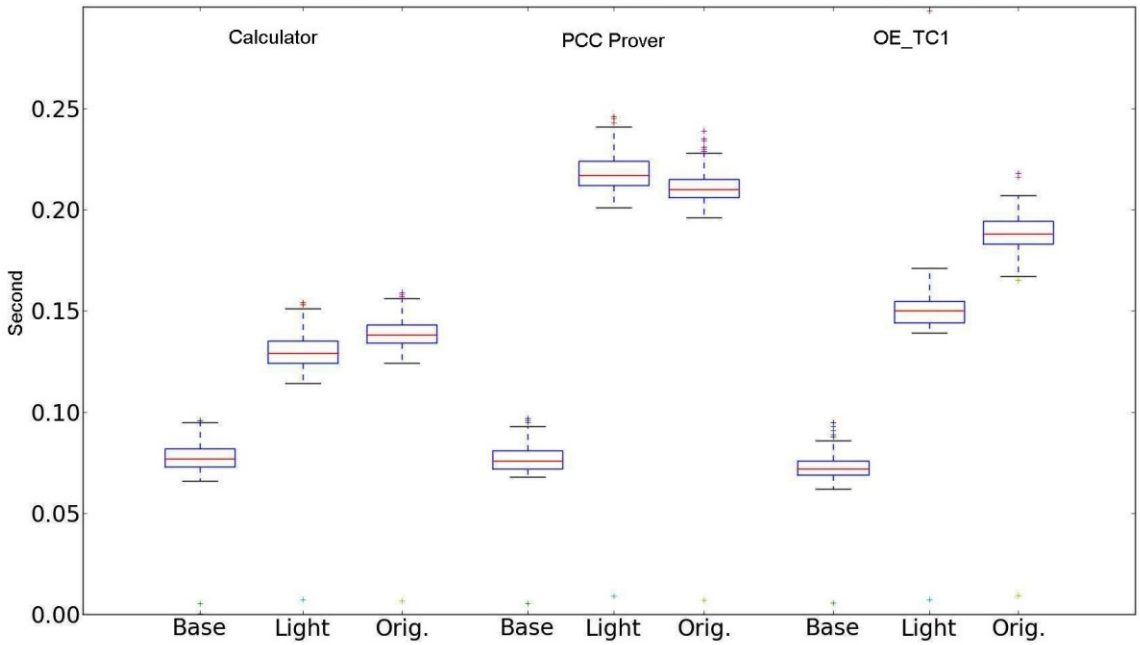
Fig. 16 shows the mean execution times for the three simple case studies (Calculator, OE and PCC Prover) for the three instrumentation strategies: no instrumentation (Base), Leduc's instrumentation (Original), Kolbah's instrumentation (Light). It also shows the corresponding box plots of execution times.

Comparing the differences between minimum and maximum execution times, most execution times lie within a narrow range (so we can discard the outliers). The largest number of points for each variation is found near the average execution time. This indicates stable operating environments (e.g., very few other processes running on the computer). The higher points probably occurred during times that the processor was handling other expensive system events we were not able to control. On average, the Light instrumentation causes the program to execute slower than it would without instrumentation (Base) but faster than or equal to the original instrumentation (Orig), especially as the number of method calls grows, i.e., the size of the instrumented program grows: OE or PCC versus Calculator.

We compared the samples of 500 execution times obtained with the Light and Orig versions and with the Orig and Base versions, for all three systems: we used a one-tailed t-test and confirmed the results with the corresponding non-parametric Wilcoxon signed-rank test. All comparisons are statistically significant (p-value threshold at 0.05), with all the p-values smaller than 0.0001. The Light instrumentation statistically leads to a smaller overhead than the Original instrumentation.

We investigated reasons for differences in execution times between the Light and the Original instrumentations. Regardless of the instrumentation strategy, the instrumentation makes system calls to write to a log file. A file needs to be created and written to repeatedly, which are execution-heavy tasks (heavier than any instrumentation-related behaviour added to the programs). In Chapter 5, we will investigate logging mechanisms other than (repeatedly) writing to a file. Also, we noticed the number of characters written into a file for Light is higher than for Original. For example, the first

trace entry for Calculator Light was 249 characters long, versus 175 for Original. The Light instrumentation is, therefore, not entirely "lighter" than the Original instrumentation for executions small enough not to be negatively affected by large numbers of instrumentation calls. Indeed, even though the Original instrumentation writes to a file at least twice as often as the Light instrumentation, the number of times the file is written to is small so the difference is not important.



	Calculator			PCC Prover			OE_TC1		
	Base	Light	Original	Base	Light	Original	Base	Light	Original
Mean time (s)	0.077	0.129	0.138	0.076	0.218	0.210	0.072	0.197	0.214

Fig. 16 Execution times for different instrumentations of Calculator, PCC Prover and OE_TC1

In addition, the Original instrumentation collects more control flow information for OE_TC1 compared to Calculator and PCC Prover. We suspect that the overhead when the program is small mostly comes from the creation of the trace file, running the

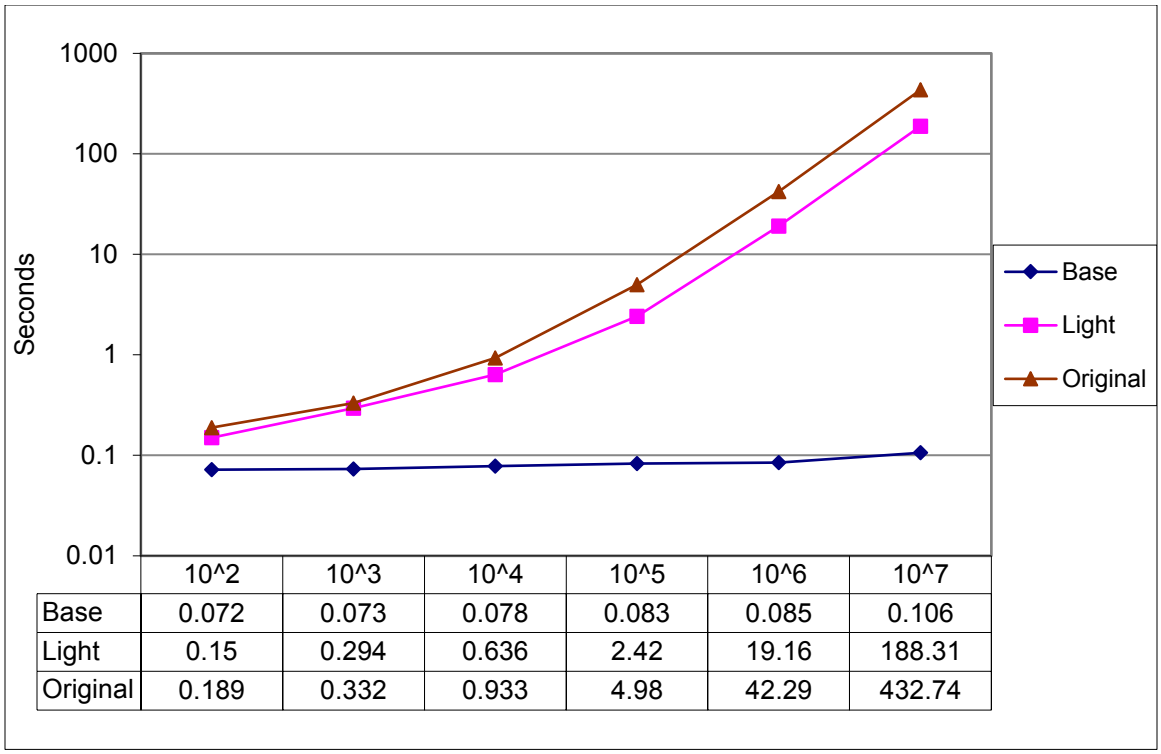
JVM and other housekeeping activities. We will further investigate the difference between the Light and Original instrumentations for large traces in the third step.

4.7.2 Simulating overhead impact on a large software system

The second step of the investigation consisted in simulating the instrumentation impact on execution time for larger systems, and systems of increasing sizes. We executed OE with command line argument values 10^2, 10^3, 10^4, 10^5, 10^6, and 10^7, 100 times each.

Fig. 17 shows the results, using a logarithmic scale on the y-axis. We only show the median execution time over the 100 executions: the standard deviation was very small. Again, the Light instrumentation causes the program to execute slower than it would without instrumentation (Base), but much faster than with the Original instrumentation: the Light instrumentation is two times faster than the Original one for 10^5, 10^6 and 10^7, %33 times faster for 10^4.

Fig. 17 also shows that additional work is required to further reduce the impact of instrumentation since the Light instrumentation is much slower than the Base execution. We believe a different tracing mechanism, with fewer accesses to the disk, to save smaller amounts of data, should be used. One may also consider instrumenting only parts of a large program to reverse engineer.

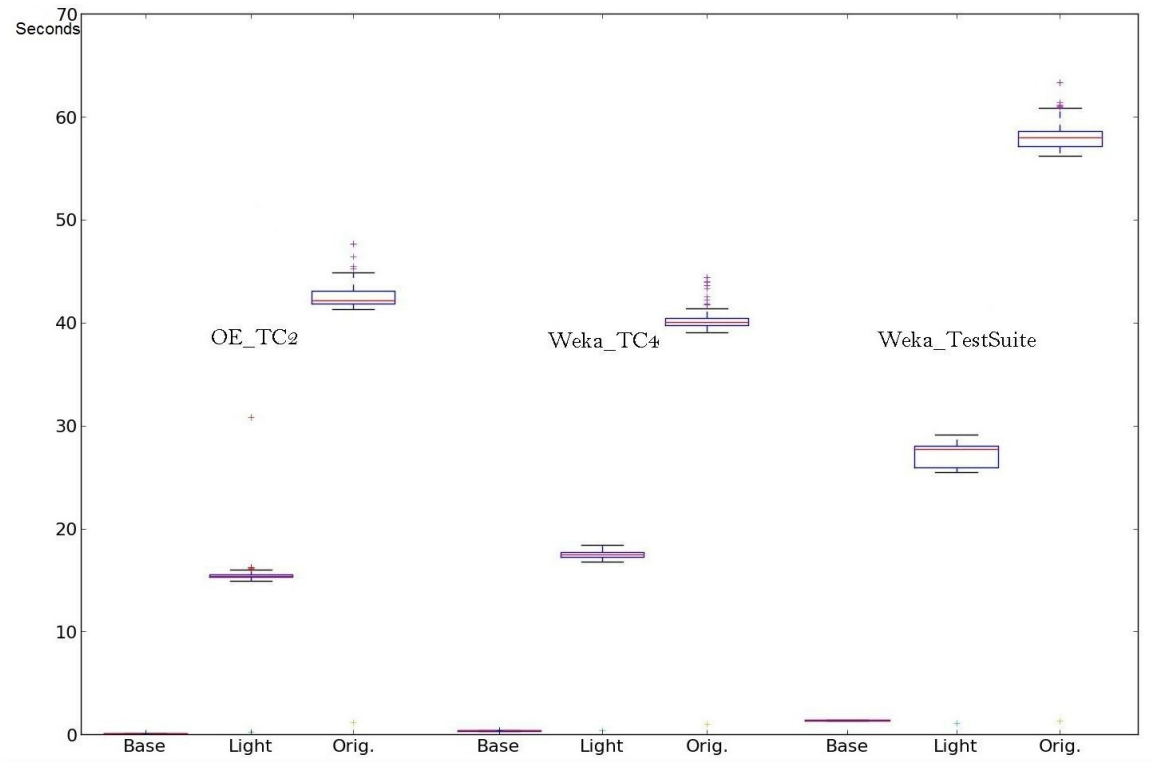


	10^2	10^3	10^4	10^5	10^6	10^7
Base	0.072	0.073	0.078	0.083	0.085	0.106
Light	0.15	0.294	0.636	2.42	19.16	188.31
Original	0.189	0.332	0.933	4.98	42.29	432.74

Fig. 17 Mean execution times (in seconds) as the number of method calls increases

4.7.3 Overhead impact on industry sized software system

In the third step, we studied the impact on execution time with Weka: Fig. 18. Once again, boxplots are the results of 100 executions. We observe that the Original instrumentation is 2 (resp. 2.3) times slower than the Light instrumentation for the whole test suite (resp., TC4), and the Light one is much slower than the Base one, confirming there is room for improvements, that is, probe effect reduction. Note that we used a one tailed t-test by comparing Light and Original, Base and Original, Base and Light for all three systems (OE_TC2, Weka_TC4, Weka_TestSuite). The results show all comparisons are statistically significant with a P-value approaching to zero (e.g., 6.72E-177).



	OE_TC2			Weka_TC4			Weka_TestSuite		
	Base	Light	Original	Base	Light	Original	Base	Light	Original
Mean time (s)	0.08	15.41	42.14	0.37	17.50	40.22	1.38	27.70	57.95
Size (MB)	N/A	989.6	3132	N/A	1250	3085	N/A	1530.7	3895

Fig. 18 Execution times (in seconds) for Weka instrumentation (100 executions)

Recall, from Table 1 (and Section 4.3.1), that executing OE_TC2, i.e., OE with input 800,000, triggers a very similar number of calls as Weka_TC4. The very similar size of traces of these case studies for Light and Original instrumentations shows the validity of our comparison. Notice also that the Light instrumentation between OE_TC2 and Weka_TC4 leads to similar execution times, indicating that OE is a reasonable mechanism to simulate large/long executions. In addition, comparing the Base to Light ratio (i.e., Base/Light) for OE_TC2 and Weka_TC4 shows lower ratio for OE_TC2 (%0.5 vs. %2.1 resp.). This confirms our hypothesis (mentioned in Section 4.2) that OE behaves

worse than a real system with a similar number of method calls (Weka_TC4 in this case) in terms of adding overhead in the Light instrumentation.

4.7.4 Conclusion

Overall, these results confirm quantitatively, beyond the initial intuition, that removing instrumentation of control flow structures significantly helped reduce the probe effect by at least halving the execution time, and that Kolbah's aspects (before rather than around advice) also helped. They also indicate that much may remain to be done to reduce overhead of the Light instrumentation. In addition, our results (Appendix D) show that Kolbah's approach is platform independent and it is possible to extend her hybrid approach to other languages. We extended the dynamic analysis part of her hybrid approach to C++ by adapting the trace model based on C++ language characteristics, developing the instrumentation tool based AspectC++, and verifying the correctness of reverse engineered traces by comparing to their Java equivalents.

We used a different execution platform than the one we originally used when reporting those results in a conference [151]. Aside from differences in hardware configuration (CPU at 2.66Ghz * 4 and hard-disk drive in our paper [151], compared to a CPU at 2.3Ghz * 8 and solid-state drive in this thesis) which can explain the faster executions we report here compared to our paper (especially the use of a solid-state drive), the Linux OS in general performs IO operations faster than Windows. The caching mechanism that Linux applies when writing on disk (Page Cache) makes Linux more effective than Windows in write operation [206].

4.8 Probe effect on functional behaviour

We showed in Section 4.7 that the hybrid (Light) approach indeed reduces the overhead compared to the Original approach, though there is significant overhead compared to no instrumentation. In this section, our objective is to compare the behaviour of the Light version versus the Base version of the Producer-Consumer. If the difference in behaviour between the two instrumentation versions indicates the presence of a probe effect, we need to further reduce the overhead. Later in Chapter 5, we compare the Light instrumentation with an Optimized Light instrumentation (i.e., the instrumentation with improved performance) to understand to what extent optimized instrumentation is successful at reducing the probe effect for our case studies.

4.8.1 Experiment design

To answer RQ2 (probe effect observation), we designed four test cases for the Producer-Consumer case study we mentioned earlier (Section 4.2). Except for the input to OE, which simulates different delays in the consumer, we keep all the other settings for test cases the same: the specified time in the producer thread to suspend execution is one second. A test case is considered a failure in this experiment if the instrumented Producer-Consumer losses data (the producer cannot produce); it is a success otherwise. Recall a standard producer-consumer works well if the consumption rate is greater or equal to the production rate, but produced items will be missed if the consumption rate is smaller than the production rate. The test cases we used confirm that production is slower or equal to consumption in the non-instrumented version: execution does not take more than one second in the consumer. Therefore, all test cases should pass. The execution time was measured using calls to the Java `System.currentTimeMillis()` method at the

start and end of the consumer to compute the delay. We ran two versions of the Producer-Consumer: the non-instrumented (Base) one and the Light one. We did not use the Original version because we already showed it is slower than Light, i.e., it introduces more execution overhead. We used the Base and Light versions for each test case 100 times. Note that the execution cycle in the Producer-Consumer starts when the producer generates an item and completes when the consumer consumes the first generated item.

4.8.2 Results

Table 3 shows actual delays due to instrumentation and a coarse grained comparison of relative executions between the consumer and the producer: "Greater" (resp. "Smaller") means that the consumption rate is greater (resp. smaller) than the production rate, i.e., the producer is slower (resp. faster) than the consumer, "Equal" means the two rates are approximately the same (both producer and consumer work at approximately the same pace). The delays in the Base and Light columns indicate the average (over 100 executions) time for the consumer to consume an object.

The producer is negligibly affected by the instrumentation as it executes only one method call (to the queue) in case the queue is not full and therefore only one call is instrumented: execution times of the producer are therefore not reported.

For Base (Table 3), as we expected, the consumer is faster than the producer in each test case. However, in Light, the consumer is slower than the producer for TC3 and TC4, i.e., faulty test cases whereby a `QueueFull` exception was observed. This is evidence that the instrumentation changes the behaviour of the SUS: the producer cannot produce objects.

This observation indicates that, despite major overhead reduction in the Light instrumentation (of the hybrid approach), compared to the Original instrumentation (recall previous sections), the probe effect due to instrumentation can change software behaviour. Therefore, more optimization is necessary to reduce the probe effect in the hybrid approach.

	Base		Light	
	Rate	Consumer delay	Rate	Consumer delay
TC1	Greater	$1.5*10^{-5}$	Greater	0.229
TC2	Greater	$4*10^{-5}$	Equal	1.008
TC3	Greater	$8.5*10^{-5}$	Smaller	2.27
TC4	Greater	$77*10^{-5}$	Smaller	23.18

Table 3 Instrumentation effect on the Producer-Consumer (Base, Light)

4.9 Identifying potential improvement areas in Light

The previous sections showed that, although the Light instrumentation improves over the Original instrumentation, there is likely room for improvements in terms of overhead reduction. In this section, we try to better understand, by means of yet other experiments, how each instrumentation component, which we define later, of the Light approach contributes to overhead. The rationale is to first identify the extent of a problem prior to studying possible optimization alternatives. In other words, if experiments show that one component of the instrumentation does not contribute much to overhead, then there is no real incentive to search for an alternative, better solution to implement this component. We already suspect that accesses to the disk to save log items may contribute significantly to overhead. However, we do not know to what extent this characteristic of the instrumentation contributes to overhead and whether other characteristics of the instrumentation may contribute more.

In this section, we study overhead based on two measures: the number of method calls and time. We first look at the number of method calls that happen when a piece of software is not instrumented (i.e., NMC_{SUS}) and the number of method calls due to the `instrument` package when that piece of software is instrumented with the Light approach (i.e., NMC_{Ins}): Section 4.9.1. We then study the contribution of different components of the Light instrumentation to overhead based on execution time (Section 4.9.2).

With experiments reported in Sections 4.7 and 4.8, we showed the NMC_{SUS} is an important characteristic of a SUS since the Light instrumentation intercepts method calls of the SUS, thereby causing overhead by collecting caller's and callee's information. We consider the NMC_{Ins} as a measure to assess the cost of instrumentation in this section. For a fixed number of NMC_{SUS}, the higher NMC_{Ins} the higher the chances of a higher cost of instrumentation. In addition, a call to another method, per se, consumes resources, for instance by loading and storing data (e.g., arguments, call instruction) in memory [168]. Therefore, programming Light instrumentation in a way to avoid unnecessary method calls can potentially improve the performance. It is notable however that reducing the NMC_{Ins} may not result in better performance in the Light instrumentation because the AspectJ compiler may already have optimized the Light instrumentation during compilation. Depending on a compiler and the mathematical models it uses for optimization, an inter-procedural call from one basic block to another can be optimized at compile time, whereby a target basic block is hard coded into source basic block to remove the call overhead. Even though the AspectJ compiler may replace some of the method calls with actual code to form fewer basic blocks at compile time, we do not

know for which type of calls AspectJ does this optimization. In addition, this kind of optimization generally requires a user to compile using a specific mode (e.g., `-O`) in AspectJ. We avoided doing so to not hinder the usability of our tool. Overall, we think the number of method calls is a good measure to show the cost of instrumentation and where to reduce calls during the optimization process. This overhead measure is focused on traits of instrumentation due to programming and computations, not the overhead due to I/O operations.

Timing different parts of a program through different techniques (e.g., calling a `timer` function directly or using performance tuning profilers, both at different levels of granularity) is a well established approach to measure and improve a program's performance [168].

4.9.1 Number of method calls

In this experiment, we are concerned with NMC_{Ins}, that is method calls originating from the `instrument` package. We look into different kinds of method calls that contribute to NMC_{Ins} in this experiment.

4.9.1.1 Experiment design

We want to identify the number of method calls due to different components of the Light instrumentation, namely: the overall number of method calls in all components in the Light instrumentation (i.e., NMC_{Ins}); the number of method calls due to advices and log preparation and log storage within inter-type declaration, though not accounting for object identification (which we refer to as NMC_{Advice}); the number of method calls due to object identification (which we refer to as $NMC_{ObjectInter}$); the number of method calls due to the Logger (which we refer to as NMC_{Logger}). We divide NMC_{Advice} into three

different numbers based on the call type: $NMC_{AdviceLogger}$, $NMC_{AdviceInfo}$, and $NMC_{AdviceObject}$. The $NMC_{AdviceLogger}$, $NMC_{AdviceObject}$ and $NMC_{AdviceInfo}$ show the number of method calls from the aspect advice: to send captured object information to the logger, to aspect methods to capture object identification information and to aspect methods to capture other required object information respectively. Similarly, we split the $NMC_{ObjectInter}$ into two NMCs based on the call type: $NMC_{ObjectIDgen}$ and $NMC_{ObjectIDCheck}$. The $NMC_{ObjectIDgen}$ and $NMC_{ObjectIDCheck}$ indicate the number of object identifiers that have been initialized and the number of times aspect methods check whether object identifiers have been initialized.

Looking at the `instrument` package shows that it initiates calls to three different destinations. First there are calls from the aspect advice (NMC_{Advice}) to three other parts: the `Logger`, e.g., method `instrument()` ($NMC_{AdviceLogger}$), the methods of the `MethodAspect` aspect that capture object information, e.g., method `getMethodSignature()` ($NMC_{AdviceInfo}$), and other parts of the advice that inquire for the value of `objectID`, e.g., method `getObjectID()` ($NMC_{AdviceObject}$). Second, there are calls by the `Logger` to itself, e.g., method `getFileWriter()` (NMC_{Logger}). Third, there are calls to object identification within the `IdentifierAspect` itself, e.g., method `objectIDgenerator()` ($NMC_{ObjectInter}$): recall Fig. 11. More precisely, $NMC_{ObjectInter}$ includes calls that are originated either from `getObjectID()` in the `ObjectID` interface (due to advice inquiry for object information), from the initialization of the inter-typed class variable `objectID` (due to the use of the singleton design pattern) to the `objectIDgenerator()` method ($NMC_{ObjectIDgen}$), or from `getObjectID()` in the

`ObjectID` interface to the `objectIDgenerator()` method ($NMC_{ObjectIDCheck}$). It is worth noting that the Light instrumentation does not make any method call to the SUS.

We have already discussed in Section 4.4.1 how a modified version of the Light instrumentation measures NMC_{Ins}. An execution of the SUS with the modified (to collect NMC) Light instrumentation produces a file that contains information about all method calls happening in the `instrument` package. The information is printed out in the form of statements, each of which includes: call type, i.e. static/non-static/constructor, caller's name/location, and callee's name. Counting all statements in the file gives us NMC_{Ins}.

Thanks to the information collected, we count these statements in different ways to understand different NMCs. First, we compute the NMC_{Advice} by counting: all statements that are initiated from the `MethodAspect` (the caller's location), and only those statements that are initiated from the `IdentifierAspect` (the caller's location) to call to the `Logger` (the callee's name). We calculate each subset of NMC_{Advice} by counting: (i) statements for which the caller's location and the callee's name correspond to `MethodAspect` and `getObjectID()` respectively ($NMC_{AdviceObject}$), (ii) statements for which the caller's location is `MethodAspect` and the callee's name is one of the helper methods that prepares object information in `MethodAspect` ($NMC_{AdviceInfo}$), and (iii) aspect calls specifically due to the `Logger`, that is statements that include `MethodAspect` and `IdentifierAspect` as the caller's location and `Logger`'s methods (i.e., `getLoggingClient()` and `instrument()`) as the callee's name ($NMC_{AdviceLogger}$). Second, we compute NMC_{Logger} by counting every statement for which the caller's location is the `Logger`. Third, we calculate $NMC_{ObjectInter}$ by counting statements with

static method `objectIDgenerator()` as a callee's name and the `IdentifierAspect` as a caller's location.

To calculate $NMC_{ObjectIDgen}$, we ran another experiment in which we removed the body of all advices in the modified Light instrumentation we used above. This way advices make no call to the `getObjectID()` method, and the `objectIDgenerator()` method is implicitly called (by the inter-typed object attribute), the logger prints object identifiers right after their execution, and the `if` statement in line 5 in Fig. 11 always returns true. Therefore, $NMC_{ObjectIDgen}$ indicates the number of object identifiers that have been initialized. Based on this number, we can obtain the $NMC_{ObjectIDCheck}$ by calculating the difference between $NMC_{ObjectIDInter}$ and $NMC_{ObjectIDgen}$. Recall from Section 3.2.1.2 that $NMC_{ObjectIDgen}$, which is equal to the number of log items after object initialization, would likely be greater than the actual number of object instances in case of inheritance.

In summary, we directly measured NMC_{Ins}, $NMC_{ObjectInter}$ (including $NMC_{ObjectIDgen}$ and $NMC_{ObjectIDCheck}$), NMC_{Advice} (including $NMC_{AdviceObject}$, $NMC_{AdviceInfo}$, and $NMC_{AdviceLogger}$), and NMC_{Logger}. Table 4 summarizes the elements of the Light instrumentation that are activated in the three experiments we just described.

		NMC_{Ins}	$NMC_{ObjectInter}$		NMC_{Advice}			NMC_{Logger}
			$NMC_{ObjectIDgen}$	$NMC_{ObjectIDCheck}$	$NMC_{AdviceObject}$	$NMC_{AdviceInfo}$	$NMC_{AdviceLogger}$	
Object identification		✠	✠					
		✠		✠				
Advices	ObjectID	✠			✠			
	ObjectInfo	✠				✠		
	CalltoLogger	✠					✠	
Logger		✠						✠

Table 4 Experiment to measure NMC in different components of Light

We picked 800,000 as the input for OE in order to have a similar NMC_{SUS} to Weka_TC4's. This can potentially give us a better understanding of optimization areas in the Light instrumentation, because two case studies of similar trace size with different characteristics (such as depth of inheritance hierarchy) can potentially expose different contributions to overhead of various elements of the Light instrumentation. It is worth noting that we use the same versions of Java and AspectJ with the same configurations for all experiments, therefore differences among different types of NMC_{Ins}s are unlikely due to neither Java nor AspectJ compilers' optimization.

4.9.1.2 Results

In Table 5, we repeat the information we showed in Table 4 about the number of method calls in the non-instrumented SUS as a basis for comparison (NMC_{SUS} row). We show NMC_{SUS} with more details by distinguishing between calls to constructor and calls to other methods. We also look at contributions to NMC_{Ins} of different types of method call: Fig. 19. Results are consistent between OE_TC2 and Weka_TC4.

The NMC_{Ins} row suggests that the Light instrumentation imposes a large additional number of method calls to the original SUS (eight times more method calls than the original SUS): 36 million as opposed to 4 million. Therefore, it is fair to say that the Light instrumentation makes on average eight method calls to reverse engineer the runtime information of one call in the SUS.

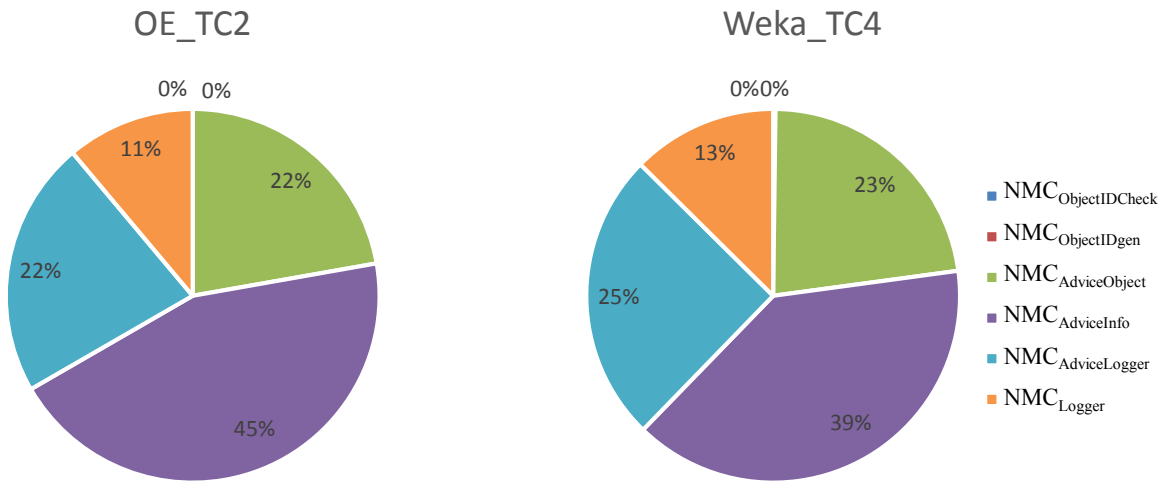


Fig. 19 Number of method calls in aspect

For the different contributions to NMC_{Ins} we provide the raw values of different types of NMC_{Ins}s as well as their percentage contribution to NMC_{Ins}. The NMC_{Advice} shows the largest percentage among all components of the Light instrumentation. This was expected since advices are responsible for collecting method call's information. Therefore optimizing advices to make fewer method calls can potentially reduce the overhead. The NMC_{Logger} accounts for nearly one tenth of the number of method call in Light (NMC_{Ins}), which is an indication to low overhead computation of the logging mechanism. However, as we mentioned earlier, this low overhead computation does not account for actual I/O operations, which may be more expensive and needs to be measured in a different way.

In addition, as we mentioned in the experiment design, $NMC_{ObjectIDgen}$ is equivalent to the number of log items after object initialization. The difference between $NMC_{SUSCons}$ and $NMC_{ObjectIDgen}$ indicates the impact of the cumulative depth of inheritance tree (DIT) on NMC for case studies. This difference is very marginal

compared to NMC_{Ins} and therefore the overhead with respect to DIT is not significant in this object identification mechanism. Recall that the Max DIT for Weka_TC4 is 5 whereas for OE_TC2 it is 2 (Table 1).

Case Study / Type of calls		**OE_TC2**	**Weka_TC4**
NMC_{SUS}	$NMC_{SUSMethod}$	3,999,997	3,933,268
	$NMC_{SUSCons}$	3	60,431
NMC_{Ins}		36,000,018 (%100)	32,270,962 (%100)
$NMC_{ObjectInter}$	$NMC_{ObjectIDCheck}$	1	114
	$NMC_{ObjectIDgen}$	4 (%0)	65,206 (%0.2)
NMC_{Advice}	$NMC_{AdviceObject}$	7,999,997 (%22)	7,311,923 (%22.6)
	$NMC_{AdviceInfo}$	16,000,003 (%44)	12,717,003 (%39.4)
	$NMC_{AdviceLogger}$	8,000,008 (%22)	8,117,810 (%25)
NMC_{Logger}		4,000,005 (%11)	4,058,906 (%12.5)

Table 5 Number of method calls in Light approach in program and in aspect

Although comparing NMC_{SUS} to NMC_{Ins} shows a large difference, any kind of optimization in the Light instrumentation cannot prevent additional method calls being made (recall overhead minimization approach in Section 2.1.4). Indeed, the Light instrumentation requires additional calls to intercept all methods and constructors to capture runtime information. Thus, we need to find ways to minimize NMC_{Ins} when we optimize the Light instrumentation, such as: not intercepting all method calls in a SUS or using method in-lining in the Light instrumentation to reduce NMC_{Ins}.

4.9.2 Execution overhead of each Light component

The objective is now to identify the contribution of several components of the Light instrumentation to execution time (overhead), namely, the contribution to overhead of: AspectJ interception mechanisms, that is aspects without advice code (i.e., empty code) and without object identification mechanism and therefore without logging (which we refer to as $Time_{AspectJ}$); Object identification mechanism only, i.e., inter-typed methods in the `IdentifierAspect`, but not accounting for the lines that prepare and log with the `objectID` information (which we refer to as $Time_{ObjectInter}$); Aspect advices accounting for log preparation in inter-typed methods though without the actual logging, i.e., the code capturing information to be recorded in trace statements (which we refer to as $Time_{Advice}$); Logging mechanism though not accounting for the fact that the logger writes to the disk (which we refer to as $Time_{Logger}$); Writing data to the disk (which we refer to as $Time_{Disk}$). $Time_{Advice}$ includes $Time_{AdviceObject}$ and $Time_{AdviceContext}$, which therefore indicates the overhead due to inquiring for the object information in advices and the overhead due to collecting and preparing the rest of the information.

It is worth noting that, for this experiment, we used the total times to execute the SUS with no instrumentation (Base), which is referred to as $Time_{Base}$ here, and Light instrumentation (Light), which is referred to as $Time_{Light}$ here, and both time values were already reported on in Section 4.6, as a basis for comparison when studying overhead.

4.9.2.1 Experiment design

Except for $Time_{AspectJ}$, which we measured directly, we did not measure $Time_{ObjectInter}$, $Time_{Advice}$ (including $Time_{AdviceObject}$ and $Time_{AdviceContext}$), $Time_{Logger}$ and $Time_{Disk}$ directly. A direct measure of the latter four could be done by inserting calls to

`System.currentTimeMillis()` at adequate places and printing out the result. This would however introduce additional overhead, though small. Instead, we used other measurements and computed these six overhead values indirectly as discussed next.

In a first experiment, we measured time while commenting out the calls to the Logger, i.e., `Logger.getLoggingClient().instrument(log)`, in all three advices and inter-type declarations in Fig. 6, Fig. 7, Fig. 8 and Fig. 11. This way the aspect code intercepts everything as in the full-fledged Light version, collects and prepares all the required information, but does not send the information to the Logger, and the information is therefore not saved on disk. In a second experiment, we measured time while not only commenting out the calls to the Logger, but also commenting out lines that prepare the object data in aspects (advices and inter-type methods) in Fig. 6, Fig. 7, Fig. 8, and Fig. 11. Again, the aspect code intercepts everything, including obtaining object identifications, but does not make calls to the logger, nor prepare the log information. In another (3^{rd}) experiment, we measured the execution time of the SUS where aspects have empty advices and no object identification, i.e., the aspect code intercepts everything (i.e., calls) as in the full-fledged Light version, but does not collect any information, does not identify any object instance, does not call the logger, which does not save anything. Similarly, in another (4^{th}) experiment, we measured the execution time of the SUS where aspects have empty advices, this time accounting for the object identification code, though lines for preparing and saving log (i.e., call to Logger) were excluded. In yet another (5^{th}) experiment, we only commented out the statements that save information to the disk in the logger (the lines that write to the file in the Logger, inside the `try-catch` blocks of method `instrument()` in Fig. 12). The aspect code

therefore intercepts everything as in the full-fledged Light version, collects and prepares all the required information, sends the information to the logger, which prepares the trace statements to be saved but does not save anything on disk.

We therefore obtain five different execution times, in addition to the execution time of the full-fledged Light version (i.e., $Time_{Light}$): $Light_{CallsToLogger}$, $Light_{AdviceObjectID}$, $Time_{AspectJ}$, $Light_{EmptyAdvice}$, and $Light_{NoDiskSave}$, respectively (to the above discussion). The components of the Light instrumentation that are activated (or not) in these experiments are summarized in Table 6.

		$Time_{Light}$	$Light_{CallsToLogger}$	$Light_{AdviceObjectID}$	$Light_{EmptyAdvice}$	$Time_{AspectJ}$	$Light_{NoDiskSave}$
AspectJ interception mechanisms		✠	✠	✠	✠	✠	✠
Object identification		✠	✠	✠	✠		✠
Advices	Object info	✠	✠	✠			✠
	Other info	✠	✠				✠
Logger		✠					✠
Disk writes		✠					

Table 6 Experiment to measure the overhead of different components of Light

With such measurements, we can determine (compute) $Time_{AspectJ}$, $Time_{Advice}$, $Time_{Logger}$, and $Time_{Disk}$ as follows:

- $Time_{ObjectInter} = Light_{EmptyAdvice} - Time_{AspectJ}$;
- $Time_{Advice} = Light_{CallsToLogger} - Light_{EmptyAdvice}$;
- $Time_{AdviceObject} = Light_{AdviceObjectID} - Light_{EmptyAdvice}$;
- $Time_{Logger} = Light_{NoDiskSave} - Light_{CallsToLogger}$;
- $Time_{Disk} = Time_{Light} - Light_{NoDiskSave}$.

We used OE_TC2 and Weka_TC4 as case studies in these experiments as we needed long executions to have a chance to observe small contributions to overhead: for instance we expected the logger code, except writing to the disk (i.e., $Time_{Logger}$) to have a small contribution to overhead since its code is small. Each time, we triggered 100 executions to average out outliers. The execution time was measured based on the Linux `time` command.

4.9.2.2 Results

Table 7 shows execution time values (in seconds) of $Time_{Base}$ and $Time_{Light}$ as basis of comparison, and $Time_{Advice}$, $Time_{ObjectInter}$, $Time_{AspectJ}$, $Time_{Logger}$, and $Time_{Disk}$, for OE_TC2 and Weka_TC4. In Fig. 20, we show the percentage of each part of $Time_{Light}$ based on mentioned Times for both case studies. The standard deviation values for all execution time samples were equal to or less than 0.1, except for samples of the Light instrumentation with values equal to or less than 0.4.

For both case studies, writing the log information to disk is a source of overhead in the Light instrumentation (last column), which is not entirely surprising. What was not expected was the amount of contribution of the write to disk operation to the overhead in our instrumentation: the largest contributor, with roughly three quarter of the overhead. Therefore, we need to look into alternative logging mechanisms to reduce this overhead. The second highest contributor to overhead is the advice (fourth column): one quarter of the overhead. Although this is much less than writing to the disk, this is still a considerable amount of overhead compared to $Time_{Base}$ (12 to 45 times more than Base).

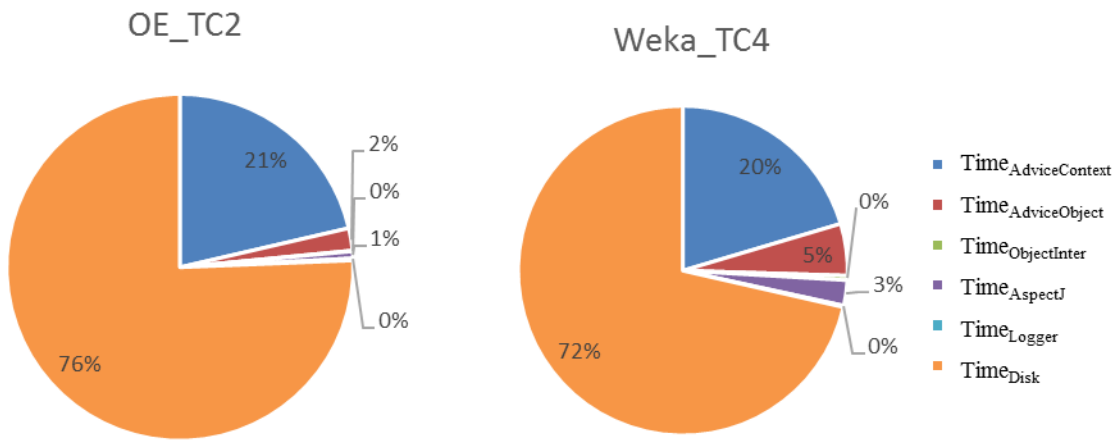


Fig. 20 Percentage of execution time for each part of Light Inst.

While the differences, in terms of proportion in two case studies, was due to different software characteristics (e.g. more object instances in Weka_TC4 and, consequently, more calls in the aspect to handle unique object IDs), data indicate the need to optimize aspect code. Despite one's initial intuition that AspectJ may contribute, possibly significantly, to overhead, $Time_{AspectJ}$ amounts at most to only 3% of total overhead, which suggests the AspectJ interception mechanisms we used are very efficient. The contributions to the overall overhead of object identification and the logger are negligible: columns $Time_{ObjectInter}$ and $Time_{Logger}$. Note that small values for the $Time_{ObjectInter}$ indicate that the inter-typed declarations we used for object identification does not impose much overhead.

Case study	$Time_{Base}$	$Time_{Light}$	$Time_{Advice}$		$Time_{ObjectInter}$	$Time_{AspectJ}$	$Time_{Logger}$	$Time_{Disk}$
			$Time_{AdviceContext}$	$Time_{AdviceObject}$				
OE_TC2	0.08	15.44 (%100)	3.30 (%21.3)	0.322 (%2)	0.011 (%0.07)	0.099 (%0.6)	0.031 (%0.2)	11.68 (%75.6)
Weka_TC4	0.37	17.50 (%100)	3.57 (%20.4)	0.89 (%5)	0.076 (%0.4)	0.435 (%3)	0.025 (%0.1)	12.5 (%71.4)

Table 7 Mean execution times (100 executions) in second for each part of Light Inst.

4.10 Conclusion

We performed several experiments with a number of case studies that indicate: (1) The hybrid approach reduces the probe effect at least by half in large systems, compared to the purely dynamic technique; (2) We need to further reduce overhead in the hybrid approach for an accurate software behaviour observation; (3) The advices and the logging mechanism are the two parts of the Light instrumentation that contribute the most to overhead, an issue we will try to address in the next chapter. We showed that, the risk of inaccurate software behaviour observation is real, especially in multi-threaded software and there is a need to further optimize the hybrid approach.

5 Optimizing the Hybrid Approach

In Chapter 3, we presented Kolbah's hybrid approach that applies both static and dynamic analyses to reduce the probe effect. We also showed in Chapter 4, through experiments, that the hybrid approach leads to less overhead than an equivalent purely dynamic approach (Leduc's). Although Kolbah tried to reduce the instrumentation overhead by analyzing control flow statically, we showed there is still a significant overhead (Section 4.6), that different components of the instrumentation approach contribute differently to overhead (Section 4.9), and that this overhead can result in an inaccurate observation of software behaviour (Section 4.8). Therefore, our research is to focus on different dimensions of instrumentations in general, and the Light instrumentation in particular, to increase the performance (i.e., reduce overhead).

In this chapter, we show how the dynamic analysis of the hybrid approach can be further optimized. To that end, first we present a classification of the characteristics for optimizing the Light instrumentation. This classification is obtained, in part, due to our investigation of overhead for each component of the Light instrumentation in Section 4.9. However, the proposed classification is deemed to be generic enough to be applied to optimizing any hybrid approach. We investigate possible ways of optimizing Kolbah's hybrid approach based on the suggested characteristics and discuss whether each optimization hurts or improves the performance of the Light instrumentation. Once we identified all successful optimizations, we combine them together in a series of instrumentations, the so called "*optimized Light*" instrumentations, and compare them to the Light instrumentation to understand the amount of overhead reduction. Finally, we

also show the effectiveness of the optimized Light instrumentations to prevent unwanted object interactions' behaviour.

In this chapter, first we propose a set of general characteristics to study an instrumentation overhead and a process to optimize the instrumentation (Section 5.1). Next, we ask research questions we intend to investigate during this chapter (Section 5.2). Before starting the optimization, we describe the case studies we use in our experimentation and the way they are configured (Section 5.3). Next, we specialize the suggested optimization process discussd in Section 5.1 for the dynamic analysis part of the hybrid approach (i.e., inter-procedural interactions) based on the suggested characteristics (Section 5.4), specifically by looking into: the mechanism to gather information (Section 5.4.1), the encoding of that information (Section 5.4.2), the mechanism to log information (Section 5.4.3), and the amount and the type of the collected information (Section 5.4.4). Once we obtain all optimizations by combining alternative design decisions investigated in Sections 5.4.1 to 5.4.4, we report on the overall overhead reduction made in an optimized hybrid approach (Section 5.5). In Section 5.6, we compare the hybrid approach and the optimized hybrid approach to understand to what extent our optimizations are effective at reducing the overhead. Finally, we conclude this chapter in Section 5.8.

5.1 Characteristics of the dynamic analysis

We characterize dynamic analysis techniques with respect to their runtime overhead. This characterisation can take place for any dynamic or hybrid analysis technique to systematically identify the cause of overhead and select proper optimization practices to reduce the overhead. Later in Section 5.4, we apply this characterization on

the Light instrumentation and explain a range of optimization practices to optimize the performance.

Any dynamic or hybrid analysis can be evaluated based on four different characteristics when analyzing its instrumentation performance:

1) the collecting of information, i.e., the mechanism by which the information is collected (e.g., use of the AspectJ API);
2) the information being collected, i.e., the amount and the type of information that needs to be collected (e.g., to instantiate elements of the Trace model);
3) the encoding of the information, i.e., the amount of transferred information from one instrumentation component to another or to the JVM (e.g., the information transferred from aspects to the logger);
4) the logging of information itself, i.e., the mechanism to store the dynamic information (e.g., recording to a file).

Table 8 summarizes instrumentation characteristics with a few examples of typical practices for each characteristic. These characteristics lead us to systematically discuss sources of overhead in instrumentations. We can study the first characteristic to identify more efficient ways to gather the information such as selecting proper instrumentation technology, refactoring pointcuts, or advices. The second characteristic may lead to collecting less information at run time and compensate for the missed runtime information by additional static analysis of the code, though this may additionally require modifying the Trace model. The third characteristic is about minimizing the generated information from each instrumentation component to minimize the overhead. For instance, we can evaluate different character encodings. Finally, for the

fourth characteristic one can study different ways of storing dynamic information such as storing to a file.

Type of analysis	Characteristics	Examples of optimization practice
Dynamic or Hybrid Analysis	Info. Collection	• Choosing proper technology (e.g., AspectJ, LTTng) • Use selected technology in an efficient way (e.g., fewer interception points, refactoring AspectJ code for performance)
	Info. being collected	• Modify dynamic info. model to capture model elements statically • Derive dynamic model elements from other dynamic model elements
	Info. Encoding	• Use proper compression algorithm (e.g., trace compression, string compression) • Use proper data structures
	Info. Logging	• Use efficient logger (logging over network)

Table 8 Overhead characteristics of instrumentations

In the case of the Light instrumentation, our experiments from the previous chapter (Section 4.9) show that the logging mechanism and the collecting of information are the first and second largest contributors to overhead. Our intuition from previous chapters is that the encoding of information is also a major contributor to overhead, though such contribution has not yet been quantified to warrant optimization activities. Therefore, we study more precisely the overhead due to this characteristic in this chapter.

5.2 Research Questions

We conduct experiments in this chapter to answer the following RQs:

(RQ1) What are proper optimizations for each characteristic in the Light instrumentation and how much do they reduce overhead?

(RQ2) To what extent combining optimizations of all characteristics can reduce from the overhead in the Light instrumentation?

(RQ3) How effective is combining optimization strategies studied in RQ2 at reducing the probe effect?

5.3 Case studies and experiment set up

Similarly to the previous chapter, we use the same case studies[50] (Weka, OE, and Producer-Consumer in Table 1), the same test cases, the same hardware and software configurations, and the same measurement framework in conducting experiments in this chapter (refer to Sections 4.2 to 4.5 for more details). We can therefore compare data in this chapter with data from previous chapters. As mentioned in Section 4.5, we are targeting large systems to study overhead, and a typical multi-threaded system to study probe effects. Therefore, OE and Weka, and Producer-Consumer are representative enough to study overhead and probe effect respectively. Note that, when using a remote logger, we used a Dell PC with an Intel(R) Xeon(R) (at 2.66 Ghz) quad core, 16 GB memory, and 240 GB Solid-State drive (Kingston SSD SH103S3), running Ubuntu 16.04, JDK 1.8.0_111; but we didn't collect any execution time on this machine. In addition, we used a dedicated Ethernet network with 1 Gb/sec bandwidth, connected through a TP-Link switch (TL-SG105) and disconnected from the internet.

We created a dataset of 100 executions to report on the average execution time for each experiment in this chapter. In addition, we compared samples with a Student t-test, and differences were always statistically significant at α=0.05. We confirmed this with the Mann–Whitney *U* non-parametric test. Therefore, we do not report on those values separately for each experiment. Unless otherwise specified, reported results apply to both Weka and OE.

For all experiments in this chapter (except the ones mentioned explicitly), we verify the correctness of the experiments by comparing the trace from an experiment with

[50] The source code for case studies and experiments is available at: http://sce.carleton.ca/~mehrfard/repository/

the trace from the Light instrumentation. Experiments in Sections 5.4.1 to 5.4.3 collect the same information as what is collected by the Light instrumentation, so we compare traces in those experiment to Light's trace to verify the correctness of collected information in those experiments. In Section 5.4.4, we use an algorithm based on static and dynamic analyses to collect trace model elements. We verify the correctness of our algorithm by comparing generated traces by the Light instrumentation and the new algorithm, we explain this verification process in detail in Section 5.4.4.2.1. Finally, we verify the correctness of the optimized solution (when all optimizations are combined) by comparing with the Light instrumentation, we follow the same verification process as what is done in Section 5.4.4.2.1. We note that Leduc, Kolbah, and optimized solutions are producing the same information: Leduc's trace is identical to merged Kolbah's trace and control flow; Kolbah's trace is identical to optimized solution's merged trace and call graph.

5.4 Optimizing the Light Instrumentation

We suggest optimization for all four characteristics in this section: collecting of the information (Section 5.4.1), encoding of the information (Section 5.4.2), logging of the information (Section 5.4.3), and the information being collected (Section 5.4.4). For each of these four characteristics, we show the impact of overhead reduction based on an experiment where each experiment indicates the amount of overhead reduction due to alternative optimizations of that characteristic. For brevity, and since experiments are very similar to ones we already detailed in the previous chapter, we do not explain details of experimental analysis that resulted in those optimizations: details can be found in Appendix A. When looking into different optimizations in this section, it is important to

bear in mind whether applying standard software engineering design/coding practices to satisfy software qualities (e.g., re-usability), which are also employed in Kolbah's hybrid design, lead to overhead or not.

5.4.1 On the gathering of information

Our analysis on instrumentation overhead in Section 4.9 showed that AdviceOverhead, i.e., collecting and formatting information for the logger, accounts for the second largest amount of overhead. In addition, we explained how advices are implemented and woven in a typical AspectJ program in Section 2.1.3. Combining our overhead analysis results and AspectJ concepts, we suggest a series of optimizations to collect information, without reducing the amount of collected information, by looking at ways to optimize: weaving rules (Section 5.4.1.1), aspect implementation based on Java refactoring (Section 5.4.1.2), and aspect implementation based on AspectJ refactoring (Section 5.4.1.3).

Although the inter-type declaration we use for identifying object unique identifiers accounts for only 0.4% of overhead of the Light instrumentation (i.e., ObjectInterOverhead in Table 7), we look at alternatives for implementing object unique identifiers to reduce that overhead.

5.4.1.1 Weaving rules

Optimizing pointcuts involves optimizing the choices of join points for each advice, and optimizing where in the SUS to capture the chosen join points during its execution. The Light instrumentation intercepts interactions in the SUS based on `call` join points whereby pointcuts (Fig. 5) execute advices before each method (either static or not) and constructor call is made. Therefore, advices get executed before each method

call in the SUS. Among many AspectJ join points, such as `execution`, `preinitialization`, `initialization`, the `call` join point is the only one that can capture sufficient information, as required by the Trace model (Fig. 3) (e.g., we can identify the caller and the callee right away), with a minimum number of join points (i.e., with the smallest overhead). Recall that capturing fewer join points reduces the instrumentation overhead. We therefore found the choice of weaving rules in the design of the Light instrumentation to be adequate from an overhead point of view and decided to not change it.

It is however worth noting that a user of this technology, with a priori knowledge about the SUS could tailor instrumentation to parts of the SUS that are of prime interest, for instance avoiding GUI components. Also, optimizing weaving rules does not include capturing object identity information as it is not managed by advices in the Light instrumentation (rather the `IdentiferAspect` aspect identifies new objects through inter-type declaration right after each object creation).

5.4.1.2 Java refactoring

In aspects, the choice of data-types and the scope in which they are defined [165], and the modularization of the aspect code into functions [164] (i.e., increasing number of method calls (NMC_{Ins})) can impact overhead.

The `MethodAspect` and `IdentifierAspect` aspects use a local variable of type `LinkedList` (e.g., Fig. 6) to prepare the logging information to be passed to the Logger. In addition, `IdentifierAspect` (Fig. 11) weaves global `static` variables of type `int` to each class in the SUS. Refactoring this Java code with performance in mind may lead to

using fewer global variables to increase performance (local variables operate more efficiently in Java [171]). In addition, other Java data-types, such as `StringBuilder`, `String`, `byte`, `shortint`, may perform more efficiently compared to `LinkedList` and `int` as they may require less memory.

We performed different experiments (Appendix A.1.1) where we examined objects of type `LinkedList`, `StringBuilder` and `String` as well as primitive Java data-types `short`, `byte` and `String`[51], declared either as a class (or aspect) variable or method (or advice) variable. Except for primitive Java data-types, all other data-types in Java should be initialized as a new object before assigning to a variable. We limited our choices of data-type to those data-types mentioned above as they efficiently support string operations we required (i.e., insert, remove, return). Other Java data-types (such as `HashMap` and `TreeSet`) do other operations efficiently rather than efficiently performing basic operations on string literals. Our analysis shows (Appendix A.1.1) that among different combinations of Java data-types and declaration types, the local variable of type `String` has the best performance when string literals are assigned directly and no concatenation is used.

Another Java based refactoring could be to reduce modularity (i.e., reducing NMC_{Ins}) within aspects [168]. Reducing NMC in advices can be counted an important refactoring activity since reducing even one method call in an advice can translate into a considerable reduction in the total number of method calls because each advice may be executed millions of times. For instance the Light instrumentation uses helper methods in its advices (Fig. 9); their body could be copied directly into advices (a.k.a. in-lining

[51] In this case Java simulates the variable of type `String` similar to primitive data-types by assigning string literals directly to the variable.

method) at the expense of reusability since we are concerned with performance. Our analysis (Appendix A.1.3) shows that even though we reduced NMC_{Ins} within aspect by 45%, this reduced overhead by only 1% for OE. In addition, the reduction is statistically insignificant compared to Light (P-value=0.88). This suggests that the AspectJ compiler already replaces many of those calls within aspects with actual code. This is a refactoring we can keep in mind though, once other design decisions leading to more important overhead reductions have been employed.

5.4.1.3 AspectJ refactoring

In this section, we examine different refactorings of AspectJ aspects. We discuss: 1) the choice of advice, 2) different solutions for object unique identification, and 3) the choice of APIs for dynamic information collection.

With respect to the **choice of advice**, an `around()` advice would be more expensive than the current `before()` advice, as we noted in the literature review (Section 2.2.3.2). One advantage of a `before()` advice over an `after()` advice is that it keeps the order of invocations in the trace as they happen at runtime; an `after()` advice would require expensive post-processing to re-construct the correct order. We therefore keep the current `before()` advice.

A different **object unique identification mechanism**, not logging `objectID` during inter-type declaration, could be to define an `after():execution` advice for constructors to log the object identification information instead of logging during inter-type declaration. In this case we can replace lines for preparing and sending logs for each class in the `IdentifierAspect` (lines 6, 8, and 9 in Fig. 11) with a new

`after():execution()` advice in the `MethodAspect` (Fig. 21). In this case, no call to the logger will happen during inter-type declaration. However, such an advice would need to collect class name information dynamically (line 15 in Fig. 21) to compensate for the class name information provided (statically) by the missing inter-type declaration.

```
1    pointcut executeConstructor() : execution (PackageName..new(..)) &&
2    !within (instrument..*);
3
4    after(): executeConstructor () {
5       String thisID = new String();
6       LinkedList log = new LinkedList();
7
        try{
8          thisID = String.valueOf(((ObjectID) thisJoinPoint.getThis())
9    .getObjectID());
10      } catch (ClassCastException e) {
11         thisID = "Cought NonInstrumented Constructor";
12      }
13
14      log.add("<Lifeline className=\" "
15   + MethodAspect.getNewBindToClassName(thisJoinPoint.toString())
16   + "\" name=\" " + thisID + "\">");
17      Logger.getLoggingClient().instrument(log);
18   }
```

Fig. 21 Object identification with `after:execution()`

Our analysis (Appendix A.1.4) shows that using `after():execution()` advice almost does not change the performance of instrumentation (the Light overhead in OE is reduced only by 1% with a P-value equal to 0.47, which is statically insignificant reduction). This is a refactoring we can keep in mind though, once other design decisions leading to more important overhead reductions have been employed.

Yet another solution could be to remove all inter-typed `objectIDgenerator()` methods (Fig. 11) as well as calls to the logger in the `IdentifierAspect` aspect. In this case, we can identify object instances, again with the `after():execution()` advice (Fig. 21), and use a Java data-structure that counts, stores, and looks up object instances. For example, storing objects in a `HashMap` data-structure based on class name can be used for unique identification of objects. More specifically, we can remove all

`objectIDgenerator()` methods and replace them with the `IdGen` class (Fig. 22). The `IdGen` class uses class name as the key in a `HashMap` to assign values (i.e., `objectID`) to the `HashMap`.

We modified the `IdentifierAspect` aspect accordingly (Fig. 23). With new modifications, each time a call is made to the `getObjectID()` method in the `IdentifierAspect` aspect, the `getObjectID()` method checks whether the object has already been identified or not (line 4 in Fig. 23). If the object has not yet been identified (i.e., the value of `objectID` is null), this method calls the `getObjectId()` method from the `IdGen` class and passes the class name as a argument to it. The method looks up in the `HashMap` based on class name to check if any value has been assigned to the class name (line 23 in Fig. 22). If the value has not been assigned to a class name, the `getObjectId()` method adds the class name as a new key to the `HashMap` and assigns a new value to the new class name (lines 28 to 30 in Fig. 22). Otherwise, that particular class has more than one object instances and the method should return a new identifier for the object of that class (lines 24 to 26 in Fig. 22).

```
1    package instrument;
2
3    import java.util.HashMap;
4    import java.util.Map;
5
6    public final class IdGen {
7       private static IdGen singletonInstance;
8       Map<String, Integer> idmap = new HashMap<String, Integer>();
9
10      private IdGen() {
11      }
12
13      public static IdGen getSingletonInstance() {
14         if (null == singletonInstance) {
15            singletonInstance = new IdGen();
16         }
17         return singletonInstance;
18      }
19
20      public String getObjectId(String key) {
21         String id;
22         Integer value = idmap.get(key);
23         if (value != null) {
24            value++;
25            idmap.put(key, value);
26            id = key + value.toString();
27         } else {
28            value = 1;
29            idmap.put(key, value);
30            id = key + value.toString();
31         }
32         return id;
33      }
34   }
```

Fig. 22 IdGenerator class implementation

```
1    private String PackageName.ClassName.objectID;
2    declare parents : PackageName.ClassName implements ObjectID;
3    public String PackageName.ClassName.getObjectID() {
4       if( objectID == null )
5          objectID = IdGen.getSingletonInstance().getObjectId("ClassName_");
6       }
7       return objectID;
8    }
```

Fig. 23 Excerpt of new IdentifierAspect

Our analysis (Appendix A.1.4) shows mixed results when we modify the object identification mechanism with the suggested `HashMap` and `after():execution()` advice implementations. On the one hand, this modification has reduced the overhead by 13% for the OE case study as few objects are initialized in this case study (only three objects). On the other hand, we observed negligible overhead reduction (1% overhead reduction and statistically insignificant with P-value = 0.57) when applying this modification on the

Weka case study where hundreds of objects are created. Therefore, this modification does not change overhead for industrial-sized software.

The JVM does not provide any facility to uniquely identify objects over time; even methods such as `hashcode()` or `identityHashCode()` do not guarantee that two distinct values will be obtained for two different objects over execution time.

To collect dynamic information in an advice one can use **the AspectJ APIs, the Java reflection APIs, or directly change the Java code** (without using any API). As mentioned in the literature (Section 2.2.3), AspectJ promises to perform faster than the Java reflection API and or an equivalent ad-hoc Java implementation. It is possible to replace uses of the AspectJ API in the Light instrumentation with uses of the Java reflection API or an ad-hoc Java implementation, though with many more, likely less optimized, lines of code. For instance, we can replace the AspectJ reflection API that captures `objectID` in advices with the Java reflection API (such as replacing lines 13 to 15 in Fig. 6 with lines 1 to 15 in Fig. 24). Our analysis (Appendix A.1.3) shows that this modification in advices worsened the performance by adding 19% to the overhead when calls to Logger are disabled.

```
1       Try {
2          Method method = ((ObjectID)thisJoinPoint.getThis())
3    .getClass().getMethod("getObjectID");
4          thisID = (method.invoke((thisJoinPoint.getThis())).toString());
5       } catch (IllegalArgumentException e) {
6          e.printStackTrace();
7       } catch (IllegalAccessException e) {
8          e.printStackTrace();
9       } catch (InvocationTargetException e) {
10         e.printStackTrace();
11      } catch (Exception e) {
12         e.printStackTrace();
13      } catch (ClassCastException e) {
14         thisID = "Caught NonInstrumentedCaller";
15      }
```

Fig. 24 example of using Java reflection API to capture ObjectID

In another experiment (Appendix A.1.3), we replaced helper functions in the Light instrumentation (lines 104 to 129 in Fig. 5) with AspectJ APIs (such as using `getLine()` and `getSignature()`). Applying this modification on OE, while disabled calls to Logger, shows 40% overhead reduction over the Light instrumentation, though statically insignificant (P-value=0.051). As we noted in Section 2.2.3.2, this performance boost was expected. Often, calling an intrinsic method in a standard library (AspectJ library in this case) to execute an operation is faster than implementing that operation with a helper method or hard coding that operation in a method (or advice) body, because a compiler can replace the whole operation with a single assembly/machine code at compile time. Using AspectJ APIs to replace the helper methods is similar to inlining helper methods (the optimization mentioned in Section 5.4.1.2), yet the former optimization is based on AspectJ refactoring and the latter optimization is based on Java refactoring.

5.4.1.4 Combining optimizations for collecting information

We conducted an experiment to understand the amount of overhead reduction when all successful optimizations discussed previously are combined to answer RQ1 for the first characteristic.

5.4.1.4.1 Experiment design

We developed a new version of the Light instrumentation, which we refer to as Partially Optimized 1 (PO1) in which: each local `LinkedList` variable is replaced with a local `String` variable in advices and as parameters to calls to the logger, no concatenation is used in `String` variables, and helper methods are removed either by

using the AspectJ API (e.g., to obtain a line number) or by copying their body where necessary. We used the Linux `time` command to measure the execution time for the OE and Weka case studies. Before measuring the time, we verify the correctness of advices by comparing the generated traces obtained from PO1 with the traces obtained with the Light instrumentation.

5.4.1.4.2 Results

Optimizations on collecting information (third column in Table 9) show a 15% overhead reduction for Weka_TC4 and 29% overhead reduction for OE_TC2. The difference in overhead reduction between OE and Weka indicates the differences between case study systems (e.g., number of created objects).

Case study	Light	PO1
OE_TC2	15.44 (100%)	10.87 (-29%)
Weka_TC4	17.51 (100%)	14.89 (-15%)

Table 9 Average execution times (in seconds) for Light and PO1

5.4.2 On the encoding of information

The (dynamic) trace information is a string of characters which is eventually converted into bytes in computer memory. Passing a lesser amount of information from one instrumentation component to another or to the JVM without losing any piece of information could reduce the amount of computer resources (memory and CPU) and therefore the overhead. The Light instrumentation can be optimized in four ways with this respect: 1) choosing a proper Java character encoding type, 2) generating a lesser amount of characters for each log item in advices and inter-typed methods, 3) condensing

the information passed by aspects to the logger based on existing trace formats, and 4) the mechanism by which the information is passed from aspects to the logger.

The first way to optimize can be to reduce the string (used to represent dynamic information) size by choosing a proper Java encoding. Aside from the mentioned Java data-types (`LinkedList` and `int`) used to record dynamic information in memory, the information is converted to bytes from those data-types based on the default Java character encoding which is specified by the type of JVM. Using a proper Java encoding with a minimal required space at compile or run time, such as ISO-8859-1 or UTF-8 (8 bit encodings), or US-ASCII (7 bit encoding), can reduce the amount of memory being used by aspects and the logger. We tried different character encodings in our instrumentation other than UTF-8, the default JVM encoding. Our analysis (Appendix A.2) shows there is no substantial difference between different encodings (US-ASCII, ISO-8859-1, UTF-8, UTF-16). The US-ASCII character encoding performed slightly better for the OE case study by reducing overhead by nearly 2%, though this overhead reduction is statistically insignificant (P-value = 0.18). However, we note that this optimization further reduces overhead when we combine the most efficient character encoding (i.e., US-ASCII) with the most efficient Java data-type (i.e., `String`).

For the second optimization, generated log items in the Light instrumentation contain dynamic information similar to XMI, which requires some (formatting) processing. Instead, the overhead of generating the trace can be easily reduced by capturing the raw information without formatting (i.e., shortening log items), and do the formatting during post-processing. For example, Fig. 25 (top) shows a sample of a formatted log item from Weka_TC4. The bottom figure contains the same runtime

information, but without any specific format. In this particular case, the size of the formatted log is 329 bytes, whereas, the unformatted log is 172 bytes. Our analysis (Appendix A.2) shows that this optimization reduced the trace size by 66% and the overhead by 25% for the OE_TC2 test case.

```
<messageLog bindToClass="weka.core.converters.ConverterUtils$DataSource"
messageSort="synchCall" signature="void
weka.core.converters.ConverterUtils.DataSource.reset()">
    <sendEvent covered="DataSource_1"/>
    <receiveEvent covered="DataSource_1"/>
    <sentFrom lineNumber="142" name="ConverterUtils.java"/>
</messageLog>
```

```
weka.core.converters.ConverterUtils$DataSource, synchCall, void
weka.core.converters.ConverterUtils.DataSource.reset(), DataSource_1, DataSource_1,
142, ConverterUtils.java
```

Fig. 25 Top: formatted log, Bottom: unformatted log

In addition, SUS packages and class names with long character strings impose more overhead as we collect the class name information for caller, callee, and object identification. Our analysis (Appendix A.2) shows that indeed classes with large names increase the runtime overhead as well as trace size. Thus, mapping fully qualified class names to integer identifiers (e.g., with a hash table) can solve this issue, though at the expense of some processing (i.e., use of a hash table); a trade-off needs to be examined.

We implemented this optimization, whereby the following model elements are mapped to integer identifiers: callee's class name (i.e., the name of class whose instance executes the called method), callee's method signature (i.e., the method being called), caller's class name (i.e., the class from where the call has been made) and caller's and callee's `objectID`s (recall that `objectID` strings are built by appending class name of an object instance to an integer object counter). We modified each static class structure of the SUS in IdentifierAspect (i.e., Fig. 11) to map callee's class name and object identification to integer identifiers using a hash table, whereby two changes are added:

first, new attribute `classNameID` (line 1 in Fig. 26), which holds a unique integer as a class name identifier; and second, new implementation of interface `getClassNameID()` (line 11 in Fig. 26), which is defined in the `ObjectID` interface and passes the `classNameID` attribute value to advices. In addition, lines 8 and 19 in Fig. 11 are modified with lines 5 and 16 in Fig. 26 to pass integer identifiers instead of actual class names. We use a hash table to map class names to integer values. We implemented such a hash table in the `ClassId` class, similar to the implementation described in Fig. 22. Another model element, caller's class name, directly accesses to the same hash table, `ClassId` class, to encode the class name to an integer identifier. Since the callee's class name is available at compile time in IdentifierAspect, the hash table can be built right prior to instrumentation, which makes class name mapping to integer numbers more efficient (no overhead). However, caller's class name mapping to hash table takes place at runtime by directly accessing the hash table since caller's information is not available beforehand. Finally, similar to the caller's class name, method signatures are directly mapped to another hash table at runtime (implemented in `MethodId` class). Once execution of a the SUS has finished, values of the hash table are printed at the end of the trace.

Our analysis based on OE (Appendix A.2) shows slightly better performance (7% overhead reduction and statistically insignificant result (P-value = 0.119)) with 33% smaller trace size (759 MB compared to 989 MB) for this optimization. We expect greater overhead reduction for Weka since class, package, and method names are longer.

```
1    private static String PackageName.ClassName.classNameID =
     String.valueOf(ClassId.getSingletonInstance().getClassId("ClassName"));
2    .
3    .
4    .
5    log.add("<lifeline className=\"" + ClassName.classNameID + "\" name=\""
6    + ClassName.classNameID + "_"+ id + "\"/>");
7    .
8    .
9    .
10   declare parents : ClassName implements ObjectID;
11   public String ClassName.getClassNameID(){ return ClassName.classNameID; }
12   public String ClassName.getObjectID() {
13      if (objectID < 1){
14         objectID = ClassName.objectIDgenerator(objectID);
15      }
16      return ClassName.classNameID + "_" + objectID;
17   }
```

Fig. 26 Optimized static class structure

The third way to optimize encoding is to make sure the aspects generate trace information according to a well-known trace format by applying compression algorithms or using condensed tree data structures (recall the related work in Section 2.2.3.3). While a more condensed format like that reduces the amount of memory being used and leads to sending lesser information to the logger, it imposes more computation to actually format the information. Depending on the system resources, the trace length, and the call chain type (i.e., multiple similar call chains), this kind of optimization may reduce the overhead, but it may not (formatting processing may be expensive). We have not conducted any analysis based on this optimization and the trade-off needs investigation, though our intuition is this optimization will improve performance marginally, if it does.

Lastly, we can optimize the way logs are passed to the logger: currently, a `LinkedList` is passed as argument to calls to the logger. The logger handles all the details of the passed information with a unique parameter, regardless of the type of each log item (e.g., logs for constructor are different from others though there is only one method in the logging API). Fewer passed arguments slightly improves the performance

in Java [171]. Such a generic logger can be specialized to the different types of logs. This would reduce a little bit the amount of information passed to the logger (e.g., no need to pass the kind of call); Since the logger is invoked for each method call in the SUS, a gain, even small, for each call can count. For this optimization, we implemented a specialized logger with four different methods to call the logger from aspect based on the type and the number of passed information pieces in each advice and inter-type declaration. The comparison of this logger with the generic logger in Light should disclose potential performance gain. Our analysis based on OE (Appendix A.2) shows a slight statistically insignificant (P-value = 0.55) overhead reduction (1% reduction) when we comparing the specialized logger (with more methods and fewer arguments) with a generic logger. Even though the performance gain might sound negligible, it may be considerable once larger contributors to overhead are removed.

5.4.2.1 Combining optimizations on encoding information

We conduct an experiment in this section to understand the amount of overhead reduction when all successful optimizations are combined to answer RQ1 for the third characteristic.

5.4.2.1.1 Experiment design

We develop new versions of the Light instrumentation, which we refer to as Partially Optimized 2 and 3 (PO2 and PO3) whereby in PO2, we use `String` instead of `LinkedList` for local variables; we use the US-ASCII character encoding instead of UTF-8 encoding; we use a specialized logger instead of a generic logger for each advice in that calls to logger in specialized logger are optimized based on the type advice (i.e.,

the type of information passed to the logger); and we use raw data instead of the textual format originally used in Light when transferring data to the logger. In PO3 instrumentation, we keep all the optimizations of PO2 in addition to encoding class names and method names to integer values. We use the Linux `time` command to measure the execution time for the OE and Weka case studies.

5.4.2.1.2 Results

Optimizations on encoding information (third and fourth columns in Table 10) show between 26% to 36% overhead reduction for both Weka_TC4 and OE_TC2. It is important to note that PO3 reduces trace size by more than 90% while the overhead due to encoding class and method names to integer identifer is negligible. It is interesting that the trade off between trace size and computational overhead in PO3 causes larger overhead reduction for OE_TC2 and smaller overhead reduction for Weka_TC4 compared to their respective execution based on PO2. OE contains fewer classes and methods compared to Weka, therefore the lookup overhead in the hash map is much smaller for OE. We think it is possible to improve the current implementation that uses a hash map (which is based on `LinkedList`) in PO3 by using a self-balancing tree data structure. This change would reduce the overhead of lookup from O(n) to O(log(n)) in a worst case scenario. Comparing Table 9 and Table 10 indicates that optimizations on encoding of information are more effective than optimizations on collecting information at reducing the overhead.

Case study	Light		PO2		PO3	
	time (sec)	size (MB)	time (sec)	size (MB)	time (sec)	size (MB)
OE_TC2	15.44 (100%)	989.6 (100%)	10.46 (-32%)	241.6 (-75%)	9.92 (-36%)	67.2 (-93%)
Weka_TC4	17.51 (100%)	1250 (100%)	12.22 (-30%)	460 (-63%)	12.93 (-26%)	102.7 (-92%)

Table 10 Average execution times (in seconds) for Light, PO2 and PO3

5.4.3 Logging mechanism

The mechanism to log runtime information composes a major part of overhead in the Light instrumentation (as shown in Section 4.9.2). As previously mentioned, during the Light instrumentation, whenever an advice (or inter-typed method) executes, the `instrument()` method in the `Logger` class is called (synchronous call), which then reads log items from the passed `LinkedList` variables in memory and writes the information to the disk.

There are two main sources of overhead in the current logging mechanism: 1) the high coupling between the log generation process (i.e., aspect instrumentation) and the log storage process (i.e., the logging mechanism) which happen in the same thread of execution. Reducing this coupling would reduce overhead. 2) For a typical program, the operation of writing to the disk adds largely to the runtime overhead, as there is one such operation per method call (either static or not), and two operations per constructor call. Thus, having fewer accesses to the disk could reduce the logging overhead.

In general, there are two main approaches for storing dynamic information: logging on the same machine as the instrumentation executes (Section 5.4.3.1) and logging on a remote machine (Section 5.4.3.2). In this section, first we discuss each of

these logging approaches and next we evaluate the amount of overhead reduction by the most efficient logging mechanism through case studies (Section 5.4.3.3).

5.4.3.1 Logging on the same machine

In this approach, instead of the current logging mechanism, which writes each log item on file as soon as they are generated, we can fill a buffer of logs in memory with a log generator thread (i.e., aspects) and flush the buffer to local disk each time the buffer(s) is full with the logger thread(s). For instance, we can use a producer-consumer model to implement this design. This way, 1) the aspect does not interact directly with (i.e., call) the logger and the instrumentation ends faster due to sending log items at a faster pace to the buffer (i.e., faster computation than I/O), 2) we reduce the number of accesses to the disk and therefore overhead, at the expense of longer disk accesses. Although the use of a buffer reduces the overhead, the high performance of this type of logging mechanism is restricted by the capacity of memory and the disk technology (e.g., SSD). We need further investigation to understand the impact of memory limit and disk technology on overhead reduction. However, we can argue intuitively that in situations where the aspect instrumentation generates a large number of log items (i.e., large trace), the faster pace of log generation over the pace of log storage (I/O operation) may eventually exceed the queue capacity and slow down the execution of aspect instrumentation. In addition, although the aspect instrumentation may not interact directly with the logger, the log storage process has a negative effect on the aspect instrumentation as it consumes resources.

We investigate the possibility of reducing overhead in multi-threaded instrumentation with a local logger using a buffer by proposing two implementations in

Section 5.4.3.1.1, namely: *CacheLogger* and *BatchLogger*. Next in Section 5.4.3.1.2, we analyze the performance of these implementations and verify our intuitions based on empirical results (Appendix A). Our research questions are:

RQ1: Which algorithm works with a better performance and what is the optimum size for cache (in bytes) and batch (in numbers of log items) in each algorithm?

RQ2: How much the new logging algorithms improve the performance over Light?

RQ3: What is the impact of different OS file formats on a logging mechanism?

We used the machine with configuration mentioned in Section 4.3.3 to answer RQ1 and RQ2 and the server with configuration mentioned in Section 5.3 to answer RQ3, except we used HDD[52] (instead of SSD) while running dual operating systems: Ubuntu 12.04 and WindowsXP.

5.4.3.1.1 Implementation

We implement two algorithms for new loggers based on a producer-consumer design: one log generator thread (i.e., aspects) generates log items, while CacheLogger uses one thread and BatchLogger uses two threads to store log items on disk.

In CacheLogger (Fig. 27), the producer thread puts log items, received from the aspects, in a queue as in a memory buffer. We used a `LinkedBlockingQueue` data structure in Java since it is thread safe and maintains the order of logging data. Each log in `LinkedBlockingQueue` is a string representing a "log item". Simultaneously, CacheLogger (the consumer thread) removes the `<String>` of a log item from the queue

[52] Hitachi HDP72503 - 320 GB

and adds the log item to a `LinkedList<String>` Cache (second buffer). If the Cache size reaches the Cache limit, the CacheLogger flushes the log items of the `LinkedList` Cache to the disk. The CacheLogger uses a variable of type `BufferedWriter` to take log items off the Cache and to flush them to the disk. Depending on a system configuration and resources, different Cache sizes may perform differently. The user can change the Cache size to understand with which Cache size CacheLogger performs the best (by changing the `cacheSize` variable). When aspects (the producer thread) finish generating log items, which indicates the end of execution of the Light instrumentation of the SUS, the aspect code calls the `forceFlush()` method in the CacheLogger. The `forceFlush()` method removes the remaining items from the queue, adds them to the Cache, and flushes them to the disk. It is imperative to synchronize both cache and queue during the logging process to not lose any log item when the consumer thread removes all log items from the queue.

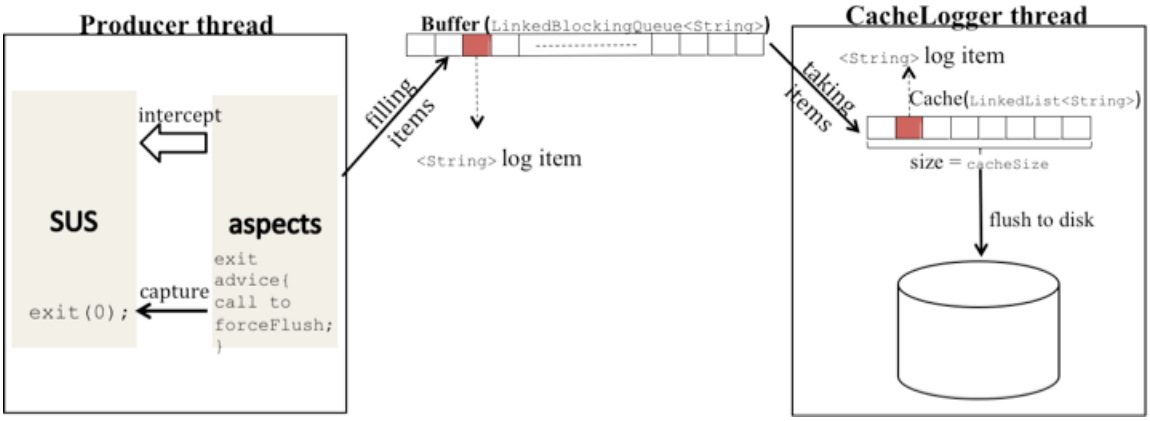


Fig. 27 CacheLogger

In order to call `forceFlush()` on the Logger, the aspect code needs to call this method just before the termination of program execution (i.e., SUS). We identify the end of program execution by defining a new pointcut that captures call to the exit signal in the

SUS (i.e., using `before():call(System.exit())`). Once the end of the program is captured, the advice body calls `forceFlush()`. If there was no exit signal at the end of program execution, the SUS needs to be changed slightly by inserting the exit signal at the end of program (i.e., adding `System.exit()` to the end of the main class called in the Java path). Although, it is preferred to not change the SUS, to the best of our knowledge, AspectJ does not provide any pointcut designator to capture the end of program execution. Thus, we need to mark the end of program execution with the exit signal to be captured later by an aspect. Alternatively, we can find the end of program execution by defining the *shutdown hook* thread in the advice body after the execution starts. In Java, the shutdown hook thread is designed to be run after a program receives a signal to exit, but before daemon threads stop. Once the program execution ends, the shut down hook calls the `forceFlush()` method. Again, we need to insert an exit signal at the end of the program to activate the shut down hook. We opted for the first solution since it is less intrusive.

We designed another logger, BatchLogger, this time replacing a `LinkedList<String>` Cache with batches of `ArrayList<String>` (Fig. 28). In this case, BatchLogger includes two threads to manage the buffering: LogAggregator thread and LogWriter thread. The producer thread in the BatchLogger design works similarly to the producer thread in the CacheLogger design and puts log items to the buffer. Simultaneously, on the consumer side, the LogAggregator thread takes log items off the `LinkedBlockingQueue` queue (i.e., buffer) and creates batches of log items (based on an `ArrayList<string>` data structure); the LogWriter thread reads each batch (i.e., a list of log items) and uses a variable of type `BufferedWriter` to flush the batch to the disk. It is

worth noting that the user can specify the capacity of the batch (i.e. the number of log items) in the LogAggregator by changing the `queueBatchSize` variable. We implemented the `forceFlush()` method and the way it is called similarly to the way we discussed earlier.

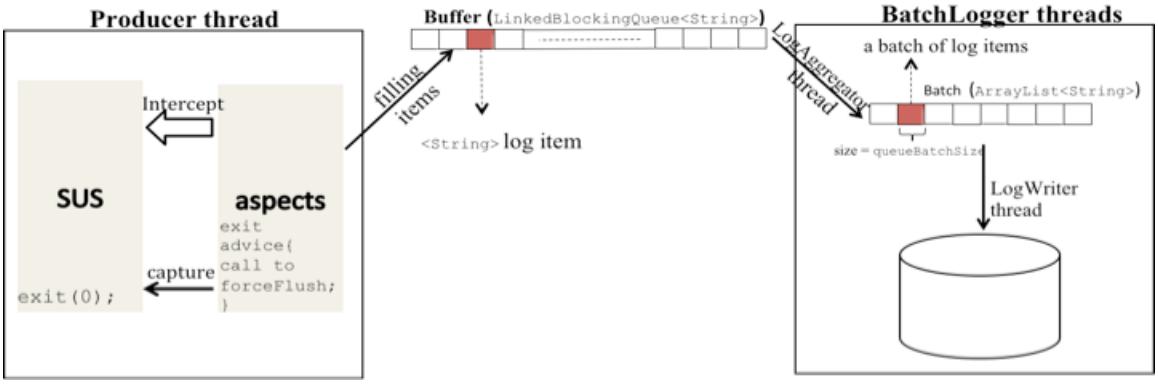


Fig. 28 BatchLogger

If we use more than one LogWriter thread in the BatchLogger to flush batches to disk, the performance will be degrading. This is due to the fact that the write to disk operation is a sequential operation in our disk technology and using more than one thread for LogWriter only consumes more resources without enhancing the write operation.

5.4.3.1.2 Analysis

Our empirical results with respect to RQ1 (Appendix A.3.2) show both loggers perform with similar overhead. When we set the value of `cacheSize` and `queueBatchSize` to 10,000, the CacheLogger and BatchLogger show the best performance in our configuration by performing OE_TC2 in 6.88 seconds and 6.42 second respectively (compared to Light execution of 15.44 seconds). However when we increased the number of threads, BatchLogger performed worst (this observation is not reported in the Appendix though). With respect to RQ2, the result (Appendix A.3.2)

shows the CacheLogger implementation enhances the performance at least by 51% for OE_TC2 and Weka_TC4 compared to the execution of those test cases based on Light. Again, this is an indication of the importance of logging optimization to reduce overhead in Light. Note that even though our results are obtained based on SSD technology, as recommended in the literature (Section 2.2.3.3), the CacheLogger implementation would boost the performance of HDD technology as well by reducing the number of disk accesses (i.e., disk look up overhead). Finally, to answer RQ3, we conducted an experiment (Appendix A.3.2) on WindowsXP and Ubuntu 12.04 based on the HDD technology to uncover the efficiency of the write operation based on OS file formats. As we expected (recall Section 4.7.3), even though we disabled all antivirus and network activities during the experimentation, the Windows file format (i.e., NTFS) performed worse than the Linux file format (i.e., Ext4). Our results are in line with what others reported about file systems performance differences (e.g., [207]). With the high volume of data storage demand in our instrumentation approach, the Linux file format is more suitable than the Windows file format.

5.4.3.2 Logging on a remote machine

In this approach, the storage process happens in a log server machine and log items are sent over the network, possibly combined with a buffer on the client (log generation) side. The use of buffer and two threads on the client side, similarly to the producer-consumer model we explained for the local logger implementation, reduces the coupling between log generation and the I/O for log storage: one thread to generate aspect and another thread to convert data into packets and send over the network. Log generation, which involves CPU operations, generally outperforms I/O operations.

Therefore, using two threads and a buffer, despite imposing minor overhead, ensures that CPU operations are not interrupted by I/O operations. If the network transfer rate is faster than log generation, buffer remains empty. However, in case the transfer rate is slower than log generation, which is more common, the buffering mechanism ensures no log will be missed until buffer size reaches its capacity. The client logger sends log item(s) to the log server as a stream of packets. Therefore, the overhead of the storage process in the Light instrumentation (with expensive, repeated writes to the disk) is replaced with the overhead of converting log items to network packets based on the selected network protocol and the overhead of sending packets to the log server through the network. This logging mechanism is bounded by the throughput of the network: a slow network communication (either due to the selected protocol or network configuration) causes a bottleneck and consequently hurts performance. In addition, depending on the protocol being used for transferring log items, there is a chance to lose some of the transferred packets from the client to the server.

There are different ways to implement a remote logger. First, we considered existing common logging frameworks such as Log4J[53], since it requires less implementation efforts compared to implementing a new remote logger. Log4J is a common logging solution for Java software and claims to be a high performance logging framework [189]. However, our analysis (Appendix A.3.1) shows the Log4J framework, for instance, leads to too much overhead by either not providing a large enough buffer size or collecting unnecessary data, thereby imposing additional overhead. Therefore, we do not use any existing logging framework for our instrumentation as they proved to be inadequate.

[53] http://logging.apache.org/log4j/1.2/

Another alternative is to implement a custom remote logger. We replace the generic logger in the Light instrumentation with two versions of a custom remote logger based on the TCP and UDP protocols while keeping everything else unmodified. Our experiments (Appendix A.3.1) show that the UDP is not as reliable a protocol as TCP to transfer (trace) data since the UDP logger lost nearly 2% of the data. The amount of data-loss in UDP depends on network configurations, network routers, distance between client and server, etc. Therefore, we choose TCP over UDP for optimizing the logging mechanism.

5.4.3.3 Optimizations on logging information

We conduct an experiment in this section to understand the amount of overhead reduction when we use the TCP based custom remote logger to answer RQ1 for the third characteristic.

5.4.3.3.1 Experiment design

Generally, operating systems manage I/O operations at the kernel level by a system-call. They however provide a system-call interface for applications to handle several categories of I/O operations, including sockets. For example, applications use blocking and non-blocking system-calls to manage a socket. A blocking call suspends the execution of the application until the completion of a system-call (i.e., I/O in this case), while a non-blocking call does not halt the application execution for an extended amount of time, and instead returns to the application immediately [208]. For instance, a blocking `send()` operation for a socket suspends the execution of the calling application until all data has been transferred through the socket to the server, while a non-blocking `send()`

operation returns to the execution of the application right after initiating a data transfer in the socket. As we mentioned in related work (Section 2.2.3.3), using a non-blocking system-call is recommended for High Performance Computing. Fortunately, we do not need to directly use the system-call interfaces to handle TCP in Java since there are two packages to handle blocking and non-blocking TCP: the `java.net` package and the `java.nio` package, respectively. In practice in Java, a blocking TCP stream accesses a sequence of bytes of data (i.e., byte-oriented), whereas a non-blocking TCP stream accesses blocks of data (i.e., block-oriented). A socket uses a channel in non-blocking TCP to transfer data. Therefore, a socket transfers blocks of data in non-blocking TCP compared to transferring bytes of data in blocking TCP, resulting in a faster transfer rate in non-blocking TCP. In addition, we implemented the server in non-blocking TCP to accept more than one client at a time, paving the way for potential future extensions to distributed logging.

As we mentioned before, we developed new versions of the Light instrumentation, which we refer to as Partially Optimized 4 and 5 (PO4 and PO5), whereby: we replaced the generic logger with the blocking TCP logger in PO4 and non-bocking TCP logger in PO5 while keeping everything else unmodified on the client side.

5.4.3.3.2 Results

Optimizations on logging information are shown in the third and fourth columns of Table 11. We show two execution times for the optimized instrumentation: $time_{SUS}$ and $time_{log}$. The $time_{SUS}$ is the amount of time to execute the SUS, which is calculated by defining two join points and capturing time stamps at the very beginning and end of the execution of the SUS. Optimization in third and fourth columns of Table 11 shows

overhead reduction based on $time_{SUS}$ for both Weka_TC4 and OE_TC2 test cases by more than half for PO4 and by more than 30% for PO5. Higher rate of converting logs to TCP packets causes more overhead in PO5. Comparing $time_{SUS}$ in both PO4 and PO5 with the time in PO1, PO2 and PO3 (Table 9 and Table 10) shows this optimization was more effective than the previous two optimizations, as we expected. The $time_{SUS}$ in Table 11 is comparable with the time in Table 9 and Table 10 since the potential behaviour change due to overhead only happens during the SUS execution.

On the other hand, $time_{log}$ shows the amount of time that the client virtual machine has to wait after the end of the execution of the SUS for all the packets to be transferred from the client to the server. This time is calculated by subtracting the $time_{SUS}$ from the client execution time (using Linux `time` command). Table 11 shows PO4 has very long waiting time on the client side (i.e., $time_{log}$) for both case studies, whereas PO5 has relatively short waiting time on the client side (i.e., $time_{log}$) at the expense of slighltly higher overhead (i.e., lager $time_{log}$). The risk for PO4, although we have not observed it, is that this may translate into overhead for the SUS if buffers used for network communication get full. As we expected, non-blocking TCP transfers data with a much faster rate compared to blocking TCP due to using channel, therefore the $time_{log}$ for PO4 is greater than the $time_{log}$ in PO5.

Case study	Light	PO4		PO5	
		$time_{SUS}$	$time_{log}$	$time_{SUS}$	$time_{log}$
OE_TC2	15.44 (100%)	6.59 (-57%)	126.95	9.92 (-36%)	4.08
Weka_TC4	17.51 (100%)	8.35 (-53%)	147.82	11.77 (-33%)	5.54

Table 11 Average execution times (in seconds) for Light, PO4 and PO5

We conclude that different criteria must be accounted for when optimizing the logging mechanism, including network throughput, network bandwidth, overhead due to encoding data for network transmission, system resources (memory and CPU), type of communication protocol, and that these criteria have various, perhaps conflicting impacts on a given solution.

5.4.4 On the information being collected

Reducing the amount of information collected at runtime is another optimization area for the Light instrumentation. To investigate to what extent this is possible, we need to answer two research questions: RQ1: is it possible to reduce the runtime overhead by deriving any model element from other model elements in the trace model? RQ2: is it possible to reduce the overhead of the dynamic analysis by not intercepting all execution paths in the SUS and compensating for the missing information on those paths with some static analysis?

5.4.4.1 On deriving a model element from other model elements

For RQ1, the objective is to further modify the trace model in that no model element can be derived from other model elements since this suggests redundant information that needs to be collected at runtime. Looking at all model elements in the trace model in Fig. 3 and their equivalent runtime information from the implementation, there are a couple of elements that can be derived from other model elements. First, the `bindToClass` attribute (the callee's class name) can be derived from the `signature` attribute (the callee's signature), because the callee's signature already contains the class name. In addition, in cases where a caller is a static method, there is no need to capture the `Covered` attribute with the `sendEvent` association. In these cases the `Covered`

attribute value can be derived from the `name` attribute (the caller's class name) in the `SourceLocation` since there is no object instantiation.

5.4.4.2 On capturing trace model elements with static analysis

For RQ2, we need to investigate different possibilities whereby all (or part of) the information required by the trace model is captured before execution through static analysis. For example, if a callee object is the only instance of a particular class during the execution, if this can be inferred a priori, and if a method of that class is called, we can capture all the required information of this object interaction during compilation without the need to intercept the SUS for this object interaction.

We turn to static analysis to examine the possibility of building sound object interactions. The closest data structure in static analysis to represent a sequence diagram is the call graph [96]. Recall that a call graph contains calling relationships between methods. Yet building a completely *sound* (i.e., avoid missing possible calls or data-flow) and *precise* (i.e., avoid considering impossible calls or data-flow) call graph for arbitrary programs is undecidable since it requires solving the halting problem [7-9]. More specifically, dynamic dispatch (late binding) is the biggest source of uncertainty in building call graphs. Therefore, building a sound and precise call graph for the execution of an arbitrary program would be possible only if dynamic analysis compensates the missing information in dynamic dispatch.

As we mentioned in the literature review (Section 2.2.1), there are different call graph construction algorithms. We choose the VTA (Variable Type Analysis) algorithm since it is relatively sound and precise and computationally more cost-effective compared to other algorithms. For each call, a call graph contains caller signature and its class name

(i.e., a node), a call site with a line number (a statement in the caller method where the callee method is invoked), and a callee signature and its class name (i.e., another node). In other words, a caller and a callee method are two nodes in the call graph, which are connected by an invocation statement in the caller as an edge in the graph. If a call site is a dynamically dispatched message, it is possible the call graph shows multiple callees. In addition, a call graph identifies the type of calls (i.e., type of invocation statements) in the graph, which is either "invokevirtual", "invokestatic", "invokeinterface" or "invokespecial". The VTA algorithm obtains the type of call based on the bytecode of the SUS. If a type of call in a call graph is "invokstatic" or "invokespecial", then the JVM resolves those calls at compile time (i.e., static dispatch). An "invokestatic" call shows a call to a class method, while "invokespecial" demonstrates specific types of calls to an instance method; these calls are either to constructor, or to private methods or to super().

Comparing the trace model (Fig. 3) with the call graph information for a call, assuming a call is sound and precise in the call graph, one can capture all pieces of information from the call graph for that call to build the trace model except object identifier for the caller and the callee (i.e., `Covered` attribute). Even though it is possible to partially construct object identifier information for static dispatch in a call graph (by looking at the context of caller and callee), producing the value of this attribute for all calls depends on which object has actually been instantiated and called at runtime. Therefore, dynamic analysis is needed to compensate for this missing information.

We suggest a modified instrumentation of Light based on call graph information and justify why the proposed hybrid approach is lossless. Depending on how to combine instrumentation information with call graph information, there are two approaches to

optimize the Light instrumentation: basing the analysis on call graphs and compensating the missing information with instrumentation or basing the analysis on instrumentation and compensating the missing information with call graphs. Obviously, we favour an approach with larger overhead reduction. Therefore, we base our analysis on call graphs and compensate the missing information with instrumentation. To do that, the call graph has to keep the order of calls as they occur according to the call site context (i.e., a flow sensitive call graph). We modify the Light instrumentation whereby, we remove `callStaticMethod` and `callConstructor` advices and modify the `callMethod` advice to not intercept calls to private methods anymore (modified line 3 in Fig. 5 by adding `!private !final` before `!static`). This way the instrumentation intercepts non-private and non-final calls to instance methods and objects once they are created (in `IdentifierAspect`). Considering these changes, we justify it is possible to safely construct the trace based on call graph information (or vice versa).

For calls to constructor or static method or private method, a flow sensitive call graph can soundly produce caller's and callee's signatures, and invocation statement and its line number based on static dispatch. Yet we further analyze the possibility of object identification for those calls based on call graph information. For the rest of calls (dynamic dispatch or calls to non-private instance methods), instrumentations happens at runtime. Recall that since the AspectJ compiler is incapable of intercepting calls to the `super()` methods, therefore this static dispatch is not accounted for in this work, i.e., the corresponding calls are not intercepted by our instrumentation.

If there is a call to a constructor at runtime, it is possible to identify the created object (i.e., callee) based on call graph information. This can be done by setting a counter

for each class and incrementing that counter each time a call to constructor has happened in a call graph path. However, it would be difficult to match these created objects (or lifelines) with objects of called instance methods a trace since object identification mechanisms are different in a call graph and a trace. Therefore, we turn to dynamic analysis to identify the created object instance in `IdentifierAspect` by printing out the class name and identifier of created object in the trace. The matching between call graph and trace happens based on order of calls in the call graph and the trace, signature of called constructor in call graph and class name of constructor in trace. It is important to note that our instrumentation shows all super-classes in the trace when an object of a sub-class is instantiated. Recall that we ruled out uninitialized super-class in the trace by matching object identification information in `IdentifierAspect` with the `callConstructor` advice in `MethodAspect`. Since the `callConstructor` advice is disabled, it is possible to identify the correct instantiation of object from inherited class in the trace by choosing the last identified object in the trace based on its class hierarchy (recall Section 3.2.1.2 that object unique identification mechanism prints an inheritance hierarchy once an object is created). In addition, it is also possible to remove uninitialized object from the trace by looking at those calls to instance methods that happen right after object creation, because identified objects usually make calls after their creation. In case there are two immediate calls to constructors of a class with different signatures, even though inter-type declaration does not retain constructor signature, it is still possible to match object instances in the trace with constructor signatures in the call graph based on the order in which constructors have been called. For the object identifier of a caller, if the caller is a static method, constructor or a private method, the call graph uniquely

identifies the object instance of those calls. If the caller is a non-private instance method, even though we cannot learn the object instance until run time, we can identify the caller's object based on trace history: we search the trace for the last occurrence of the caller (or any instance method in caller's class) who called another method or was called by another method, then capture the object identification of that occurrence and use it as the caller object identifier.

If there is a call to a static method at runtime, the callee object can be uniquely identified based on the call graph (because a static method belong to a class and this biding happens at compile time). A call to a static method can happen either from another static method or from an instance method. If a caller is another static method, the class instance can be uniquely identified based on the call graph information. If a caller is an instance method, the object of that method should be an instance of the class from where the static method also has been called. In such cases, similar to calls to constructors, looking back at the trace to where the callee has made a call or been called identifies the instance of the caller.

If there is a call to a private or final method at runtime, it is possible to learn caller and callee object instances from a call graph. Recall that private methods are only accessible by the objects of a class in which they are declared (i.e., not overridden or accessible in sub-class or other classes). Therefore, since a called private method is not overridden, a compiler knows the callee method and uses static dispatch to uniquely identify the called method. Therefore, similar to calls to a constructor or a static method, we search the trace history for the last occurrence of instance methods of the callee's class or object instantiation of the callee's class. Once such a method is found, we

identify the object associated with the caller and the callee based on the identified object of the found method. In case of a call to an overloaded private method, the Java compiler (and consequently a call graph) can distinguish which private method is called, by looking at the caller's argument list. In general, the Java compiler can resolve method overloading at compile time.

If there is a call to a non-private method at runtime, dynamic analysis intercepts such calls and resolves dynamic dispatches in the call graph by identifying the execution path in the call graph and capturing object identification information.

5.4.4.2.1 Verifying correctness of static analysis

In this section, we examine the correctness of the procedure/algorithm described in the previous section based on three case studies: OE, calculator and Weka. We use three test cases already mentioned in Table 1: OE_TC1b (input= 200), Calculate (1+1), Weka_TC5 (a test case from Weka_TestSuite). Weka_TC5 lists information for the specific package from the Weka server, which was locally cached as the network was off at the time of experiment. We slightly modified OE_TC1 into OE_TC1b by adding a few method calls to test our approach. Compared to OE_TC1, OE_TC1b is modified by adding two calls to static methods, one call to a private method, and four calls to instance methods. Overall, as shown in Table 12, new modifications for test case OE_TC1b resulted in seven more calls compared to the NMC_{SUS} for OE_TC1 in Table 1. We showed the number of method calls for each test case based on call types in Table 12. We choose these three test cases since the NMC_{SUS} is small enough to study static and dynamic effects by skimming call graph and trace information.

Case study	Test case	Instance methods	Private methods	Const.	Static
OE	OE_TC1b	997+4	1	3	2
	OE_TC2	3,999,997	0	3	0
Calculator	Calculator	45	0	13	55
Weka	Weka_TC5	164	12	15	44
	Weka_TC4	3,514,605	7,035	60,431	411,628

Table 12 Number of method calls based on their type for case studies

We compared the combined static and dynamic analysis approaches mentioned in the previous section with Light, whereby for each test case, call graph information, generated based on the VTA algorithm, is merged with trace information generated based on a modified instrumentation. Then, trace information is merged with call graph to be compared with the trace from Light.

The comparison is successful if both sets (i.e., merged trace-call graph and Light trace) show equivalent information. For each case study, we produce VTA-based call graphs using the *Spark* package from the *Soot*[54] library [209], we produce traces based on a modified instrumentation (mentioned in the previous section), and merged them together based on the order of method calls. Recall that the modified instrumentation is a Light instrumentation where we do not intercept calls to static methods, to private methods and to constructors, but we intercept calls to instance methods and object creations.

For each case study, first we look at the trace to remove those object instances which have not been initialized. Then, we start to traverse the call graph, each time there is a dynamic dispatch (instance method calls), the trace is used to resolve the call and identify correct path, each time there is a static dispatch (static, private and constructor

[54] https://github.com/Sable/soot

calls), the trace is used to resolve object identification. Once the trace information is used to resolve dynamic dispatch and object identification in the call graph, we compare the updated call graph data with the trace from Light.

Our analysis shows the information in the resolved trace/call graph is equal to the information in the trace from Light. Therefore, the suggested hybrid approach can produce Light information in a sound way.

5.4.4.3 Combining optimizations on information being collected

We conduct an experiment in this section to understand the amount of overhead reduction when we combine suggested optimizations in this section to answer RQ1 for the fourth characteristic.

5.4.4.3.1 Experiment design

As we mentioned before, we developed a new version of the Light instrumentation, which we refer to as Partially Optimized 6 (PO6), whereby: we only intercept calls to non-private instance method in `MethodAspect`, and object instance creation in `IdentifierAspect`. The other advices in `MethodAspect` are disabled. In addition, we combined the couple of optimizations mentioned in Section 5.4.4.1 by not capturing: a) the callee's class name and b) the caller's object identifier in case a caller is a static method. We examine PO6 based on OE and Weka to measure the overhead reduction.

5.4.4.3.2 Result

Table 13 shows execution time and trace size based on optimizations on information being collected. The amount of overhead reduction in a third column (PO6)

for OE_TC2 is minor (only 6%) since there are very few static dispatches (only three calls to constructor). However, this optimization affected Weka_TC4 by reducing both overhead and trace size by nearly a fifth since, as shown in Table 12, more static dispatches are occurring in Weka (number of static dispatches for Weka_TC4 is equal to 479,094). Overall, the suggested optimizations will reduce the amount of information being collected and consequently the overhead in of the dynamic analysis.

Case study	Light		PO6	
	time (sec)	size (MB)	time (sec)	size (MB)
OE_TC2	15.44 (100%)	989.6 (100%)	14.53 (-6%)	953.6 (-4%)
Weka_TC4	17.51 (100%)	1250 (100%)	14.24 (-19%)	1000 (-20%)

Table 13 Average execution times (in seconds) and trace size for Light and PO6

5.5 Optimized dynamic analysis

Considering the improvements made in Sections 5.4.1, 5.4.2, 5.4.3 and 5.4.4, we conduct an experiment by merging optimizations of four characteristics together to understand the overall amount of overhead reduction for RQ2. Once we understood the best optimized solution, we further analyzed overhead with more test cases.

5.5.1 Experiment design

We develop new versions of the Light instrumentation, which we refer to as Optimized Light 1, Optimized Light 2 and Optimized Light 3, where we combined optimizations of collecting, encoding, and logging information and collected information all together. More specifically, Optimized Light 1 instrumentation combines optimizations in PO1, PO2 and PO4; Optimized Light 2 instrumentation combines optimizations in PO1, PO3 and PO5; Optimized Light 3 instrumentation combines

optimizations in PO1, PO3, PO5, PO6. We summarized optimizations for each of previous experiments in Table 14.

Experiment	Type of optimization
PO1	• replace `String` instead of `LinkedList` for local variables in advices and for parameters to calls to the logger • use no concatenation in `String` variables • replace helper methods either by using the AspectJ API or inlining where necessary
PO2	• replace `String` instead of `LinkedList` for local variables in advices and for parameters to calls to the logger • use the US-ASCII character encoding instead of UTF-8 encoding • use raw data instead of the textual format used in Light • replace a generic logger with a specialized logger where the logger invocation API is optimized based on the type advice
PO3	• replace `String` instead of `LinkedList` for local variables in advices and for parameters to calls to the logger • use the US-ASCII character encoding instead of UTF-8 encoding • use raw data instead of the textual format used in Light • replace a generic logger with a specialized logger where the logger invocation API is optimized based on the type advice • encoding class names and method names to integer values
PO4	• replace the generic logger with a blocking TCP logger
PO5	• replace the generic logger with a non-blocking TCP logger
PO6	• only intercepts calls to non-private instance method in `MethodAspect`, and object instance creation in `IdentifierAspect` • not capture the callee's class name • not capture the caller's object identifier in cases where a caller is a static method

Table 14 List of optimizations for each experiment

We did not keep many of the optimizations we mentioned earlier due to increased overhead (e.g., using the Java reflection API instead the AspectJ API), or no overhead reduction (e.g., `Hashmap` to uniquely identify objects). Different combinations of optimizations will disclose possible trade offs between trace size, computational overhead and rate of sending logs to the server.

We use the Linux `time` command to measure the execution time for OE and Weka case studies.

5.5.1.1 Results

Table 15 shows the execution times and trace size for the Light and optimized Light instrumentations. Similarly to the experiment for optimizations on logging information (Section 5.4.3.2), we report on two times: $time_{SUS}$ and $time_{log}$. The $time_{SUS}$ in the second and fifth column, which shows that both optimized Light 1, 2 and 3 instrumentations reduced overhead at least by 67% in our case studies. However, in the case of optimized Light 1 instrumentation, as we mentioned in Section 5.4.3.2, this reduction comes with the drawback of long waiting times once the execution of the SUS ends ($time_{log}$ in the third column).

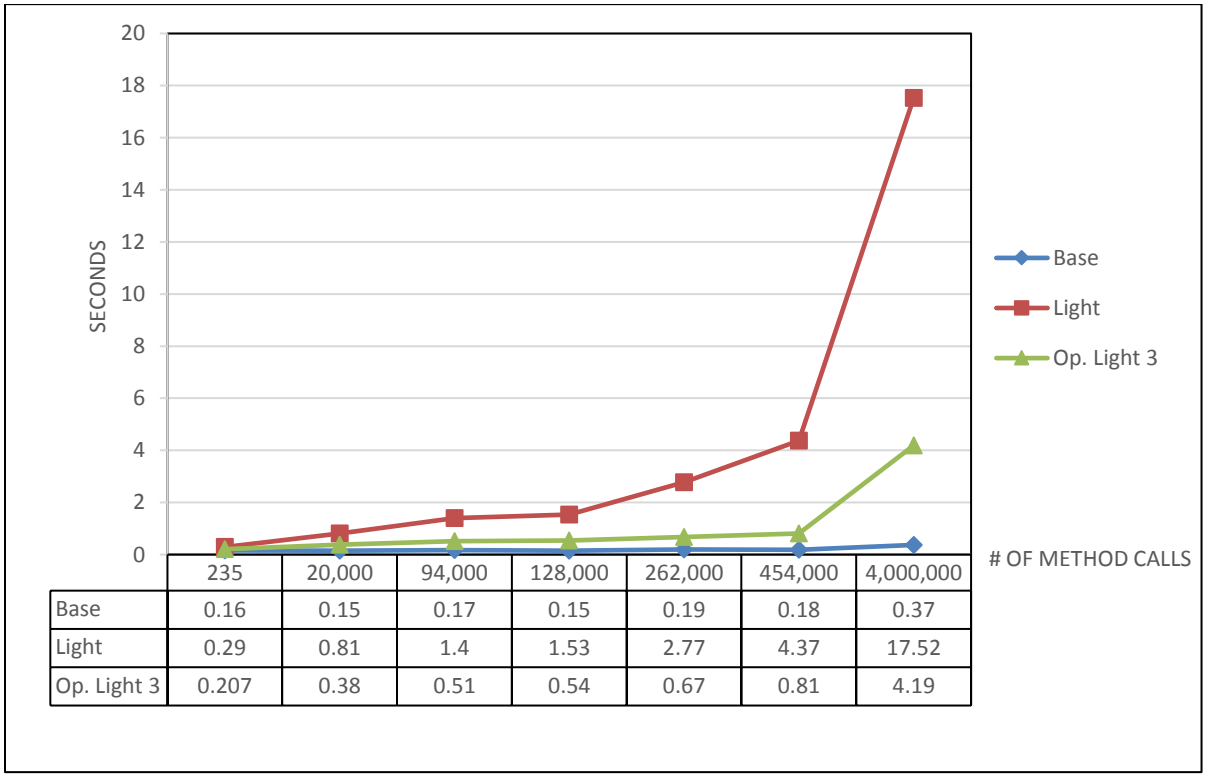
Case study	OE_TC2			Weka_TC4		
	$time_{SUS}$ (sec)	$time_{log}$(sec)	size(MB)	$time_{SUS}$(sec)	$time_{log}$(sec)	size(MB)
Light	15.44 (100%)	N/A	989.6 (100%)	17.51 (100%)	N/A	1250 (100%)
Optimized Light 1	2.34 (-85%)	143.99	241.6 (-75%)	4.37 (-75%)	145.09	460 (-63%)
Optimized Light 2	3.08 (-80%)	2.73	67.2 (-93%)	5.789 (-67%)	1.45	102.7 (-92%)
Optimized Light 3	2.91 (-82%)	2.94	49.6 (-95%)	4.18 (-77%)	2.23	71.5 (-94%)

Table 15 Average execution times as well as trace size for Light and Optimized Light instrumentations

In addition, the fourth and seventh columns in Table 15 show that the optimized Light 1, 2 and 3 instrumentations minimize the trace size by at least 60, 92 and 94 percent

respectively in our case studies. The greater reduction in OE compared to Weka in Optimized Light 1 is due to longer class names in Weka.

Among optimized solutions, Optimized Light 3 shows the best performance since both $time_{log}$ and $time_{SUS}$ are low in this optimization. Fig. 29 shows additional overhead evaluation based on Weka test cases (see Section 4.3.1 for test cases' description). Test cases in Fig. 29 are sorted based on their NMC_{SUS} (horizontal axis). The vertical axis in this figure shows $time_{SUS}$.



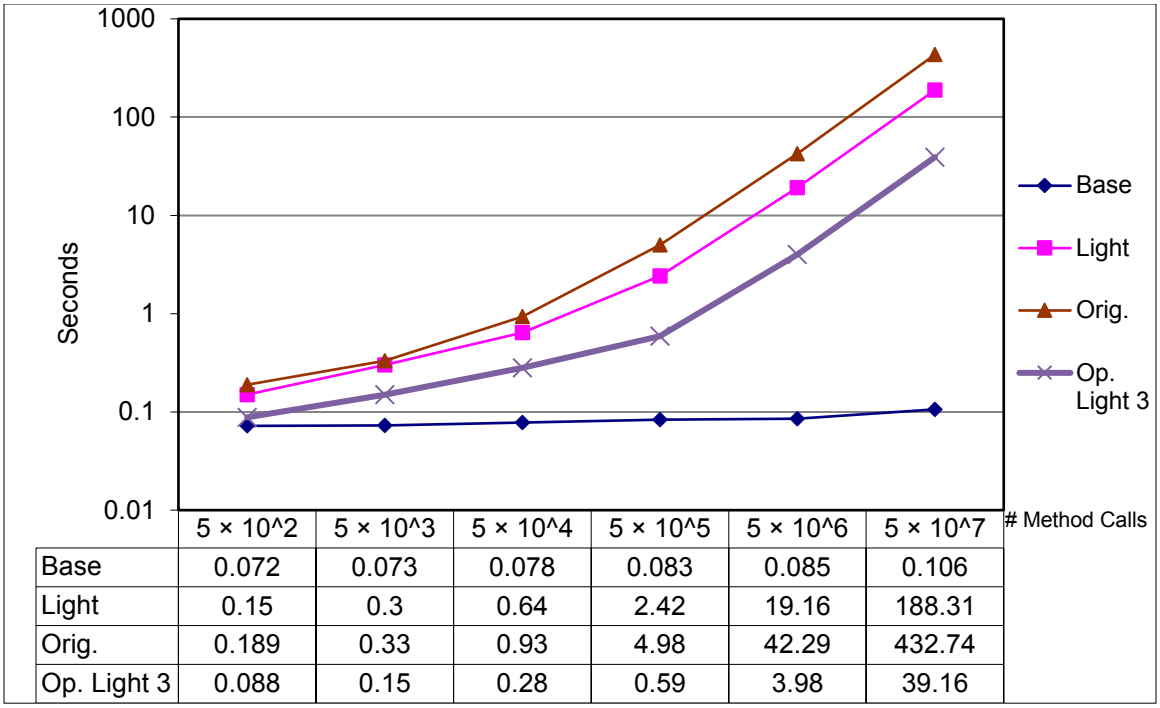
	235	20,000	94,000	128,000	262,000	454,000	4,000,000
Base	0.16	0.15	0.17	0.15	0.19	0.18	0.37
Light	0.29	0.81	1.4	1.53	2.77	4.37	17.52
Op. Light 3	0.207	0.38	0.51	0.54	0.67	0.81	4.19

Fig. 29 Overhead reduction for Weka test cases

Fig. 29 shows the pace of overhead increase is much slower in Optimized Light 3 as the NMCSUS increases.

We also sorted additional OE test cases based on their NMC_{SUS} in Fig. 30 (horizontal axis) to analyze the performance of Optimized Light 3. The vertical axis shows $time_{SUS}$ based on a logarithmic scale for each test case. We also added the

execution time for Leduc's Original instrumentation (the same as what is reported in Fig. 17) to Fig. 30 to be able to compare three instrumentations reported in this thesis (i.e., Leduc's, Kolbah's, and Optimized Light 3). Looking at execution times in Fig. 30, similar to Fig. 29, the pace of overhead increase is slower in Optimized Light 3 compared to Original and Light instrumentations.



	5 × 10^2	5 × 10^3	5 × 10^4	5 × 10^5	5 × 10^6	5 × 10^7
Base	0.072	0.073	0.078	0.083	0.085	0.106
Light	0.15	0.3	0.64	2.42	19.16	188.31
Orig.	0.189	0.33	0.93	4.98	42.29	432.74
Op. Light 3	0.088	0.15	0.28	0.59	3.98	39.16

Fig. 30 Overhead reduction for OE test cases

5.6 Frequency of probe effect observation

As we discussed in Section 2.1.4, our optimizations only minimize the probe effect; they do not remove the probe effect. The overhead minimization reduces the potential of behaviour deviation in the observed software, especially in a real-time system (RTS) and a distributed real-time system (DRTS). We conduct a new experiment based on the Producer-Consumer case study to understand to what extent the Optimized Light instrumentations reduce the probe effect compared to the Light instrumentation.

5.6.1 Experiment design

In order to answer RQ3, we add other versions of the Producer-Consumer to the experiment we discussed in Section 4.8: the so-called optimized Light 1, Optimized Light 2, and Optimized Light 3 instrumentations, which we refer to as Optimized 1, Optimized 2 and Optimized 3 respectively. Note that as we use the same case study, configurations, test cases, and measurement as we discussed in Section 4.8.1, we do not reiterate them in this section.

5.6.2 Results

Table 16 contains the information for Light and Base instrumentations that we already presented in Table 3. Again we do not explain the terminology we used in the table as it was discussed in Section 4.8.2. Delay values in the Optimized 1 column in Table 16 show two different measures: the first (left) column indicates the average consumption time when the aspects do not have to wait for access to the logging buffer (i.e., the buffer is not full); the subscript indicates the number of queue accesses the consumer performs while this condition holds; the second (right) column indicates the average consumption time when the logging buffer gets full (because of low network transfer rate) and the aspects have to wait for access to the logging buffer. However, we did not observe similar situation in logging buffer for Optimized 2 and Optimized 3. The producer is negligibly affected by the instrumentation as it executes only one method call (to the queue) in case the queue is not full and therefore only one call is instrumented: its execution times are therefore not reported.

	Base		Light		Optimized 1			Optimized 2		Optimized 3	
	Rate	***Consumer delay***	***Rate***	***Consumer delay***	***Rate***	***Consumer delay***		***Rate***	***Consumer delay***	***Rate***	***Consumer delay***
TC 1	Greater	$1.5*10^{-5}$	Greater	0.229	Greater	0.07_{165}	3.5	Greater	0.078	Greater	0.071
TC 2	Greater	$4*10^{-5}$	Equal	1.008	Greater	0.32_{36}	14.7	Greater	0.30	Greater	0.29
TC 3	Greater	$8.5*10^{-5}$	Smaller	2.27	Greater	0.71_{14}	35	Greater	0.613	Greater	0.58
TC 4	Greater	$77*10^{-5}$	Smaller	23.18	Smaller	5.7_{1}	355	Smaller	6.13	Smaller	5.65

Table 16 Effect of instrumentation overhead on the producer-consumer

Table 16 shows that with our optimization ("Optimized 1", "Optimized 2", and "Optimized 3" columns), only TC4 shows a change of behaviour, indicating that we reduce the risk of behaviour change due to instrumentation. The amount of delay (second column) for the Optimized 1 solution is worse than Light for all four test cases as the instrumentation buffer gets full. The Optimized 1 logger is not fast enough at removing log items from the instrumentation buffer and sending them over the network because of the low throughput of the blocking TCP network protocol. Yet, non-blocking TCP logger with a better network throughput and smaller trace size in Optimized 2 and Optimized 3 solved this bottleneck. However, the overhead reduction for TC4 instrumenting with Optimized 2 and 3 were not large enough to stop behaviour change in Producer-Consumer. Our analysis shows the consumer delay reduced to about 1.45 seconds for both Optimized 2 and 3 when we disabled calls to the remote server. Therefore, establishing non-blocking TCP connection, compressing packets and transmitting over network account for the large part in consumer delay. The number of consumer executions in the first column of optimized 1 (subscript) declines from TC1 to TC4 as the length of consumer executions grows.

5.7 Threats to Validity

We look at different threats [200] to evaluate the validity of our results. With regards to *conclusion* validity, the standard deviation of all samples was below 1%, (we did not report on those values). We compared samples with a Student t-test, and differences are always very statistically significant at α=0.05 (except reported statistically insignificant results). We confirmed this with the Mann–Whitney *U* non-parametric test. We limited threats to *internal* validity by using practices such as: choosing precise and robust measures, choosing test cases large enough to report on considerable numbers of events, running each experiment 100 times, allocating all resources to experiments by not running any application on both the client and the server computers. We minimized threats to *construct* validity by choosing case studies that reflect on our proposed problem, by evaluating alternative designs in experiments (e.g., alternative remote and local loggers). We limited threats to *external* validity by presenting an experimental protocol and optimization strategies that are pertinent to other instrumentation strategies (e.g., fewer probe insertion points, more efficient probes, proper choice of data-type).

5.8 Conclusion

The Light (hybrid) instrumentation approach imposes execution time overhead since it has not been designed with performance in mind but rather with maintainability, and understandability objectives (i.e., good design principles). This results in runtime overhead that may become unacceptable if one wants to reverse engineer object interactions from multi-threaded software. In Section 4.9, we precisely quantified possible sources of overhead. This gave us actionable findings that helped us characterize the Light instrumentation and optimize the instrumentation to reduce execution time

overhead. We implemented many of the suggested optimizations for each characteristic and experimented on two large case study systems: a real one and a synthetic one. In addition, we evaluated to what extent execution time overhead can indeed change the observed behaviour when one reverse-engineers a multi-threaded software.

Results indicate that our optimizations can reduce overhead at least by 67% compared to the Light instrumentation approach. We believe our findings can be used more broadly to reduce instrumentation overhead in other contexts. We believe, our results indicate that we may be approaching the limits of overhead reduction in a hybrid-based instrumentation with current technologies.

6 Conclusion

In this chapter, first we summarize the result of the research reported in this dissertation (Section 6.1). Next, we suggest a set of recommendations for those who want to develop an efficient instrumentation or optimize their current instrumentation (Section 6.2). Finally, we explain future works for our research (Section 6.3).

6.1 Contribution

In this work, we focused on efficient instrumentation based on reverse engineering code to produce method calls to increase the accuracy of observed interaction for a software system. We split this goal into a number of contributions, among which we accomplished the following:

Contribution 1: We confirmed through extensive experimentation that the promising hybrid approach proposed by Kolbah can indeed reduce the overhead for large scale software systems compared to the overhead of the similar, purely dynamic approach with the same amount of collected information (overhead reduction by half), but that the existing hybrid approach was nevertheless imposing a large overhead (in the order of nearly fifty times longer execution) compared to the non-instrumented system, which can be detrimental when reverse-engineering a multi-threaded system (i.e., instrumentation may change the behaviour to the extent the reverse engineered behaviour is not the right one).

Contribution 2: We systematically measured the amount of overhead imposed by each component of Kolbah's instrumentation (i.e., Light) by designing different experiments. This gave us clues about heavy contributing components to overhead that would need to be optimized. Unsurprisingly, saving execution trace information to disk

was the main contributor to execution overhead. Collecting trace information (data collection in dynamic analysis) was a significant contributor too. Results confirmed our decision to use AspectJ as it showed to have marginal overhead impact compared to other components of Light instrumentation and the overall overhead.

Contribution 3: We suggested a systematic process for studying overhead and optimizing instrumentation techniques based on different characteristics of a typical instrumentation: e.g., the amount of data being collected, the actual collecting process, the logging mechanism. For each characteristic we discussed a range of optimization activities and experimented with each one of them. Depending on the characteristics of the instrumentation technique and the SUS, one has to consider a number of trade-offs when selecting an optimization activity. Our experiments show that generally data logging and data encoding characteristics are the best candidates to optimize instrumentation. Depending on the type of trace data, moving from a dynamic data collection to the static data collection is the next best candidate to optimize instrumentations.

Contribution 4: We performed the abovementioned optimization process on the Light, hybrid instrumentation. Our main observations include that, despite our initial intuition, optimizing the generic logger to the customized local logger (i.e., ChacheLogger) greatly improved the performance (by more than a half) and using existing logging frameworks (e.g., Lo4J) proved to be unsuccessful. After analyzing many of the suggested optimization activities (supported with empirical data), we combined the chosen optimizations on an optimized Light instrumentation solution. This version reduces the overhead of the Light instrumentation at least by three quarters based

on our case studies. Components of our optimized hybrid instrumentation can be used in other instrumentations such as local and remote loggers, and aspects.

Contribution 5: We used case studies of varying sizes, including industry size software and a synthetic case study that is used to mimic large software. Even though we targeted large software systems in our work, our results show that performing simulations with this synthetic software is in fact helpful and can lead to accurate performance analysis results on large software systems.

Those contributions, although obtained in the specific context of a hybrid technique, based on AspectJ, to reverse engineer object interactions, should apply more broadly to other techniques to reverse engineer runtime details, neither necessarily hybrid (e.g., dynamic) nor based on AspectJ. We believe our empirical results can be seen as actionable findings that others can build on when devising reverse-engineering technology.

6.2 Recommendation

We suggest a number of recommendations for optimizing current instrumentation solutions or developing an efficient instrumentation.

1. For offline instrumentation, it is recommended to use non-blocking TCP remote server. It is possible to use efficient local logger on the same machine (e.g., CacheLogger) if the imposed overhead does not prevent the instrumented SUS from meeting deadlines.
2. If the SUS contains many loops and conditions in its classes, using of compaction algorithms (e.g., Snappy) to minimize trace size boosts the performance of instrumentation.

3. Before collecting trace in instrumentation, one should have a good knowledge of static analysis tools and constructs and carefully analyze the trace model to use static analysis as much as possible and establish one-to-one relation between the trace and static anlysis artefact to merge them together.
4. If the trace model contains system level information (e.g., pid<#> or cpuinfo) without any need to capture userspace/application level information, it is recommended to capture such information at the operating system level based on technologies such as Perf Event or DTrace. Otherwise, one should access Java application level information: a) If the development team has access to experienced developers with knowledge of byte-code manipulation technologies (e.g., ASM or BCEL), it is recommended to use low level byte-code manipulation technologies, b) otherwise, it is recommended to use AspectJ to capture trace data. When developing instrumentation tool, one need to have a good knowledge of the instrumentation language to develop instrumentation according to the best practices in that instrumentation language (e.g., use of efficient data structure).

6.3 Future work

There are a number of future directions for this research:

1. The next logical step for the optimized hybrid approach is to capture real-time and distributed features of a Java system by adding proper aspects to the instrumentation. In fact, Leduc [56] devised an approach to do that for Java real-time software that relies on the RMI distribution middleware. We have initial experiments showing that merging her solution to the optimized hybrid approach we presented is feasible. We add an aspect,

thread aspect, to capture a thread ID and time stamps by intercepting all calls to a thread (through a `before` advice) and a thread execution itself (through an `around` advice). We add another aspect, node aspect, to intercept execution of each node on RMI middleware to capture node ID (through a `before` advice). We do not need to capture any time stamp for nodes since thread time stamps and the order of logging on log server would be enough for synchronization, and using Leduc's solution, RMI nodes exchange times stamps (no need for a global clock). In case RMI is not used, mechanisms exist in the literature that are complementary as we discussed in the related work chapter, though their combination with our approach remains to be studied.

2. As we mentioned in the literature review, it is possible to further investigate the overhead reduction due to instrumentation mechanisms (i.e., collecting information) by using alternative instrumentation mechanisms (e.g., DiSL, Spoon, or byte-code manipulating libraries such as ASM) other than AspectJ. However, even in case of successful implementation of a hybrid approach with those alternative instrumentation mechanisms, the performance improvement would be, we believe, marginal. Indeed, as we showed in our work, the instrumentation mechanism is not the major cause of the overhead. In addition, the potential improvements on technologic platforms (newer versions of the AspectJ compiler, the Java compiler or the JVM) over time due to wide usage and constant improvement can make our instrumentation running more efficiently compared to less used technologies.

3. From a software comprehension perspective, it would be possible to apply trace abstraction techniques on traces generated from our approach (from the trace analysis research field) to provide higher views of a system (e.g., architectural view). In addition,

since generated scenarios conform to the UML meta-model, a stored scenario (in the form of XMI file) contains all the required information to visualize sequence diagrams. Thus, we can transform an XMI file to other file formats (e.g., MCR) to use visualization capabilities provided by different program comprehension tools such as PragmaDev Tracer [210].

4. As we mentioned in Section 5.4.2, we can investigate further overhead reduction based on encoding information by changing the trace representation from a tree to a more condensed representation (e.g., Directed Acyclic Graph format [55, 182]) before logging on disk. This way less information would be stored on disk resulting in less overhead due to recording on disk. However a compromise has to be found between the overhead of the condensing algorithm and the overhead of storing on disk.

5. The dynamic analysis part is generally unable to produce a sound scenario diagram unless the test suite provides adequate coverage for the whole program. (Recall program feature versus test case feature we discussed in Chapter 1.) Along the same line of work as Tonella and Potrich [138], we can provide a complete scenario diagram by feeding the instrumented SUS with a test suite with adequate coverage. To achieve this level of coverage we can use different coverage criteria or devise new criteria based on static control flow graph or call graph.

6. Of course the approach we presented should be evaluated on more case studies, of various sizes (e.g., using DaCapo[55] and SPECjvm2008[56] benchmarks).

7. Since AOP solutions exist for other programming languages than Java (we mentioned a few earlier in the related work), future can also look into adapting our

[55] http://dacapobench.org/
[56] https://www.spec.org/jvm2008/

solution to other programming languages and AOP solutions. For instance, our results show a successful implementation of the Light hybrid approach for C++ based on AspectC++ in Appendix D. However, extending this implementation based on the optimized hybrid approach possibly to support real-time and distributed software systems is left for future efforts.

References

1. H.G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358-366, 1953.
2. M.D. Ernst, "Static and Dynamic Analysis: Synergy and Duality," *Proc. of ICSE Workshop on Dynamic Analysis*, pp. 24-27, 2003.
3. Y. Guéhéneuc, "A Systematic Study of Uml Class Diagram Constituents for Their Abstract and Precise Recovery," *Proc. of Asia-Pacific Software Engineering Conference*, IEEE, pp. 265-274, 2004.
4. Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering Binary Class Relationships: Putting Icing on the Uml Cake," *Proc. of conference on Object-oriented programming, systems, languages, and applications*, ACM, pp. 301-314, 2004.
5. S. Horwitz, "Precise Flow-Insensitive May-Alias Analysis Is Np-Hard," *ACM Trans. Programming Languages Systems*, vol. 19, no. 1, pp. 1-6, 1997.
6. R. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations*, The Ibm Research Symposia Series, Springer, pp. 85-103, 1972.
7. W. Landi, "Undecidability of Static Analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323-337, 1992.
8. G. Ramalingam, "The Undecidability of Aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467-1471, 1994.
9. T. Reps, "Undecidability of Context-Sensitive Data-Dependence Analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162-186, 2000.
10. P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.
11. A. Rountev, S. Kagan and J. Sawin, "Coverage Criteria for Testing of Object Interactions in Sequence Diagrams," *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science 3442, Springer, pp. 289-304, 2005.
12. A. Andrews, R. France, S. Ghosh and G. Craig, "Test Adequacy Criteria for Uml Design Models," *Software Testing, Verification and Reliability*, vol. 13, no. 2, pp. 95-127, 2003.
13. P. Chevalley and P. Thevenod-Fosse, "Automated Generation of Statistical Test Cases from Uml State Diagrams," *Proc. of Computer Software and Applications Conference*, pp. 205-214, 2001.
14. J. Offutt and A. Abdurazik, "Generating Tests from Uml Specifications," *Uml'99 — the Unified Modeling Language*, Lecture Notes in Computer Science 1723, Springer pp. 416-429, 1999.

15. C. Ackermann, M. Lindvall and R. Cleaveland, “Recovering Views of Inter-System Interaction Behaviors,” *Proc. of IEEE Working Conference on Reverse Engineering*, pp. 53-61, 2009.

16. S. Stevenson, “Investigating Software Reconnaissance as a Technique to Support Feature Location and Program Analysis Tasks Using Sequence Diagrams,” CS, University of Victoria, 2013.

17. D.-P. Nguyen, C.-T. Luu, A.-H. Truong and N. Radics, “Verifying Implementation of Uml Sequence Diagrams Using Java Pathfinder,” *Proc. of IEEE Knowledge and Systems Engineering*, pp. 194-200, 2010.

18. Z. Zhou, L. Wang, Z. Cui, X. Chen and J. Zhao, “Jasmine: A Tool for Model-Driven Runtime Verification with Uml Behavioral Models,” *Proc. of IEEE High Assurance Systems Engineering Symposium*, pp. 487-490, 2008.

19. K. Havelund and G. Roşu, “An Overview of the Runtime Verification Tool Java Pathexplorer,” *Formal Methods in System Design*, vol. 24, no. 2, pp. 189-215, 2004.

20. M.K. Ganai, “Dynamic Livelock Analysis of Multi-Threaded Programs,” *Runtime Verification*, Lecture Notes in Computer Science 7687, Springer, pp. 3-18, 2013.

21. C. Nan, I. Eusgeld and W. Kroger, “Analyzing Vulnerabilities between Scada System and Suc Due to Interdependencies,” *Reliability Engineering & System Safety*, vol. 113, no. Supplement C, pp. 76-93, 2013.

22. R. Gronmo and B. Moller-Pedersen, “From Uml 2 Sequence Diagrams to State Machines by Graph Transformation,” *Journal of Object Technology*, vol. 10, no. 8, pp. 1-22, 2011.

23. D. Petriu and Y. Sun, “Consistent Behaviour Representation in Activity and Sequence Diagrams,” *Uml 2000 —the Unified Modeling Language*, Lecture Notes in Computer Science 1939, Springer, pp. 369-382, 2000.

24. C. Artho and A. Biere, “Combined Static and Dynamic Analysis,” *Electronic Notes in Theoretical Computer Science*, vol. 131, no. 0, pp. 3-14, 2005.

25. B. Kolbah, “Reverse Engineering of Java Programs through Static and Dynamic Analysis to Generate Scenario Diagrams,” ECE, Carleton University, Ottawa, 2011.

26. B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen and R. Koschke, “A Systematic Survey of Program Comprehension through Dynamic Analysis,” *IEEE Transaction on Software Engineering*, vol. 35, no. 5, pp. 684-702, 2009.

27. E.J. Chikofsky and J.H. Cross, “Reverse Engineering and Design Recovery: A Taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13-17, 1990.

28. G. Canfora, M.D. Penta and L. Cerulo, “Achievements and Challenges in Software Reverse Engineering,” *Communications of the ACM*, vol. 54, no. 4, pp. 142-151, 2011.

29. P. Grubb and A.A. Takang, *Software Maintenance: Concepts and Practice*, World Scientific, 2003.

30. D. Abramson, I. Foster, J. Michalakes and R. Sosic, "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Communication of the ACM*, vol. 39, no. 11, pp. 69-77, 1996.

31. C. Zamfir and G. Candea, "Execution Synthesis: A Technique for Automated Software Debugging," *Proc. of ACM European conference on Computer systems*, pp. 321-334, 2010.

32. M. Ducasse, "A Pragmatic Survey of Automated Debugging," *Automated and Algorithmic Debugging*, Lecture Notes in Computer Science 749, Springer Berlin Heidelberg, pp. 1-15, 1993.

33. J. Silva, "A Survey on Algorithmic Debugging Strategies," *Advances in Engineering Software*, vol. 42, no. 11, pp. 976-991, 2011.

34. D.G. Waddington, N. Roy and D.C. Schmidt, "Dynamic Analysis and Profiling of Multithreaded Systems," *Designing Software-Intensive Systems: Methods and Principles*, IGI Global, pp. 290-334, 2009.

35. T. Reps, T. Ball, M. Das and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *Software Engineering — Esec/Fse'97*, Lecture Notes in Computer Science 1301, Springer, pp. 432-449, 1997.

36. R. Patel and A. Rajawat, "A Survey of Embedded Software Profiling Methodologies," *International Journal of Embedded Systems and Applications*, vol. 1, no. 2, 2011.

37. M.-C. Hsueh, T.K. Tsai and R.K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75-82, 1997.

38. Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, 2011.

39. A. Zaidman, "Scalability Solutions for Program Comprehension through Dynamic Analysis," University of Antwerp, 2006.

40. B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Trans. on Software Engineering*, vol. 35, no. 5, pp. 684-702, 2009.

41. N. Delgado, A.Q. Gates and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE Trans. on Software Engineering*, vol. 30 no. 12, pp. 859-872, 2004

42. A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," *Proc. of 2007 Future of Software Engineering*, IEEE Computer Society, pp. 85-103, 2007.

43. A. Hamou-Lhadj and T.C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques,” *Proc. of of the Centre for Advanced Studies on Collaborative research*, IBM Press, pp. 42-55, 2004.

44. A. Hamou-Lhadj and T.C. Lethbridge, “Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System,” *Proc. of International Conference on Program Comprehension*, IEEE pp. 181-190, 2006.

45. A. Hamou-Lhadj, E. Braun, D. Amyot and T. Lethbridge, “Recovering Behavioral Design Models from Execution Traces,” *Proc. of European Conference on Software Maintenance and Reengineering*, pp. 112-121, 2005.

46. P. Dugerdil and J. Repond, “Automatic Generation of Abstract Views for Legacy Software Comprehension,” *Proc. of India Software Engineering Conference*, ACM pp. 23-32, 2010.

47. J. Bohnet, M. Koeleman and J. Doellner, “Visualizing Massively Pruned Execution Traces to Facilitate Trace Exploration,” *Proc. of IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 57-64, 2009.

48. H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh and A. Shafiee, “Stratified Sampling of Execution Traces: Execution Phases Serving as Strata,” *Science of Computer Programming*, vol. 78, no. 8, pp. 1099-1118, 2013.

49. A. Zaidman and S. Demeyer, “Managing Trace Data Volume through a Heuristical Clustering Process Based on Event Execution Frequency,” *Proc. of European Conference on Software Maintenance and Reengineering*, pp. 329-338, 2004.

50. A. Hamou-Lhadj and T.C. Lethbridge, “Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools,” *Proc. of IEEE International Conference on Engineering of Complex Computer Systems*, pp. 559-568, 2005.

51. H. Pirzadeh and A. Hamou-Lhadj, “A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension,” *Proc. of IEEE International Conference on Engineering of Complex Computer Systems*, pp. 221-230, 2011.

52. B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen and J.J. van Wijk, “Execution Trace Analysis through Massive Sequence and Circular Bundle Views,” *Journal of Systems and Software*, vol. 81, no. 12, pp. 2252-2268, 2008.

53. B. Cornelissen, A. Zaidman, A. van Deursen and B. van Rompaey, “Trace Visualization for Program Comprehension: A Controlled Experiment,” *Proc. of IEEE International Conference on Program Comprehension*, pp. 100-109, 2009.

54. S.P. Reiss and M. Renieris, “Encoding Program Executions,” *Proc. of ACM/IEEE International Conference on Software Engineering*, pp. 221-230, 2001.

55. A. Hamou-Lhadj and T. Lethbridge, “A Metamodel for the Compact but Lossless Exchange of Execution Traces,” *Software & Systems Modeling*, vol. 11, no. 1, pp. 77-98, 2012.

56. L.C. Briand, Y. Labiche and J. Leduc, "Towards the Reverse Engineering of Uml Sequence Diagrams for Distributed Java Software," *IEEE Trans. on Software Engineering*, vol. 32, no. 9, pp. 642-663, 2006.

57. A. Knüpfer, R. Brendel, H. Brunst, H. Mix and W.E. Nagel, "Introducing the Open Trace Format (Otf)," *Lecture Notes in Computer Science*3992, Springer, pp. 526-533, 2006.

58. P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhotak, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble and M. Verbaere, "Aspects for Trace Monitoring," *Formal Approaches to Software Testing and Runtime Verification*, Lecture Notes in Computer Science 4262, Springer Berlin Heidelberg, pp. 20-39, 2006.

59. D.J. Pearce, M. Webster, R. Berry and P.H.J. Kelly, "Profiling with Aspectj," *Software: Practice and Experience*, vol. 37, no. 7, pp. 747-777, 2007.

60. R. Laddad, *Aspectj in Action: Enterprise Aop with Spring Applications*, Manning Publications Co., p. 568, 2010.

61. M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger and E. Kapsammer, "A Survey on Uml-Based Aspect-Oriented Design Modeling," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1-33, 2011.

62. O. Aldawud, T. Elrad and A. Bader, "Uml Profile for Aspect-Oriented Software Development," *Proc. of International Workshop on Aspect-Oriented Modeling*, 2003.

63. R.M. Golbeck, S. Davis, I. Naseer, I. Ostrovsky and G. Kiczales, "Lightweight Virtual Machine Support for Aspectj," *Proc. of Int. Conference on Aspect-oriented software development*, ACM, pp. 180-190, 2008.

64. D. Mahrenholz, O. Spinczyk and W. Schroder-Preikschat, "Program Instrumentation for Debugging and Monitoring with Aspectc++," *Proc. of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* pp. 249-256, 2002.

65. O. Spinczyk and D. Lohmann, "The Design and Implementation of Aspectc++," *Knowledge-Based Systems*, vol. 20, no. 7, pp. 636-651, 2007.

66. R. Tartler, D. Lohmann, F. Scheler and O. Spinczyk, "Aspectc++: An Integrated Approach for Static and Dynamic Adaptation of System Software," *Knowledge-Based Systems*, vol. 23, no. 7, pp. 704-720, 2010.

67. J.A. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *Computer*, vol. 21, no. 10, pp. 10 - 19, 1988.

68. A. Wellings, "Real-Time Software," *Software Engineering Journal*, vol. 6, no. 3, 1991.

69. A.S. Tanenbaum and M. Van Steen, *Distributed Systems Principles and Paradigms*, Pearson Prentice Hall, 2007.

70. W. Schütz, "Fundamental Issues in Testing Distributed Real-Time Systems, Real-Time Systems," *Real-Time Systems*, vol. 7, no. 2, pp. 129-157, 1994.

71. J. Gait, "A Probe Effect in Concurrent Programs," *Software: Practice and Experience*, vol. 16, no. 3, pp. 225-233, 1986.

72. H. Thane and H. Hansson, "Testing Distributed Real-Time Systems," *Microprocessors and Microsystems*, vol. 24, no. 9, pp. 463-478, 2001.

73. M. Hurfin, N. Plouzeau and M. Raynal, "Erebus: A Debugger for Asynchronous Distributed Computing Systems," *Proc. of Workshop on Future Trends of Distributed Computing Systems*, pp. 93-98, 1992.

74. C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593-622, 1989.

75. H. Garcia-Molina, F. Germano and W.H. Kohler, "Debugging a Distributed Computing System," *IEEE Transactions Software Engineering*, vol. SE-10, no. 2, pp. 210-219, 1984.

76. N. Kumar, B.R. Childers and M.L. Soffa, "Low Overhead Program Monitoring and Profiling," *Proc. of ACM workshop on Program analysis for software tools and engineering*, pp. 28-34, 2005.

77. T. Ball, "Efficiently Counting Program Events with Support for on-Line Queries," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1399-1410, 1994.

78. J.J.P. Tsai, K.Y. Fang, H.Y. Chen and Y. Bi, "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 897-916, 1990.

79. M.H. Alalfi, J.R. Cordy and D. Thomas, "Automated Reverse Engineering of Uml Sequence Diagrams for Dynamic Web Applications," *Proc. of IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 287-294, 2009.

80. A. Zaidman, N. Matthijssen, M.-A. Storey and A. van Deursen, "Understanding Ajax Applications by Connecting Client and Server-Side Execution Traces," *Empirical Software Engineering*, vol. 18, no. 2, pp. 181-218, 2013.

81. Y. Imazeki and S. Takada, "Reverse Engineering of Sequence Diagrams from Framework Based Web Applications," *Proc. of International Conference on Software Engineering and Applications*, pp. 202-209, 2009.

82. C.M. Gail, N. David, G.G. William and S.L. Erica, "An Empirical Study of Static Call Graph Extractors," *ACM Trans. Software Engineering Methodology*, vol. 7, no. 2, pp. 158-191, 1998.

83. E.A. Frances, "Control Flow Analysis," *Proc. of a symposium on Compiler optimization*, ACM, pp. 1-19, 1970.

84. S.M. Steven, *Advanced Compiler Design Implementation*, Morgan Kaufmann, 1997.

85. V. Garousi, L. Briand and Y. Labiche, "Control Flow Analysis of Uml 2.0 Sequence Diagrams," *Model Driven Architecture Ture – Foundations and Applications*, Lecture Notes in Computer Science 3748, Springer, pp. 160-174, 2005.

86. A. Rountev, O. Volgin and M. Reddoch, "Static Control-Flow Analysis for Reverse Engineering of Uml Sequence Diagrams," *Proc. of workshop on Program analysis for software tools and engineering*, pp. 96-102, 2005.

87. M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?," *Proc. of ACM workshop on Program analysis for software tools and engineering*, pp. 54-61, 2001.

88. A. Rountev, A. Milanova and B.G. Ryder, "Points-to Analysis for Java Using Annotated Constraints," *Proc. of ACM SIGPLAN on Object-oriented programming, systems, languages, and applications*, pp. 43-55, 2001.

89. R. Chatterjee, B.G. Ryder and W.A. Landi, "Relevant Context Inference," *Proc. of ACM SIGPLAN symposium on Principles of programming languages*, pp. 133-146, 1999.

90. B. Ryder, "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages," *Compiler Construction*, Lecture Notes in Computer Science 2622, Springer Berlin Heidelberg, pp. 126-137, 2003.

91. D.F. Bacon and P.F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," *Proc. of ACM SIGPLAN on Object-oriented programming, systems, languages, and applications*, ACM, pp. 324-341, 1996.

92. A. Rountev, S. Kagan and M. Gibas, "Static and Dynamic Analysis of Call Chains in Java," *Proc. of ACM SIGSOFT international symposium on Software testing and analysis*, ACM, pp. 1-11, 2004.

93. M. Sagiv, T. Reps and S. Horwitz, "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation," *Theoretical Computer Science*, vol. 167, no. 1–2, pp. 131-170, 1996.

94. T. Reps, S. Horwitz and M. Sagiv, "Precise Interprocedural Dataflow Analysis Via Graph Reachability," *Proc. of ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 49-61, 1995.

95. B. Steensgaard, "Points-to Analysis in Almost Linear Time," *Proc. of ACM SIGPLAN symposium on Principles of programming languages*, pp. 32-41, 1996.

96. D. Grove, G. DeFouw, J. Dean and C. Chambers, "Call Graph Construction in Object-Oriented Languages," *Proc. of ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 108-124, 1997.

97. F. Tip and J. Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," *Proc. of ACM SIGPLAN on Object-oriented programming, systems, languages, and applications*, pp. 281-293, 2000.

98. D. Grove and C. Chambers, "A Framework for Call Graph Construction Algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 6, pp. 685-746, 2001.

99. J. Dean, D. Grove and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," *9th European Conference in Object-Oriented Programming*, Springer Berlin Heidelberg, pp. 77-101, 1995.

100. D.F. Bacon and P.F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," *Proc. of 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, 1996.

101. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon and C. Godin, "Practical Virtual Method Call Resolution for Java," *Proc. of 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, 2000.

102. P. Tonella and A. Potrich, "Reverse Engineering of the Interaction Diagrams from C++ Code," *Proc. of International Conference on Software Maintenance*, pp. 159-168, 2003.

103. A. Rountev and B.H. Connell, "Object Naming Analysis for Reverse-Engineered Sequence Diagrams," *Proc. of International Conference on Software Engineering*, pp. 254-263 2005.

104. D. Callahan, K.D. Cooper, K. Kennedy and L. Torczon, "Interprocedural Constant Propagation," *Proc. of ACM SIGPLAN symposium on Compiler construction*, pp. 152-161, 1986.

105. S. Jagannathan and S. Weeks, "Analyzing Stores and References in a Parallel Symbolic Language," *Proc. of ACM conference on LISP and functional programming*, 294-305, 1994.

106. C. von Praun and T.R. Gross, "Static Conflict Analysis for Multi-Threaded Object-Oriented Programs," *Proc. of ACM SIGPLAN on Programming language design and implementation*, pp. 115 - 128, 2003.

107. S.M. Shatz and W.K. Cheng, "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior," *Journal of Systems and Software*, vol. 8, no. 5, pp. 343-359, 1988.

108. H. Garavel, F. Lang, R. Mateescu and W. Serwe, "Cadp 2010: A Toolbox for the Construction and Analysis of Distributed Processes," *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 6605, Springer Berlin Heidelberg, pp. 372-387, 2011.

109. A. Bouajjani, M. Emmi and G. Parlato, "On Sequentializing Concurrent Programs," *Static Analysis*, Lecture Notes in Computer Science 6887, Springer Berlin Heidelberg, pp. 129-145, 2011.

110. J. Huang and C. Zhang, "An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs," *Static Analysis*, Lecture Notes in Computer Science 6887, Springer Berlin Heidelberg, pp. 163-179, 2011.

111. D. Sereni and O. de Moor, "Static Analysis of Aspects," *Proc. of international conference on Aspect-oriented software development*, pp. 30-39, 2003.

112. J. Zhao, "Control-Flow Analysis and Representation for Aspect-Oriented Programs," *Proc. of International Conference on Quality Software*, pp. 38-48, 2006.

113. M.L. Bernardi and G.A. Di Lucca, "An Interprocedural Aspect Control Flow Graph to Support the Maintenance of Aspect Oriented Systems," *Proc. of IEEE International Conference on Software Maintenance*, pp. 435-444, 2007.

114. G. Xu and A. Rountev, "Ajana: A General Framework for Source-Code-Level Interprocedural Dataflow Analysis of Aspectj Software," *Proc. of International Conference on Aspect-oriented software development*, pp. 36-47, 2008.

115. A. Serebrenik, S. Roubtsov, E. Roubtsova and M. van den Brand, "Reverse Engineering Sequence Diagrams for Enterprise Javabeans with Business Method Interceptors," *Proc. of IEEE Working Conference on Reverse Engineering*, pp. 269-273, 2009.

116. S. Roubtsov, A. Serebrenik, A. Mazoyer and M. van den Brand, "I2sd: Reverse Engineering Sequence Diagrams from Enterprise Java Beans with Interceptors," *Proc. of IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 155-164, 2011.

117. S. Roubtsov, A. Serebrenik, A. Mazoyer, M. G. J. van den Brand and E. Roubtsova, "I2sd: Reverse Engineering Sequence Diagrams from Enterprise Java Beans with Interceptors," *IET Software*, vol. 7, no. 3, pp. 150-166, 2013.

118. M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," 2004.

119. B.A. Malloy and J.F. Power, "Exploiting Uml Dynamic Object Modeling for the Visualization of C++ Programs," *Proc. of ACM Symposium on Software Visualization*, pp. 105-114, 2005.

120. T. Systa, K. Koskimies and H. Muller, "Shimba - an Environment for Reverse Engineering Java Software Systems," *Software - Practice and Experience*, vol. 31, no. 4, pp. 371-394, 2001.

121. A. Hamou-Lhadj and T.C. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," *Proc. of IEEE International Conference on Program Comprehension*, pp. 181-190, 2006.

122. K. Koskimies, "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs," *Proc. of IEEE International Conference on Software Engineering*, pp. 366-375, 1996.

123. J. Koskinen, M. Kettunen and T. Systa, "Profile-Based Approach to Support Comprehension of Software Behavior," *Proc. of IEEE International Conference on Program Comprehension*, pp. 212-224, 2006.

124. M. Salah and S. Mancoridis, “Toward an Environment for Comprehending Distributed Systems,” *Proc. of of Working Conference on Reverse Engineering*, pp. 238-247, 2003.

125. T. Souder, S. Mancoridis and M. Salah, “Form: A Framework for Creating Views of Program Executions,” *Proc. of of IEEE International Conference on Software Maintenance*, pp. 612-620, 2001.

126. R. Oechsle and T. Schmitt, “Javavis: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (Jdi),” *Software Visualization*, Lecture Notes in Computer Science 2269, Springer Berlin Heidelberg, pp. 176-190, 2002.

127. L.C. Briand, Y. Labiche and J. Leduc, “Tracing Distributed Systems Executions Using Aspectj,” *Proc. of International Conference on Software Maintenance*, IEEE pp. 81-90, 2005.

128. T. Ishio, Y. Watanabe and K. Inoue, “Amida: A Sequence Diagram Extraction Toolkit Supporting Automatic Phase Detection,” *Proc. of Companion of the ACM International Conference on Software Engineering*, 2008.

129. S. Munakata, T. Ishio and K. Inoue, “Ogan: Visualizing Object Interaction Scenarios Based on Dynamic Interaction Context,” *Proc. of IEEE International Conference on Program Comprehension*, pp. 283-284, 2009.

130. 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,” *Proc. of IEEE International Conference on Engineering of Complex Computer Systems*, pp. 107-116, 2011.

131. J.K.-Y. Ng, Y.-G. Guéhéneuc and G. Antoniol, “Identification of Behavioural and Creational Design Motifs through Dynamic Analysis,” *Wiley Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 8, pp. 597-627, 2010.

132. F. Taïani, M.-O. Killijian and J.-C. Fabre, “Cosmopen: Dynamic Reverse Engineering on a Budget. How Cheap Observation Techniques Can Be Used to Reconstruct Complex Multi-Level Behaviour,” *Software: Practice and Experience*, vol. 39, no. 18, pp. 1467-1514, 2009.

133. H. Grati, H. Sahraoui and P. Poulin, “Extracting Sequence Diagrams from Execution Traces Using Interactive Visualization,” *Proc. of Working Conference on Reverse Engineering*, pp. 87-96, 2010.

134. N. Bawa and S. Ghosh, “Visualizing Interactions in Distributed Java Applications,” *Proc. of IEEE International Workshop on Program Comprehension*, pp. 292-293, 2003.

135. D. Kortenkamp, R. Simmons, T. Milam and J.L. Fernandez, “A Suite of Tools for Debugging Distributed Autonomous Systems,” *Proc. of IEEE International Conference on Robotics and Automation*, pp. 169-174 vol.161, 2002.

136. C. Bennett, D. Myers, M.-A. Storey, D.M. German, D. Ouellet, M. Salois and P. Charland, "A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams," *Wiley Software Maintenance and Evolution*, vol. 20, no. 4, pp. 291-315, 2008.

137. D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon and O. Nierstrasz, "Exploiting Dynamic Information in Ides Improves Speed and Correctness of Software Maintenance Tasks," *IEEE Transactions Software Engineering*, vol. 38, no. 3, pp. 579-591, 2012.

138. P. Tonella and A. Potrich, "Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram," *Proc. of International Conference on Software Maintenance*, pp. 54-63, 2002.

139. D. Myers, M.-A. Storey and M. Salois, "Utilizing Debug Information to Compact Loops in Large Program Traces," *Proc. of IEEE European Conference on Software Maintenance and Reengineering*, pp. 41-50, 2010.

140. M. Desnoyers, *Common Trace Format (Ctf) Specification (V1. 8.2)*, 2012; http://diamon.org/ctf/.

141. L.C. Briand, Y. Labiche and J. Leduc, "Towards the Reverse Engineering of Uml Sequence Diagrams for Distributed Java Software," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 642-663, 2006.

142. B. Cornelissen, A. van Deursen, L. Moonen and A. Zaidman, "Visualizing Testsuites to Aid in Software Understanding," *Proc. of European Conference on Software Maintenance and Reengineering*, pp. 213-222, 2007.

143. R. Kollmann and M. Gogolla, "Capturing Dynamic Program Behaviour with Uml Collaboration Diagrams," *Proc. of European Conference on Software Maintenance and Reengineering*, pp. 58-67, 2001.

144. S. Fischmeister and P. Lam, "Time-Aware Instrumentation of Embedded Software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652-663, 2010.

145. D. Bruening, T. Garnett and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," *Proc. of International Symposium on Code Generation and Optimization*, pp. 265-275, 2003.

146. J.K. Hollingsworth, B.P. Miller and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *Proc. of Scalable High-Performance Computing Conference*, pp. 841-850, 1994.

147. F. Chen and G. Rosu, "Mop: An Efficient and Generic Runtime Verification Framework," *Proc. of ACM conference on Object-oriented programming systems and applications*, 2007.

148. M. Arnold and B.G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," *Proc. of ACM conference on Programming language design and implementation*, pp. 168-179, 2001.

149. M.L. Corliss, E.C. Lewis and A. Roth, "Low-Overhead Interactive Debugging Via Dynamic Instrumentation with Dise," *Proc. of International Symposium on High-Performance Computer Architecture*, pp. 303-314, 2005.

150. C. Watterson and D. Heffernan, "Runtime Verification and Monitoring of Embedded Systems," *IET Software*, vol. 1, no. 5, pp. 172-179, 2007.

151. Y. Labiche, B. Kolbah and H. Mehrfard, "Combining Static and Dynamic Analyses to Reverse Engineer Scenario Diagrams," *Proc. of IEEE International Conference on Software Maintenance*, pp. 130-139, 2013.

152. M. Desnoyers and M.R. Dagenais, "The Lttng Tracer: A Low Impact Performance and Behavior Monitor for Gnu/Linux," *Proc. of Ottawa Linux Symposium*, 2006.

153. L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder and P. Tuma, "Introduction to Dynamic Program Analysis with Disl," *Science of Computer Programming*, pp. 1 - 16, 2014.

154. E. Kuleshov, "Using the Asm Framework to Implement Common Java Bytecode Transformation Patterns," *Proc. of Aspect-Oriented Software Development*, 2007.

155. M. Dahm, *Byte Code Engineering with the Bcel Api*, Technical Report B-17-98, Freie Universitat Berlin, 2001;

156. S. Chiba, "Load-Time Structural Reflection in Java," *Proc. of 14th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 313–336, 2000.

157. M. Desnoyers and M.R. Dagenais, "The Lttng Tracer: A Low Impact Performance and Behavior Monitor for Gnu/Linux," *Proc. of Ottawa Linux Symposium (OLS)*, Linux Symposium, pp. 209-224, 2006.

158. M. Desnoyers, P.E. McKenney, A.S. Stern, M.R. Dagenais and J. Walpole, "User-Level Implementations of Read-Copy Update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375-382, 2012.

159. M. Desnoyers and M. Dagenais, "Lttng: Tracing across Execution Layers, from the Hypervisor to User-Space," *Proc. of Linux symposium*, 2008.

160. P.-M. Fournier, M. Desnoyers and M.R. Dagenais, "Combined Tracing of the Kernel and Applications with Lttng," *Proc. of Linux Symposium*, pp. 87-93, 2009.

161. B. Dufour, C. Goard, L. Hendren, O.d. Moor, G. Sittampalam and C. Verbrugge, "Measuring the Dynamic Behaviour of Aspectj Programs," *Proc. of Object-oriented programming, systems, languages, and applications*, ACM, pp. 150-169, 2004.

162. J.D. Gradecki and N. Lesiecki, *Mastering Aspectj - Aspect-Oriented Programming in Java*, Wiley, 2003.

163. T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

164. M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

165. P. Haggar, *Practical Java: Programming Language Guide*, Addison-Wesley, p. 320, 2000.

166. J. Bloch, *Effective Java*, Pearson Education, 2008.

167. J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, M. Snir and R.D. Lawrence, "Java Programming for High-Performance Numerical Computing," *IBM Systems Journal*, vol. 39, no. 1, pp. 21-56, 2000.

168. K. Wadleigh and I. Crawford, *Software Optimization for High-Performance Computing*, Prentice Hall Professional, 2000.

169. M.A. Weiss, *Data Structures and Algorithm Analysis in Java*, Pearson, 3'rd edition, p. 640 2011.

170. D.F. Bacon, S.L. Graham and O.J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345-420, 1994.

171. J. Shirazi, *Java Performance Tuning*, O'Reilly Media, 2003.

172. E. Hilsdale and J. Hugunin, "Advice Weaving in Aspectj," *Proc. of Aspect-oriented Software Development*, ACM, 2004.

173. P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble, "Optimising Aspectj," *Proc. of on Programming Language Design and Implementation*, ACM, 2005.

174. C. Bockisch, S. Kanthak, M. Haupt, M. Arnold and M. Mezini, "Efficient Control Flow Quantification," *Proc. of Object-oriented programming systems, languages, and applications*, ACM, pp. 125-138 2006.

175. C. Hundt, D. Stohr and S. Glesner, "Optimizing Aspect-Oriented Mechanisms for Embedded Applications," *Objects, Models, Components, Patterns*, Lecture Notes in Computer Science, Springer, pp. 137-153, 2010.

176. E. Bodden, L. Hendren and O. Lhotak, "A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring," *Proc. of European conference on Object-Oriented Programming*, Springer-Verlag, pp. 525 - 549, 2007.

177. P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble, "Abc : An Extensible Aspectj Compiler," *Transactions on Aspect-Oriented Software Development I*, Lecture Notes in Computer Science 3880, Springer, pp. 293-334, 2006.

178. R.M. Golbeck, P. Selby and G. Kiczales, "Late Binding of Aspectj Advice," *Objects, Models, Components, Patterns*, Lecture Notes in Computer Science 6141, Springer, pp. 173-191, 2010.

179. R.M. Golbeck and G. Kiczales, "A Machine Code Model for Efficient Advice Dispatch," *Proc. of Virtual machines and intermediate languages for emerging modularization mechanisms*, ACM, 2007.

180. C.-H. Lung, S. Ajila and W.-L. Liu, "Measuring the Performance of Aspect Oriented Software: A Case Study of Leader/Followers and Half-Sync/Half-Async Architectures," *Information Systems Frontiers*, pp. 1-14, 2013.

181. R.B. Setty, R.E. Dyer and H. Rajan, *Weave Now or Weave Later: A Test-Driven Development Perspective on Aspect-Oriented Deployment Models*, Technical Report, 2008;

182. A. Hamou-Lhadj and T.C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces," *Proc. of International Workshop on Program Comprehension*, IEEE, pp. 159-168, 2002.

183. T. Wang and A. Roychoudhury, "Using Compressed Bytecode Traces for Slicing Java Programs," *Proc. of 26th International Conference on Software Engineering*, pp. 512-521, 2004.

184. D. Baca, "Tracing with a Minimal Number of Probes," *Proc. of IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 74-83, 2013.

185. T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. of ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 46-57, 1996.

186. F. Chen, D.A. Koufaty and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," *Proc. of international conference on Supercomputing*, ACM, 2011.

187. C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim and Y. Choi, "A High Performance Controller for Nand Flash-Based Solid State Disk (Nssd)," *Proc. of 21st IEEE Non-Volatile Semiconductor Memory Workshop*, pp. 17-20, 2006.

188. P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz and D.A. Patterson, "Raid: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145-185, 1994.

189. C. Gulcu, "Log4j Architecture," *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java*, 2003.

190. A. Chervenak, I. Foster, C. Kesselman, C. Salisbury and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187-200, 2000.

191. B. Allcock, J. Bester, J. Bresnahan, A.L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel and S. Tuecke, "Data Management and Transfer in High-Performance Computational Grid Environments," *Parallel Computing*, vol. 28, no. 5, pp. 749-771, 2002.

192. W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger and K. Stockinger, "Data Management in an International Data Grid Project," *Grid Computing — Grid 2000*, Lecture Notes in Computer Science 1971, Springer, pp. 77-90, 2000.

193. R.L. Grossman, Y. Gu, M. Sabala and W. Zhang, "Compute and Storage Clouds Using Wide Area High Performance Networks," *Future Generation Computer Systems*, vol. 25, no. 2, pp. 179-183, 2009.

194. , *Infiniband Architecture Specification: Release 1.3*, InfiniBand Trade Association, 2015; http://www.infinibandta.org/.

195. T. Pender, *Uml Bible*, Wiley, 2003.

196. OMG, *Uml 2.0 Superstructure Specification*, Final Adopted Specification ptc/2007-11-02, Object Management Group, 2007;

197. L.C. Briand, Y. Labiche and J. Leduc, "Tracing Distributed Systems Executions Using Aspectj," *Proc. of IEEE International Conference on Software Maintenance*, pp. 81-90, 2005.

198. Y.D. Liang, *Introduction to Java Programming: Brief Version*, Prentice Hall, 2010.

199. T. Copeland, *Generating Parsers with Javacc: An Easy-to-Use Guide for Developers*, Centennial Books, 2007.

200. C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering: An Introduction*, The Kluwer International Series In Software Engineering, 2000.

201. G.C. Necula, "Proof-Carrying Code," *Proc. of ACM Symposium on Principles of Programming Languages*, pp. 106--119, 1997.

202. A. Silberschatz, P.B. Galvin and G. Gagne, "Process Synchronization," *Operating System Concepts*, 7'th ed., Wiley, 2005.

203. F.T. Sheldon, K. Jerath and H. Chung, "Metrics for Maintainability of Class Inheritance Hierarchies," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 3, pp. 147-160, 2002.

204. R.R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald and D. Scuse, *Weka Manual, Version 3-7-7*, User manual, 2012;

205. J. Waller, "Performance Benchmarking of Application Monitoring Frameworks," Department of Computer Science, Kiel University, 2014.

206. R. Love, "The Page Cache and Page Writeback," *Linux Kernel Development*, 2'nd ed., Addison-Wesley, 2004.

207. M.G. D'Elia and V. Paciello, "Performance Evaluation of Labview on Linux Ubuntu and Window Xp Operating Systems," *Proc. of 19th Telecommunications Forum (TELFOR)*, IEEE, pp. 1494-1498, 2011.

208. A. Silberschatz, P.B. Galvin and G. Gagne, "I/O Systems," *Operating System Concepts*, 9 ed., Addison-wesley Reading, 2012.

209. P. Lam, E. Bodden, O. Lhota¡k and L. Hendren, "The Soot Framework for Java Program Analysis: A Retrospective," *Proc. of Cetus Users and Compiler Infastructure Workshop*, pp. 35, 2011.

210. "Pragmadev Tracer," 2013; http://www.pragmadev.com/product/tracing.html.

Appendix A Finding best optimization activities for each characteristic

In this appendix, we conduct different experiments to find best optimization activities for: the gathering of information (Section A.1), the encoding of information (Section A.2), and the logging of information (Section A.3). We report on conducted experiments in this appendix in Chapter 5 when we explain different optimization activities. It is worth to note that we evaluate the Light instrumentation based on one optimization activity at the time. We investigate which optimization alternative worsens and which reduces the overhead.

For experiments in this appendix, we use OE and Weka case studies and OE_TC2 and Weka_TC4 respectively. We present each experiment based on the format we discussed in Chapter 4. Though, we do not report on the following common components for each experiment separately: We used Linux time command for all experiments. In addition, except for two experiments, we measured the average execution time of 100 executions for all other experiments. Before measuring the time, we verified the correctness of the experiment by confirming the equivalency of the generated trace from that experiment with the generated trace from the Light instrumentation. Once the execution times are collected, we verified the correctness of data-sets by comparing samples with a *Student t-test* (we observed differences are statistically significant at α=0.05). We confirmed this with *Mann-Whitney U* non-parametric test.

We conduct most of experiments in this appendix only based on OE_TC2 test case since OE is easier to set up. However, we note that a few number of object initializations in OE (compared to Weka) may impact overhead when we investigate different object identification mechanisms. Therefore, we turned to Weka for

experimenting different object identification mechanisms. For other cases of optimization, different optimization activities would not be impacted by the differences of the two cases study systems since the number of calls to constructor has the same impact on overhead as the number calls to method has.

A.1 The collecting of information

In this section, we conduct different experiments to find best refactoring activities suggested in 5.4.1. These experiments address optimizations on: variable's data-type and declaration type (Sections A.1.1 and A.1.2), dynamic context collection (Section A.1.3), and object identification mechanism (Section A.1.4).

A.1.1 The variable's data-type and declaration type

The choice of data-type and declaration type can potentially change the performance of Light instrumentation. We replace local variable of type `LinkedList` (such as line 8 in Fig. 6) and class variable of type `int` (such as line 1 and 2 in Fig. 11) in `MethodAspect` and `IdentifierAspect` with other local or global Java data-types to understand which data-type with which declaration type performs more efficiently in the Light instrumentation. We conduct experiments with objects of type `LinkedList`, `StringBuilder` and `String` as well as primary Java data-types `short`, `byte` and `String`[57] where each data-type (or object) is declared either as a class (or aspect) variable or method (or advice) variable. Note that except primary Java data-types all other data-types in Java should be initialized as a new object before assigning to a variable.

[57] In this case Java simulates the variable of type `String` similar to primary data-types by assigning string literals directly to the variable.

A.1.1.1 Experiment design

We conduct nine experiments based on OE_TC2 to evaluate data-type and declaration type alternatives other than local `LinkedList` and global `int`. We look at data-types `String` and `StringBuilder` to replace `LinkedList` and primary data-types `short` (16 bit) and `byte` (8 bit) to replace `int` (32 bit). Data-types `String` and `StringBuilder` are chosen since they are common Java data-types for immutable and mutable string variables respectively (we do not choose `StringBuffer` since Java documentation suggests `StringBuilder` performs faster). Primary data-types `short` and `byte` are chosen since they require less memory for initialization compared to `int`. In the first experiment, we modify the Light instrumentation by replacing the local variable of type `LinkedList` in advices and inter-typed methods (i.e., `className.objectIDgenerator()`) with the `static` variable of type `LinkedList` in aspects (i.e., global variable) to measure the execution time. In another set of experiments, we conduct four experiments where, in the first two, we replace the local variable of type `LinkedList` in the Light instrumentation with the local variable of types `String` and `StringBuilder` and in the second two, similarly to the first experiment, we replace the local variable of type `LinkedList` with the `static` variable of types `String` and `StringBuilder` in aspects. When we create `StringBuilder` object, we assign the length of log item to the constructor of this object. Note that we initialized local `String` variable as a new object where we used the keyword `new` for a single `String` variable creation throughout the advice or inter-typed methods for the whole log item (in case of the global variable, a single `static String` variable is `new`ed in the aspect). Alternatively, we can pass string literals directly to the `String` variable without using the

keyword `new`. We think `String` is performing more efficiently in this case since JVM uses its string pool for variable initialization. In addition, we suspect there is overhead due to concatenation operation since a local `LinkedList` variable in advice uses the concatenation operation 12 times to form a log item. Therefore, we run another experiment (sixth experiment) to measure execution time where we replace the local variable of type `LinkedList` in advices and inter-typed methods with 17 and three local variables of type `String` (literals assigned directly) respectively. In this case the Logger needs to be modified based on advices and methods. Note that multiple local `String` declarations in this situation cause very little variable initialization overhead since Java initializes `String` data-type (with direct assignment of string literals) similar to its primary data-types with very little initialization overhead. In fact, in the next experiment (in Section A.1.2), we show that the multiple initializations overhead is negligible when string literals are directly assigned to `String` data-type in the Light instrumentation (the difference between single declaration and multiple declarations is a fraction of millisecond). Similarly to the sixth experiment, we conduct the seventh experiment this time based on local `String` objects with no concatenation operation to measure the execution time. This execution time helps us to better understand the concatenation overhead in the Light instrumentation. Finally, we conduct the last two experiments by replacing the `static` variable of type `int` in the `IdentifierAspect` (lines 1 to 4 in Fig. 11) with `short` and `byte` to investigate performance improvement due to using less memory.

It is important to note that when `static` mutable data-types (i.e., global `LinkedList` and `StringBuilder`) replace the local `LinkedList`, unlike the local

`LinkedList`, they do not cause any initialization overhead in advices (or inter-typed methods). However, `static` mutable data-types impose an additional overhead due to deletion operation at the end of advices (or methods) by emptying the global variable after passing the log item string to the Logger. There is no such need (i.e., deletion operation) in the case of global immutable variable (i.e., global `String` object) since a new immutable object has to be created when a that object is modified. Note that, except the sixth and seventh experiments, new data-types (either local or global) do not add or remove any new concatenation operation to/from the ones already exist in the local `LinkedList`.

For all nine experiments, we measure execution time by commenting out calls to the Logger class in advices and inter-typed methods (same as the way Light\CallsToLogger time is calculated in Table 6). This way, we remove the logging overhead from the Light instrumentation and able to better monitor the impact of data-type changes in the collecting of information. Note that we limited our choices of data-type only to those data-types mentioned above since they satisfy simple string operations efficiently, namely: insert string operation, remove operation, and return string operation. Other Java data-types (such as `HashMap` and `TreeSet`) do other operations efficiently (such as searching on big amount of data) rather than efficiently performing basic operations on string literals and integer numbers.

A.1.1.2 Results

Table 17 shows the execution times of experimentations on OE_TC2 with different data-types and declaration-types when no call is made from aspects to the Logger. We reiterate the execution time of Light\CallsToLogger (second column) from

Chapter 4 (4.9.2) as a index to compare with other execution times. Before calculating the execution time, we verified the equivalency of the generated trace in all experiments with the trace of Light instrumentation by activating calls to the logger. Table 13 shows that both global `LinkedList` and `StringBuilder` data-types (third and ninth columns) performed worse than local `LinkedList`. The slight increase (2%) in the execution time of global `LinkedList` (compared to execution time of local `LinkedList`) shows that the overhead due removing log item at the end of advices is more than the overhead due to `LinkedList` creation (deletion operation (remove for n elements: O(n)) versus creation). In the case of global `StringBuilder`, the execution time (due to replacing local `LinkedList`) marginally increased (by 7%). Even the local declaration of `StringBuilder` (in the eighth column) did not perform better than local `LinkedList` (by adding 9% to overhead). The `StringBuilder` object is an array based data-type where a variable-length array contains a sequence of characters (of type `CharSequence`). Therefore, the overhead of inserting new strings to a log item and removing the log item in the local `LinkedList` is still better than the type `StringBuilder` (declared either locally or globally). Note that by assigning the number of characters in the log item to the constructor of `StringBuilder`, JVM does not change the length of array dynamically (and consequently do not add more overhead).

Time (second) / Case Study	LinkedList (local)	LinkedList (global)	String (local object)	String (global object)	String (distinct local object)	String (local, passing literals)	StringBuilder (local)	StringBuilder (global)	short (IdentifierAspect)	byte (IdentifierAspect)
OE_TC2	3.72 (100 %)	3.80 (+2%)	5.81 (+52 %)	6.02 (+62 %)	2.22 (-40%)	2.02 (-46%)	4.05 (+9%)	3.99 (+7%)	3.66 (-1.5%)	3.66 (-1.5%)

Table 17 Mean execution times (100 executions) for different data-types with no call to the Logger

When we replaced local `LinkedList` with a single `String` object, declared either locally in advices (second experiment) or globally in aspects (fourth experiment), execution times (fourth and fifth columns) increased by more than 50 percent. When we removed all concatenation operations and declared 17 local `String` objects in advices (seventh experiment), the execution time (sixth column) reduced by 40 percent. This difference between execution times (when a single or multiple `String` objects declared) indicates the heavy cost of using concatenation operation due to the immutable creation of `String` instances: each time a `String` object is appended with new characters, a new `String` instance is created. As we expected, when string literals are directly assigned to the `String` without using any concatenation operation (sixth experiment), the execution time (seventh column) cut nearly half of the local `LinkedList` execution time. When we replaced `static int` with `static short` or `static byte` (eighth and ninth experiments), execution times (tenth and eleventh columns) slightly reduced. Though, variables of type `short` and `byte` need to be cast to `int` again before returning the value when calls to the logger are activated. Therefore, it does not worth to change the `int` variables in the Light instrumentation.

A.1.2 The overhead of String declarations

We conduct a small experiment to understand the difference in overhead of single declaration versus multiple declarations of the local variable of type `String` when string literals are directly assigned to the variable.

A.1.2.1 Experiment design

We define two `for` loops with nearly four million iterations where each loop assigns a string of character, with a size equal to the average size each log item (i.e., average size = trace size/number of log items) in OE_TC2 (average size = 231 bytes), to the local variable(s) of type `String`. The first loop assigns the one string of character to a `String` variable whereas the second loop splits the string of characters into 17 substrings where each substring is assigned to a local variable of type `String`. Note that the number of iteration for both loops is the same as the number times that Logger (in Light) is called to write each individual log item on the disk in OE_TC2. The 17 substrings are equal to the number of `String` variables that are passed to the Logger from advices. Therefore, the difference between execution times should indicate the amount of overhead due to multiple local `String` declarations (similar to the seventh column of in Section A.1.1) as opposed to the single local `String` declaration. We measure time for each loop execution based on Java `System.nanoTime()` method and run each loop for 500 times.

A.1.2.2 Results

Table 18 shows the average execution time (in millisecond) for each loop, which includes initializing `String` variable(s) and assigning the log item to the `String` variable(s): either assigning to a single `String` variable (the second column) or assigning to 17 `String` variables (third column). The difference between execution time is in the

scale of 10^{-4} which is not a significant in our calculation for the overhead in Light. Thus, declaring multiple `String` variables does not cause overhead in the Light instrumentation.

case study	One String variable	17 String Variable
OE_TC2 (Millisecond)	3.93	4.86

Table 18 Mean execution times (500 executions) for string literals assignment

The reason for very small `String` initialization overhead is due to use permanent memory pool by JVM. This memory pool is a limited cache size compared to the heap and it takes very little time to initialize and assign string literals to the `String` variable.

A.1.3 Modifying dynamic context collection in aspects

In this section, we want to understand the effect of modifying dynamic context collection on the performance of Light instrumentation. More precisely, we ask the following RQ: How does the overhead in the Light instrumentation change when we modify dynamic context collection by a) reducing the number of method calls within aspect (i.e., NMC_{Ins}), b) using AspectJ APIs, c) using Java reflection APIs?

A.1.3.1 Experiment design

We design three experiments to answer the RQ. In the first experiment we hardcode all helper methods (Fig. 9) within `MethodAspect` (lines 87 to 111 in Fig. 5) in advice bodies. More specifically, we replace 7 helper methods (i.e., `getStaticClassName`, `getBindToClassName`, `getMethodSignature`, `getLineNumber`, `getFileName`, `getStaticBindToClassName`, `getNewBindToClassName`) with their bodies' in advices. In the second experiment, we use AspectJ APIs to replace helper methods within `MethodAspect` and hard code helper methods in advice body if there is no AspectJ API for that purpose. More specifically, we replace four static methods with

AspectJ APIs: replacing methods `getBindToClassName`, `getMethodSignature`, `getFileName`, `getLineNumber` with AspectJ APIs `getName()`, `getSignature()`, `getFileName()`, `getLine()` respectively. In addition, we replace three helper methods (i.e., `getStaticClassName`, `getStaticBindToClassName`, `getNewBindToClassName`) with their bodies in advices. In the third experiment, we replace the AspectJ API we used for capturing object identifiers (e.g., lines 10 to 15 in Fig. 6) with the Java equivalent using Java reflection API (e.g., Fig. 24).

We evaluate experiments based on two criteria: NMC_{Ins} and execution time. Similarly to the previous experiments, after verifying the equivalency of the generated trace from each experiment with the trace from the Light instrumentation, we disable calls to the Logger in advices and inter-typed methods when we measure execution times. This provides solid observation over the dynamic context collection's modifications. In addition, we are concerned with method calls originating from the `instrument` package. Recall that the number of method calls (i.e., NMC_{Ins}) is the number of places (callers) in the instrumentation package where an invocation (or a series of invocations) is made to another method in the instrumentation package. We developed another instrumentation for each experiment, as we discussed in Section 4.4.1, to report NMC_{Ins}.

A.1.3.2 Results

Table 19 shows execution times and NMC_{Ins}s of OE_TC2 for different experiments. The second column (index column) shows the NMC_{Ins} and execution time for the Light instrumentation. The result of the first experiment (third column) shows that even though the NMC_{Ins} was reduced nearly by half, the execution was reduced slightly. This indicates that AspectJ compiler does many of optimizations that we manually

hardcoded in the advice body. However, when we used AspectJ APIs and not used any helper methods (fourth column), the execution time has reduced largely (by 40%). This shows the efficiency of AspectJ APIs over the plain Java implementation. When we replaced AspectJ API with Java reflection API in the third experiment (fifth column), the execution time largely increased. Therefore, we can conclude from our experiments that AspectJ APIs performs more efficiently compared to plain Java or reflective Java APIs implementations.

Case Study		Light	NoHelper	NoHelper WithAsjAPI	LightWith JavaAPI
OE_TC2	Time	3.72 (100%)	3.68 (-1%)	2.22 (-40%)	6.96 (+74%)
	NMC_{Ins}	36,000,018 (100%)	20,000,013 (-45%)	20,000,013 (-45%)	28,000,022 (-22%)

Table 19 Mean execution times (100 executions) for different dynamic context collection mechanisms

A.1.4 Modifying object identification mechanism

In this section, we examine different modifications in the object identification mechanism for the possibility of overhead reduction. We want to understand how does the overhead of Light instrumentation change when a) the callees's object identifier is captured with a new advice instead of inter-typed methods (`objectIDgenerator()`), b) a `HashMap` structure is used instead inter-typed methods.

A.1.4.1 Experiment design

We design two experiments where in the first experiment we replace the lines in `objectIDgenerator()` method in the `IdentifierAspect` (lines 6, 8, 9 in Fig. 11) with a new `after():execution(`constructor`)` advice (Fig. 21) in the `MethodAspect`. In this case, no call to the logger will happen during inter-type declaration. However, such an

advice would need to collect class name information dynamically (line 15 in Fig. 21), whereas the class name information is retrieved statically in the missing inter-typed method (line 8 in Fig. 11). In the second experiment, we modify Light instrumentation in that we remove all inter-typed `objectIDgenerator()` methods as well as calls to the logger in the `IdentifierAspect` aspect (Fig. 11). In this case, we can identify object instances, again with the `after(): execution(`constructor`)` advice (Fig. 21), and use a Java data-structure that counts, stores, and looks up object instances. For this experiment, we choose the `HashMap` data-structure (`IdGen` class in Fig. 22). As we mentioned in 5.4.1.3, the `IdGen` class uses class name as the key to assign values (i.e., `objectID`) to the `HashMap`. In addition, we modify `IdentifierAspect` aspect to access the `IdGen` class (Fig. 23). We refer a reader to 5.4.1.3 for more details about these classes.

We measure execution time based on OE_TC2 test case for the first experiments and Weka_TC test case for the second experiment.

A.1.4.2 Results

The third column in the Table 20 shows that when we captured callee's object identifier with the new advice in the first experiment, the overhead reduction was negligible in OE. In the second experiment (using of `HashMap` and `after():execution()` advice together), even though the overhead has reduced by 13% for the OE case study, there was no overhead reduction for the Weka case study (third column in the Table 21). As we mentioned at the beginning of this Appendix, a few object initialization was the reason of overhead reduction in OE, whereas in the case of hundreds of object initializations in Weka we did not observe such reduction.

Case Study	Light	4AdviceLight
OE_TC2	15.44 (100%)	15.28 (-1%)

Table 20 Mean execution times (100 executions) for Light with new execution advice

Case Study	Light	HashMap
OE_TC2	15.44 (100%)	13.42 (-13%)
Weka_TC4	17.51 (100%)	17.29 (-1%)

Table 21 Mean execution times (100 executions) for Light with `HashMap`

A.2 The encoding of information

In this section, we investigate different optimization activities suggested for encoding information (5.4.2) and their impact on overhead. We want to answer the following RQs in this section: RQ1: do different Java character encodings change the overhead and if so to what extent? RQ2: how does the use of raw format instead of Light format can change the overhead? RQ3: does the use of a logger with a specialized method for each advice reduce the overhead? RQ4: How does long class names increase the overhead?

A.2.1 Experiment design

We design four experiments to answer each RQ: First, we examine different Java character encodings (US-ASCII (7-bit encoding), ISO-8859-1 (8-bit encoding), UTF-16 (16-bit encoding)) on the Light instrumentation and compare them with the default character encoding (UTF-8) of JVM in our system. We try different encodings since using a Java encoding with a minimal required space at compile time can reduce the amount of memory being used by aspects. We pass the encoding type as a JVM parameter at runtime, however, we note that SUS should be compatible with the chosen type of encoding (i.e., SUS does not output any character which does not exist in a chosen encoding).

Second, we change the Light instrumentation in that use raw format instead of Light format. In other words, aspects only pass the required dynamic information to the logger without using any specific format. Processing this information to any specific format is left for after execution. Therefore, we expect reduction on the trace size and consequently on the overhead. The local `LinkedList` variable in `callMethod`, `callStaticMethod`, `callConstructor` advices and `objectIDgenerator` method contains seven, six, six, and two pieces of information respectively.

Third, in order to understand whether modifying the logger to pass fewer arguments from aspects to the logger reduces the overhead, we design two versions of modified Light instrumentation: one with specialized logger and one without specialized logger. In the instrumentation with specialized logger, we modify the `instrument()` method in the logger based on the type of each advice and method. In this case, aspects pass fewer arguments to the logger. We change the Light instrumentation in that `callMethod`, `callStaticMethod`, `callConstructor` advices and `objectIDgenerator` method pass six, five, five, and two variables of type `String`, when literals are assigned directly, respectively to the logger. In the instrumentation without specialized logger, aspects and inter-typed method pass 17 and 3 `String` variables respectively to a logger that does not use of specialized methods (this instrumentation is the same as instrumentation mentioned in sixth experiment of A.1.1.1). Then, the comparison between execution times of these two instrumentations should show the effect of passing fewer arguments. Note that generated trace from both experiments should be equal.

Fourth, we examine the impact of long class names on overhead by changing the name of each class in OE from a single character name to a 20-character name. This

should disclose whether there is any impact on overhead if SUS contains long class names and to what extent the overhead can change.

We compare the execution times of all experiments based on OE_TC2 test case. Note that except the second experiment, all other experiments generate trace according to the Light format.

A.2.2 Results

Columns two to five in Table (RQ1: first experiment) show no substantial differences in overheads of different Java character encodings, only US-ASCII encoding performed slightly better. However, we note that combining the most efficient character encoding and Java data-type would considerably reduce the overhead for this type of optimization (as shown for the efficient data-type in A.1.1). Sixth column (RQ2: second experiment) shows 25% overhead reduction when only dynamic data (raw format) is passed to the logger. In addition, we observed the 66% shorter trace (trace size: 337.6 MB) compared to the Light trace (trace size: 989.6 MB). Eighth and ninth columns (RQ3: third experiment) indicate that specialized logger can slightly reduce the overhead compared to the instrumentation without specialized logger (eighth column). However, this reduction is not considerable in our optimization. Seventh column (RQ4: fourth experiment) shows that the Light instrumentation based on the modified OE case study with larger class names indeed boosted both trace size (trace size: 1320.8) and overhead compared to unmodified OE. However, despite our initial intuition, class name was not a big source of overhead.

Case study	ASCII	ISO-8859-1	UTF-8 (default)	UTF-16	RawFormat	LargeClassName	AllArgsLogger	FewerArgsLogger	EncodedClassMethodName
OE_TC2	15.05 (-2%)	15.12 (-1.3%)	15.31 (100%)	15.32 (+1%)	11.46 (-25%)	16.47 (+7.5%)	11.40 (100%)	11.28 (-1%)	14.37 (-7%)

Table 22 Mean execution times (100 executions) of different practices for encoding data

A.3 The logging of information

In this section we conduct experiments to study different optimization activities suggested for logging information. In Section A.3.1, we look into activities suggested in 5.4.3 for logging on a remote machine. Next, in Section A.3.2, we investigate optimization activities for logging on a local logger machine.

A.3.1 Remote logger

We want to understand to what extent using Log4J framework and using customized logger based on TCP and UDP protocols can change the overhead.

A.3.1.1 Experiment design

We design three experiments based on OE case study to examine different logging mechanisms. In the first experiment, we modify Light instrumentation to log both on client (**Error! Reference source not found.** and **Error! Reference source not found.**) and server using Log4J library. We change `MethodAspect` by adding a new advice to start a new thread on client and establish a client-server connection before the start of SUS. We configure Log4J on client-side accordingly (Fig. 31): the client logger uses asynchronous communication (line 5 in Fig. 31), increases the buffer size to five gigabytes (line 6 in Fig. 31), and opens socket to establish connection with server (line 10 in Fig. 31). In the second experiment, we replace the generic logger of Light

instrumentation (Fig. 12) with a TCP-based logger on client (**Error! Reference source not found.**) and server (**Error! Reference source not found.**). Similarly to the previous experiment, we change `MethodAspect` to start a new thread before SUS starts and open a TCP socket on the client. In addition, we use queue as a buffering mechanism in both client (**Error! Reference source not found.**) and server. Using queue makes aspects, T CP connections, and writes to disk operation (on server) work asynchronously from each other. In the third experiment, we replace the generic logger in the Light instrumentation (Fig. 12) with a UDP-based logger on client (**Error! Reference source not found.**) and server (**Error! Reference source not found.**). Similarly to the second experiment, we start client socket (`DatagramSocket()`) before the beginning of SUS's execution, and use a queue both on client and server.

```
1   #Define the log4j configuration for local application
2   log4j.rootLogger=ERROR, server, ASYNC
3
4   #We specify that client establishes asynchronous communication with 5 GB
5   buffer
6   log4j.appender.ASYNC = org.apache.log4j.AsyncAppender
7   log4j.appender.ASYNC.BufferSize = 5000000
8   log4j.appender.ASYNC.LocationInfo = true
9
10  #We will use socket appender
11  log4j.appender.server=org.apache.log4j.net.SocketAppender
12
13  #Port where socket server will be listening for the log events
14  log4j.appender.server.Port=4712
15
16  #Host name or IP address of socket server
17  log4j.appender.server.RemoteHost=134.117.61.172
18
19  #Define any connection delay before attempting to reconnect
    log4j.appender.server.ReconnectionDelay=10
```

Fig. 31 client configuration

A.3.1.2 Results

Table 23 shows the execution time and recovered trace size on the server side based on OE_TC2 for each experiment. We report on two types of execution times for each experiment: $time_{SUS}$ and $time_{log}$. The $time_{SUS}$ indicates the amount of time spent to

execute SUS on the client whereas the $time_{log}$ shows the time span that the client virtual machine has to wait from after the end of the execution of the SUS until all the packets has transferred from the client to the server.

Note that we calculated the average execution time from 10 executions in the first experiment and 100 executions for the second and third experiments.

We show two execution times for the optimized instrumentation: $time_{SUS}$ is the amount of time to execute Weka_TC and illustrates nearly 74% overhead reduction compared to Light. On the other hand, the client virtual machine has to wait for 285 seconds ($time_{log}$) after the end of the execution of the SUS (Weka) for all the packets to be transferred from the client to the server. The risk, although we have not observed it, is that this may translate into overhead for the SUS if buffers used for network communication get full. The reason for such TCP behaviour is that the TCP congestion control mechanism slows down the rate by which the client packets are sent and consequently creates a bottleneck.

The second column (first experiment) shows that using Log4J library not only did not improve the Light instrumentation's overhead, but it worsened the Light's overhead by a very large margin. In addition, the $time_{SUS}$ and the $time_{log}$ were the same in this experiment. This indicates that even though we enabled asynchronous network communication and allocated a large buffer size, the Log4J framework did not perform well based on the requirements of our instrumentation. The trace size shows that Log4J is lossless for transferring data.

The $time_{log}$ in the third and fourth columns (second and third experiments) shows that both TCP and UDP based loggers were able to largely reduce the overhead of Light

instrumentation. However, as we noted in 5.4.3, $time_{log}$ columns show a large waiting time for the next execution task of the SUS on the client. The trace size of these loggers in the table indicates that the TCP-based logger is a lossless logger, while the UDP-based logger loses on average 2% of total trace size. Therefore, since we are looking for a lossless transfer of information, UDP-based logger would not be a proper logger for our instrumentation.

Case Study		Log4J		TCP		UDP	
		$time_{SUS}$	$time_{log}$	$Time_{SUS}$	$time_{log}$	$Time_{SUS}$	$time_{log}$
OE_TC2	Time (sec)	5357	5357	6.67 (-57%)	345.8	6.75 (-56%)	98.55
	Size (MB)	989.6		989.6		974.2	

Table 23 Mean execution times (10 and 100 executions) for logging mechanisms

A.3.2 Local logger

In this section, we want to understand which one of the local logger implementations (CacheLogger and BatchLogger) performs better than another. In addition, we want to know that is there any performance gain in a better performing implementation over the Light and if so, how much is that performance improvement. Furthermore, we want understand how does the type of disk technology impact the logging mechanism. Finally, we design an experiment to realize the effectiveness of the type of OS on our instrumentation approach.

A.3.2.1 Experiment design

We design three experiments thereby: in the first experiment we examine the performance of CacheLogger and BatchLogger implementations based on OE_TC2 case study by changing `cacheSize` and `queueBatcheSize` according to the following cache size and number of logs in each batch (resp.): 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9.

Recall that `cacheSize` indicates the size of second buffer (in byte) in CacheLogger, while queueBatchSize shows the size of each batch (based on the number of log items) in BatchLogger. In the second experiment, we examine the performance of Base, Light, and CacheLogger based on two case study systems with two disk technologies: executing Weka_TC4 and OE_TC2 on the SSD with the configuration mentioned in Section 4.3.3 and on the HDD with the configuration mentioned for the server in Section 5.3. Note that the JVM and AspectJ versions of two systems are the same. We specify the `cacheSize` capacity to its optimum value (i.e., 10,000) which should be obtained from the past experiment. In the third experiment, we execute TC4 and OE_TC2 test cases for Base, Light, and CacheLogger (set with optimum cache size) on the same hardware (with HDD disk) with different OSs: WindowsXP 64x (NTFS file format) and Ubuntu 12.04 64x (Ext4 file format). Note that for all experiments, there were no other executions other than test case executions, and antivirus and network were disabled during test case executions.

A.3.2.2 Results

Similar to past experiments, $Time_{SUS}$ shows the time span from the beginning until the end test case execution and $Time_{Log}$ shows the time required for logging after the completion of the test case execution. Table 24 shows that the performance of CacheLogger and BatchLogger (i.e., $Time_{SUS}$) is roughly the same and using two threads design in BatchLogger does not add to the overhead. Both loggers showed their best performance when the chosen value for `cacheSize` and `batchQueueSize` is in the vicinity of 10,000. Interestingly, the value of $Time_{Log}$ was very small for all cases in this

experiment. This indicates the efficiency of write to disk operation in new logging designs.

cacheSize/batch QueueSize		10	100	1k	10k	25k	50k	75k	100k	150k	200 k	300 k
CacheLogger	$Time_{SUS}$	6.75	7.23	7.18	6.88	6.92	6.97	7.03	7.10	7.25	7.44	7.72
	$Time_{Log}$	0.20	0.20	0.19	0.18	0.19	0.20	0.20	0.22	0.22	0.28	0.29
BatchLogger	$Time_{SUS}$	14.9	7.23	6.61	6.42	6.84	6.60	6.56	6.56	7.71	7.15	7.46
	$Time_{Log}$	0.37	0.24	0.23	0.25	0.24	0.22	0.23	0.23	0.24	0.26	0.27

Table 24 Average execution times (100 executions) in second for CacheLogger and BatchLogger

Table 25 shows tremendous performance enhancement for both case studies with SSD technology: CacheLogger implementation cut the overhead (i.e., $Time_{SUS}$) by more than half compared to the Light instrumentation. Results show this performance improvement on HDD disk is at least 35% compared to the Light execution on the same disk. The differences between execution times in Light and CacheLogger indicate that access to disk (i.e., disk look up) is an expensive operation for both disk technologies.

Time(Sec) / Test case	Disk techn.	Base	Light	CacheLogger(cacheSize=10k)	
				$Time_{SUS}$	$Time_{Log}$
OE_TC2	SSD	0.085	15.44 (%100)	6.88 (%-55)	0.18
	HDD	0.092	32.36 (%100)	18.87 (%-42)	0.94
Weka_TC4	SSD	0.373	17.51 (%100)	8.58 (%-51)	0.27
	HDD	0.457	37.53 (%100)	24.66 (%-34)	0.60

Table 25 Average execution times (100 executions) for Base, Light and CacheLogger on HDD and SSD

However, looking at execution times of the same test cases with different disk technologies in Table 25, show that an access to magnetic disk is more expensive than access to SSD due to mechanical nature of HDD technology.

Comparing TimeLogs for the same test cases executed on different OSs with same hardware configuration in the third experiment () shows that the OS file format can impact the logging mechanism in our instrumentation: the write operation in Windows is much more expensive than Linux. Therefore, Linux file format (i.e., Ext4) better suits for our instrumentation to perform the write operation.

Test case	OS	Disk techn.	Base	Light	CacheLogger(`cacheSize`=10k)	
					$Time_{SUS}$	$Time_{Log}$
OE_TC2	Ubuntu	HDD	0.092	32.36 (%100)	18.87 (%-42)	0.94
	Windows	HDD	0.203	39.07 (%100)	18.73 (%-52)	15.47
Weka_TC4	Ubuntu	HDD	0.457	37.53 (%100)	24.66 (%-34)	0.60
	Windows	HDD	0.547	47.657 (%100)	25.57 (%-46)	26.45

Table 26 Average execution times (100 executions) for Base, Light and CacheLogger based on different OSs

A.4 Conclusion

In this appendix, we conducted detailed experiments based on various aspects of instrumentation optimizations, naming: collecting information, encoding information and logging information. The results of these experiments become handy when we explain our overall instrumentation optimization approach in chapter 5.

Appendix B Method Call Counter Package

```
1   package count;

2   import java.util.*;
3   import NMC.*;

4   public aspect collectMethodCallCounter {

       //This counts all calls (static, non-static, constructor) happening
    within JDK/JRE
5      //+ AspectJ + NMC + instrument (or other possible third part library
6   except count package)
       //pointcut callMethod() : call(!static * *..*(..)) && !call(*
7   count..*(..));
       //pointcut callStaticMethod() : call(static * *..*(..)) &&
    !call(static * count..*(..));
8      //pointcut callConstructor() : call (*..new(..))&& !call
    (count..new(..));
9
       //This counts all calls due to instrument and NMC(PUS) packages.
10     //pointcut callMethod() : call(!static * NMC..*(..)) || call(!static *
    instrument..*(..));
       //pointcut callStaticMethod() : call (static * NMC..*(..)) || call
11  (static * instrument..*(..));
12     //pointcut callConstructor() : call (NMC..new(..)) || call
13  (instrument..new(..));

14     //This counts all calls due to NMC(PUS) package.
       pointcut callMethod() : call (!static * NMC..*(..));
15     pointcut callStaticMethod() : call (static * NMC..*(..));
16     pointcut callConstructor() : call (NMC..new(..));
17
       pointcut mainMethod() : execution (static * *..main(..));
18
19     before(): callMethod () {
20        counter.methodCallCounter();
       }
21
22     before(): callStaticMethod()  {
23        counter.methodCallStaticCounter();
       }
24
25     after(): callConstructor()    {
26        counter.constructorCallCounter();
27     }
28
29     after(): mainMethod()         {
          System.out.println("MethodCalls   " + counter.methodCallCounter);
          System.out.println("staticMethodCalls    " +
    counter.methodCallStaticCounter);
          System.out.println("ConstructorCalls     " +
    counter.constructorCallCounter);
       }
    }
```

Fig. 32 collectMethodCallCounter.aj

```
1   package count;
2
3   public class counter {
4
5       static int methodCallCounter = 0;
6       static int methodCallStaticCounter = 0;
7       static int constructorCallCounter = 0;
8
9       public static void methodCallCounter(){
10          methodCallCounter++;
11      }
12
13      public static void methodCallStaticCounter(){
14          methodCallStaticCounter++;
15      }
16
17      public static void constructorCallCounter(){
18          constructorCallCounter++;
19      }
20  }
21
```

Fig. 33 counter.java

Appendix C Weka Test Suite

```
1    #!/bin/bash

2    # This test suite contains 7 test cases
3    export
     CLASSPATH=$CLASSPATH:/home/hossein/devtools/commandLine_executions/Weka_
     orig/build/classes:/home/hossein/devtools/commandLine_executions/Weka_or
     ig/lib/packageManager.jar:/home/hossein/devtools/commandLine_executions/
     Weka_orig/lib/java-cup.jar:/home/mehrfard/aspectj1.6/lib/aspectjrt.jar

4    time for ((j=0 ; j<100; j++)) do
5    (
6    time (for (( i = 0 ; i < 1 ; i++ )) do

7       java weka.core.Instances data/soybean.arff >> out 2>&1
8       java weka.classifiers.rules.ZeroR -t data/weather.arff >> out 2>&1
9       java weka.classifiers.trees.J48 -t data/weather.arff >> out 2>&1
10      java weka.classifiers.meta.Stacking -B "weka.classifiers.lazy.IBk -K
     10" -M "weka.classifiers.meta.ClassificationViaRegression -W
     weka.classifiers.functions.LinearRegression -- -S 1" -t data/iris.arff -
11   x 2 >> out 2>&1
        java weka.core.WekaPackageManager -package-info repository
12   isotonicRegression >> out 2>&1
        java weka.filters.supervised.attribute.Discretize -i data/iris.arff -
13   o data/iris-nom.arff -c last >> out 2>&1
        java weka.filters.supervised.instance.Resample -i data/soybean.arff -
     o soybean-uniform-5%.arff -c last -Z 5 -B 1 >> out 2>&1

14
15      done)
16   ) >> WekaOrig_TS.txt 2>&1
17     rm -f soybean-uniform-5%.arff
       rm -f out
18
     done
19
     rm -f Trace.txt
```

Fig. 34 Weka_TS.sh

Appendix D Extending the hybrid approach to support C++

In this section, we present an extension of the hybrid approach to reverse engineer object interactions for C++ programs. We only discuss the dynamic analysis part the hybrid approach in this appendix since the static part is all about parsing and is relatively straight-forward. We will report on the static analysis part in the final dissertation.

The remainder of this section is organized as follows: first, we present the modified trace model for C++ programs in Section D.1; next, we explain the implementation of the instrumentation tool that captures dynamic information in Section D.2; we verify the correctness of this approach in Section D.3; finally, we conclude this chapter in Section D.4.

D.1 The modified trace model

As we mentioned in Section 2.1.3.3, the characteristics of the base programming language and the specification of the instrumentation language affects the dynamic analysis process. In the case of C++, the base programming language, language features lead us to slightly modify the trace model: adding the `deleteMessage` attribute to the `MessageSort` class (Fig. 35). The `deleteMessage` attribute signifies calls to the destructor at the end its life-time.

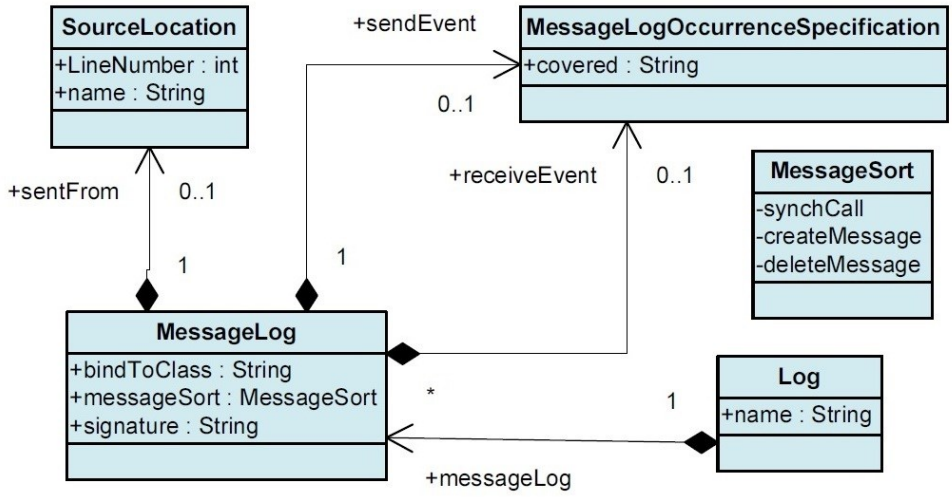


Fig. 35 C++ trace model

We use AspectC++ (AC++), the instrumentation language, to capture each model element at runtime [65]. Unlike AspectJ, the AC++ join point model does not provide any join point to intercept calls to constructors or destructors. Therefore, we need to intercept their executions instead. This, however, results in losing the caller's identity in the scenario diagram.

The former modification (adding attribute) causes changes in OCL mapping rules for transforming a Trace model instance to a UML scenario instance. We updated the send message rule (i.e., `mapSendMessage()` operation from the `Matching` class) that maps `MessageLog` in the Trace model to the `Message` in UML in Kolbah's mapping rules (Figure 30 in [25]). In the updated rule (Fig. 36), we add `deleteMessage` to UML scenario (lines 12, 13).

```
1    Matching :: mapSendMessage(m: Message, ml : MessageLog)
2    post : m.name = ml.signature
3       and
4       m.ownedComment.body =matchMessageLogToMethodCall(ml).isInClass
5        + "{"
6        + matchMessageLogToMethodCall(ml).lineNumber
7        + "}"
8       and
9       if ml.messageSort = MessageSort::createMessage
10      then m.messageSort = MessageSort::createMessage
11      else
12         if ml.messageSort = MessageSort::deleteMessage
13         then m.messageSort = MessageSort:: deleteMessage
14         else m.messageSort = MessageSort::synchCall
15      endif
16      and
17      m.connector.end->exists(end : ConnectorEnd |
18       Lifeline.allInstances->exists(ll |
19         Matching.mapSendLifeline(ll, ml)
20         and
21         end.role.type = ll.represents) )
22      and
23      m.connector.end->exists(end : ConnectorEnd |
24       Lifeline.allInstances->exists(ll |
25         Matching.mapReceiveLifeline(ll, ml)
26         and
27         end.role.type = ll.represents) )
28      and
29      Matching.mapSendLifeline(m.sendEvent.covered, ml)
30      and
31      Matching.mapReceiveLifeline(m.receiveEvent.covered, ml)
```

Fig. 36 Send message mapping rule

The latter modification, the absence of message sender to constructor (or destructor), does not impose any change to OCL mappings; nevertheless it results in missing the creator or destroyer of an object in the scenario diagram. Static analysis techniques (e.g., call graph, backward or forward slicing) can be applied to discover creator or destroyer of objects; however we do not apply such techniques for C++ programs in our work.

D.2 Implementing Light instrumentation with AspectC++

Our instrumentation contains two aspects (`MethodAspect`, `IdentifierAspect`) and two C++ classes (`Logger`, `IdGen`). The design of our instrumentation tool is based on `after` object execution advice and a hash map data-structure for object identification, that is the design we mentioned in Section 5.4.1.3. Note that we did not use the object

identification mechanism we used in the Light instrumentation with AspectJ since the `slice` mechanism for changing the static structure in AC++ (similar to inter-type declaration in AspectJ) does not allow to declare a new static method in a class. We briefly explain two aspects in this section as there are many similarities between the AC++ implementation and the AspectJ implementation. For instance, the implementation of the `IdGen` class (Fig. 22) we explained in Section 5.4.1.3 is very similar to the `IdGen` class in C++ (Fig. 22).

The `MethodAspect` (Fig. 37) contains six advices: two `before():call` advices are to capture method calls information, one `before():call` advice is to capture static method calls information (caller is static), one `after():execution` is to capture calls to destructor information, and one `after():execution` is to capture calls to constructor information. We used two advices for method call where in one advice we capture calls from a method to a non-static method (line 31) and in another advice we capture calls from a method to a static method (line 37).

```
1    aspect MethodAspect {
2
3       pointcut callMethod() = "% example::%::%(...)" && !"static %
4    example::%::%(...)";
5       pointcut callStaticMethod() = "static % example::%::%(...)";
6       pointcut executeConstructor() = "example::%";
7
8       advice construction(executeConstructor()): before(){
9          tjp->that()->id = 0;
10         string className = typeid(tjp->that()).name();
11         className = className.substr( 3, className.size());
12         className = className.replace(className.find("1"), 1, "::");
13         tjp->that()->idName = className;
14      }
15
16      advice construction(executeConstructor()): after(){
17         .
18         .
19      }
20
21      advice destruction(executeConstructor()): before(){
22         .
23         .
24      }
25
26      advice call(callStaticMethod()) : before(){
27         .
28         .
29      }
30
31      advice call(callMethod()) && that(ptr) && target(ptr1) : before(void*
32   ptr, void* ptr1){
33         .
34         .
35      }
36
37      advice call(callMethod()) : before(){
38         .
39         .
40      }
41   }
```

Fig. 37 Excerpt of MethodAspect

The `IdentifierAspect` (Fig. 38) alters the structure of each class in the SUS based on the `slice` mechanism by adding three variables (`idName`, `objectID`, `id`) and a `getObjectID()` method to that class. Once the object is called, the `before` execution advice in `MethodAspect` (line 8) initializes `id` to zero and `idName` to the name of class. Then, after construction (line 16) or before destruction (line 21) of each object, those two execution advices call the `getObjectID()` method to get the value of `objectID`. The `getObjectID()` method calls `IdGen` and passes class name as a hash map key to return

the value of `objectID`. The `IDGen` class stores a map of class names and number of objects created. `IDGen` provides a centralized mechanism for creating `objectID`s. Note that we used a generic implementation for the logger, whereby it writes the passed information from advices to disk every time it is called.

```
1    aspect IdentifierAspect  {
2
3       pointcut objects() = "packageName::ClassName1" ||
     "packageName::ClassName2" | ...;
4
5       advice objects() : slice class {
6
7                    private:
8          string objectID;
9          int id;
10         string idName;
11
12         public:
13         string getObjectID() {
14            if( id  == 0 ){
15                int nID = IDGenerator::instance().next(idName);
16                stringstream s;
17                s << idName.c_str() << "_" << nID;
18                objectID = s.str();
19                id++;
20            }
21            return objectID;
22         }
23      };
24   };
```

Fig. 38 IdentifierAspect

D.3 Case study

We translated the OE case study (recall Section 4.2) from Java into C++. We do not explain programming details of this translation since discussing such programming language features is out of the scope of this work. This new case study should aid us to verify our AC++ instrumentation.

D.3.1 Verifying the correctness of generated traces

We verified the correctness of our instrumentation by comparing traces generated from two instrumentation mechanisms: based on Light instrumentation of OE in Java and C++. We repeated this verification for two test cases: OE_TC1 and OE_TC2. Our

analysis shows that when we accounted for all the differences due to language syntax (e.g., dots versus double colons for name domain separator) or instrumentation limitation (e.g., missing object creator in C++ traces), generated traces were the same for each case study. This shows our AC++ instrumentation captures object interaction correctly.

D.3.2 Executing instrumented OE

We executed the non-instrumented version (which we refer to as Base) and Light C++ instrumentation for 100 executions to calculate the average time. We used the Linux `time` command to measure the time (in second). Table 27 shows the Light C++ instrumentation is very costly compared to Base. We need to apply the optimization process that we mentioned in the previous chapter on the Light C++ instrumentation (developed with AC++). In addition, we need to apply this instrumentation tool on a larger size case study to have a better understanding about the performance of our instrumentation tool.

	Base	Light
OE_TC2	0.05895	17.52802

Table 27 Average execution time of Base and Light instrumentations in C++

D.4 Conclusion

We explained our contribution on extending the hybrid approach to C++. We showed that our instrumentation can capture correct object interactions by comparing traces of the same program which was implemented both in C++ and Java.