UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of

Informatics, Systems and Communications

PhD program: Informatics Cycle XXXI

# In-The-Field Monitoring of Interactive Applications

Surname: Cornejo Olivares Name: Oscar Eduardo

Registration number: 811032

Tutor: Prof. Francesca Arcelli-Fontana

Supervisor: Prof. Leonardo Mariani

Dr. Daniela Briola

Prof. Daniela Micucci

Coordinator: Prof. Stefania Bandini

**ACADEMIC YEAR 2018/2019**

# Abstract

Monitoring techniques can extract accurate data about the behavior of software systems. When used in the field, they can reveal how applications behave in real-world contexts and how programs are actually exercised by their users. However, the collection, processing, and distribution of field data must be done seamlessly and unobtrusively while users interact with their applications.

To limit the intrusiveness of field monitoring a common approach is to reduce the amount of collected data (e.g., to rare events and to crash dumps), which, however, may severely affect the effectiveness of the techniques that exploit field data.

This Ph.D. thesis investigates the trade-off between field monitoring and the degradation of the user experience in interactive applications, that is, applications that require user inputs to continue its operations. In particular, we identified two big challenges: to understand how the user perceives monitoring overhead and to study how to collect data in a non-intrusive way without losing too much information.

In brief, we provide three main contributions. In the first place, we present an empirical study aimed at quantifying if and to what extent the monitoring overhead introduced in an interactive application is perceived by users. The reported results can be exploited to carefully design analysis procedures running in the field. In particular, we realized that users do not perceive significant differences for an overhead of 80% and seldom perceived an overhead of 140%.

Secondly, we introduce a monitoring framework for deriving comprehensive runtime data without affecting the quality of the user experience. The technique produces a finite state automaton that shows possible usages of the application from the events observed in the field. From the model, it

is also possible to extract accurate and comprehensive traces that could be used to support various tasks, such as debugging, field failures reproduction and profiling.

Finally, we present a strategy to further reduce the impact of monitoring by limiting the activity performed in parallel with users' operations: the strategy delays the saving of events to idle phases of the application to reduce the impact on the user experience. The approach considerably decreases the impact of monitoring on user operations producing highly accurate traces.

The results obtained in this Ph.D. thesis can enable a range of testing and analysis solutions that extensively exploit field data.

# Contents

**Abstract** **3**

**1 Practical Field Monitoring** **15**
1.1 Monitoring Software in the Field . . . . . . . . . . . . . . . . . . 15
1.2 Field Monitoring Challenges . . . . . . . . . . . . . . . . . . . . . 17
1.3 On the Perception of the Monitoring Overhead . . . . . . . . . 18
1.4 Main Contributions of this Thesis . . . . . . . . . . . . . . . . . . 19

**2 A User Study About the Impact of the Overhead on the Users 21**
2.1 Operation Categories: Perception of Delayed System Response Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.2 Preliminary Study . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2.1 Results of the Preliminary Study . . . . . . . . . . . . . 25
2.3 Experiment Setting . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.3.1 Goal and Research Questions . . . . . . . . . . . . . . . 27
2.3.2 Hypotheses, Factors and Treatment Levels . . . . . . . 27
2.3.3 Response Variables and Metrics . . . . . . . . . . . . . . 28
2.3.4 Experimental subjects . . . . . . . . . . . . . . . . . . . 29
2.4 Experiment Design . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.4.1 Full Design . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.4.2 Design Decisions . . . . . . . . . . . . . . . . . . . . . . 32
2.4.3 The Final Design . . . . . . . . . . . . . . . . . . . . . . 33
2.4.4 Experimental Objects . . . . . . . . . . . . . . . . . . . . 34
2.4.5 Experiment Procedure . . . . . . . . . . . . . . . . . . . 35
2.4.6 Data Analysis . . . . . . . . . . . . . . . . . . . . . . . . 36
2.5 Experiment Results . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.5.1 RQ1 - Impact of Overhead Levels . . . . . . . . . . . . . 40
2.5.2 RQ2 - Order of Operations . . . . . . . . . . . . . . . . . 43

2.5.3 Threats to validity . . . . . . . . . . . . . . . . . . . . . . 46
2.6 Findings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
2.7 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**3 Controlled Burst Recording 51**
3.1 Inferring Knowledge from Field Executions . . . . . . . . . . . 51
3.1.1 Overview of the Technique . . . . . . . . . . . . . . . . . 53
3.2 *Controlled Burst Recording* Framework . . . . . . . . . . . . . 57
3.2.1 Step 1: Classes Identification . . . . . . . . . . . . . . . 58
3.2.2 Step 2: Function Extraction . . . . . . . . . . . . . . . . 58
3.2.3 Step 3: Training Phase . . . . . . . . . . . . . . . . . . . 62
3.2.4 Step 4: Bursts Collection . . . . . . . . . . . . . . . . . . 66
3.2.5 Step 5: *FSA* Synthesis and Traces Simulation . . . . . 67
3.3 Empirical Validation . . . . . . . . . . . . . . . . . . . . . . . 74
3.3.1 Goal and Research Questions . . . . . . . . . . . . . . . 74
3.3.2 Experiment Design . . . . . . . . . . . . . . . . . . . . . 75
3.3.3 Experimental Subject . . . . . . . . . . . . . . . . . . . . 76
3.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
3.4.1 Performance Results . . . . . . . . . . . . . . . . . . . . 79
3.4.2 Precision Results . . . . . . . . . . . . . . . . . . . . . . 80
3.4.3 Recall Results . . . . . . . . . . . . . . . . . . . . . . . . 82
3.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**4 Delayed Saving 87**
4.1 Prioritizing User Interactions . . . . . . . . . . . . . . . . . . . 87
4.2 Delayed Saving Parameters . . . . . . . . . . . . . . . . . . . . 91
4.2.1 CPU Usage Percentage . . . . . . . . . . . . . . . . . . . 91
4.2.2 Sampling Frequency of the CPU . . . . . . . . . . . . . . 92
4.2.3 Threshold level of the CPU . . . . . . . . . . . . . . . . 92
4.3 The Delayed Saving Process . . . . . . . . . . . . . . . . . . . 93
4.3.1 Step 1: Monitor Generation . . . . . . . . . . . . . . . . 93
4.3.2 Step 2: Training . . . . . . . . . . . . . . . . . . . . . . . 94
4.3.3 Step 3: Field Monitoring . . . . . . . . . . . . . . . . . . 97
4.4 Empirical Validation . . . . . . . . . . . . . . . . . . . . . . . 98
4.4.1 Goal and Research Questions . . . . . . . . . . . . . . . 99
4.4.2 Experiment Design . . . . . . . . . . . . . . . . . . . . . 99
4.4.3 Experimental Subject . . . . . . . . . . . . . . . . . . . . 101

4.4.4 Training Step . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.5 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
4.5.1 Performance Results . . . . . . . . . . . . . . . . . . . . . 104
4.5.2 Quality of Data Results . . . . . . . . . . . . . . . . . . . 107
4.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

**5 Software Monitoring 111**
5.1 Perception of the System Response Time . . . . . . . . . . . . . 111
5.2 Field Monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.2.1 Distributive Monitoring . . . . . . . . . . . . . . . . . . . 116
5.2.2 Probabilistic Monitoring . . . . . . . . . . . . . . . . . . . 117
5.2.3 State-Based Monitoring . . . . . . . . . . . . . . . . . . . 119

**6 Summary of Contributions and Open Research Directions 121**
6.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
6.2 Open Research Directions . . . . . . . . . . . . . . . . . . . . . 123

# List of Figures

2.1 Actions classifications: on the left, to classify actions with no overhead, on the right when monitoring is on. Blue for Instantaneous, black for Immediate. . . . . . . . . . . . . . . . 24
2.2 Percentage of slow actions for different overhead intervals. . . 25
2.3 Experiment overview . . . . . . . . . . . . . . . . . . . . . . . 30
2.4 Normality plot for fractional model . . . . . . . . . . . . . . . . 38
2.5 Mean PRT of each operation in each Task for different overhead levels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.6 Interaction between Operation Category and OH for Task 1 and 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.7 Interaction between Operation Category and Task for different OHs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
2.8 Interaction between Operation Category and Task Order . . . 44
2.9 Significant Interactions between Task, Task Order, and Overhead . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

3.1 Overview of *Controlled Burst Recording* approach. . . . . . . . 53
3.2 Function Extraction process. . . . . . . . . . . . . . . . . . . . . 59
3.3 Example of two executions collected in the field. . . . . . . . . 63
3.4 Vertical concatenation of executions $E^1$ and $E^2$. . . . . . . . . 64
3.5 Example of filtering process of *AF*. . . . . . . . . . . . . . . . . 65
3.6 Example of simulation of traces. . . . . . . . . . . . . . . . . . 68
3.7 *FSA* node with state information and incoming and outgoing transitions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.8 Recall validation process. . . . . . . . . . . . . . . . . . . . . . . 70
3.9 Example of resulting *FSA* . . . . . . . . . . . . . . . . . . . . . 72
3.10 *Controlled Burst Recording* Validation. . . . . . . . . . . . . . . 75

3.11 Activity Diagram example in ArgoUML . . . . . . . . . . . . . 78
3.12 Precision results with respect to different runs of the application. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
3.13 Recall results with respect to different runs of the application. 84

4.1 Time performance comparison when saving to memory/file. . . 89
4.2 Comparison of performances for different monitoring approaches. 90
4.3 Delayed Saving Process . . . . . . . . . . . . . . . . . . . . . . . 93
4.4 Timing Classification problem. . . . . . . . . . . . . . . . . . . . 95
4.5 Using Delayed Saving in the field. . . . . . . . . . . . . . . . . . 98
4.6 Example of DS traces. . . . . . . . . . . . . . . . . . . . . . . . . 100
4.7 Threshold selection using the Timing Classification procedure. 102
4.8 Performance results for Objects level. . . . . . . . . . . . . . . . 104
4.9 Performance results for Objects + Attributes level. . . . . . . . . 105
4.10 Performance results for Objects + Attributes Recursively level. 106
4.11 Quality traces comparison for Objects level. . . . . . . . . . . . 107
4.12 Quality traces comparison for Objects + Attributes level. . . . 108
4.13 Quality traces comparison for Objects + Attributes Recursively level. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

# List of Tables


2.1 Factors and Treatment Levels . . . . . . . . . . . . . . . . . . . . 29
2.2 Students' background and characteristics. . . . . . . . . . . . . . 30
2.3 Adopted design and setup. . . . . . . . . . . . . . . . . . . . . . . 34
2.4 Type III Tests of Fixed Effects . . . . . . . . . . . . . . . . . . . . 40
2.5 Results interaction Task, Operation Category, and OH by comparing different Operation Categories. . . . . . . . . . . . . . . . 41
2.6 Results interaction Task, Operation Category, and OH by comparing different OH levels. . . . . . . . . . . . . . . . . . . . . . 42
2.7 Results interaction Overhead, Task, and Task Order. . . . . . 45

3.1 Precision for each node in the *FSA* . . . . . . . . . . . . . . . . . 73
3.2 Recall of each trace generated by the *FSA* . . . . . . . . . . . . 74
3.3 Monitoring Target and Symbolic Execution Outcome . . . . . 77
3.4 Details from the abstraction function filtering process. . . . . 78
3.5 Performance results with respect to system response time categories. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.6 Precision results for *Controlled Burst Recording*. . . . . . . . . . 81
3.7 Recall results for *Controlled Burst Recording*. . . . . . . . . . . 82

4.1 Time performance comparison when saving to memory/file. . . 88
4.2 Selection of optimal threshold level of the CPU using timing classification procedure. . . . . . . . . . . . . . . . . . . . . . . . 102
4.3 Selection of optimal threshold level of the CPU using Behavioral Classification procedure. . . . . . . . . . . . . . . . . . . . . 103

5.1 Shneiderman's and Seow's SRT categorization . . . . . . . . . 112

# Listings

3.1 Shopping Cart Class . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.2 Functions for method addItem . . . . . . . . . . . . . . . . . . . . . 60
3.3 Functions for method applyDiscount . . . . . . . . . . . . . . . . 60
3.4 Functions for method calculateTotal . . . . . . . . . . . . . . . . 60
3.5 Functions for method emptyCart . . . . . . . . . . . . . . . . . . 61
3.6 Burst 1: clickOnAddItem . . . . . . . . . . . . . . . . . . . . . . . 71
3.7 Burst 2: clickOnAddItem . . . . . . . . . . . . . . . . . . . . . . . 71
3.8 Burst 3: clickOnPay . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.9 Burst 4: clickOnStartNewSession . . . . . . . . . . . . . . . . . . 72
3.10 Burst 5: clickOnStartNewSession . . . . . . . . . . . . . . . . . . 72
3.11 Trace 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
3.12 Trace 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
3.13 Trace 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

# Chapter 1

# Practical Field Monitoring

The presentation of this chapter is organized as follows. Section 1.1 presents the key role played by field monitoring to support software testing and analysis. Section 1.2 discusses the open challenges in field monitoring. Section 1.3 discusses the relation between field monitoring and user experience. Section 1.4 presents the main contributions of this Ph.D. thesis work and the organization of this document.

## 1.1 Monitoring Software in the Field

In order to give a better understanding about monitoring software in the field we first need to give a formal definition of what a monitor is within the context of this Ph.D. thesis: according to *WordNet*[1], 6th interpretation, a monitor is *a piece of electronic equipment that keeps track of the operation of a system continuously and warns of trouble*. This definition remarks the fact that a monitor should capture and observe events, but also must process the information acquired by the piece of software/hardware. The Runtime Verification (RV) community conceives a monitor as a piece of software/hardware that observes some behavior and checks its correctness with respect to some property [6]. Differently, in the Software Engineering community, a monitor is *a piece of software that dynamically collects information about a program's behavior, but do not performs any active control to search for problems* [10]. In this Ph.D. thesis we adopt the definition according to the Software Engineering community, however, the

[1] https://wordnet.princeton.edu

findings of this work and the proposed solutions might very useful for the RV community as well.

Fully assessing the robustness of software applications in-house is infeasible, especially if we consider the huge variety of hardly predictable stimuli, environment and configurations that applications must handle in the field.

Field monitoring techniques are useful because they can extract accurate data about the behavior of software systems. When used in the field, they can reveal how applications behave in real-world environments and how programs are actually used by their users, capturing scenarios and configurations relevant to practitioners, and how failures manifest and impact user activity [31].

It is also well known that modern approaches to software development consider the distinction between design-time and run-time as less and less obvious [5]. For instance, DevOps extensively uses automation and monitoring tools to reduce the gap between development and operation [69], while several analysis solutions exploit the operational environment as a testbed to analyze and test software [3, 32, 37].

The importance of observing the software when running in the field has been already well recognized by industry and academia. For instance, the video streaming company Netflix [55] has reported that its system has grown so much that performing realistic testing in-house is nearly impossible, so it has started testing and collecting data directly in the field, using fault-injection and monitoring [8].

Indeed quality control activities such as testing [32, 31, 56] cannot be limited to in-house verification and validation but must cross organization boundaries spanning the field. Not only testing techniques are used in the field. But other functionalities, such as crash reporting [24, 51, 20], are extensively present in commercial and open source software.

Bug isolation techniques may also greatly benefit from the volume of data that can be automatically extracted from the field [48, 40]. For instance, field data may help profiling applications [27], improving code coverage [60, 67], discovering and reproducing failures [41, 15], supporting debugging tasks [61] and controlling software evolution [65].

Not only software engineering techniques take advantage of field data, also runtime verification approaches exploit the collected data on the end-user environment to verify the correctness of the runtime behavior of the

system with respect to some property [34, 6].

However when the application is interactive, since monitoring might need significant storage and computational resources, it may interfere with users activities degrading the quality of the user experience [63, 18]. While simple crash reporting requires taking a snapshot of the system at the time of the crash [24, 51, 20] other solutions require monitoring applications more extensively.

For instance, collecting the sequences of function calls produced by a monitored system, and shipping the recorded information to developers are activities that interfere with regular executions. Indeed, they might introduce an annoying overhead and cause unacceptable slowdowns. Since preventing any interference with the user activity is a mandatory requirement, for both real applications and research tools to be exploited by real end users, monitoring techniques should address the challenge of collecting field data by reducing the amount of collect data.

## 1.2 Field Monitoring Challenges

Actual techniques address the challenge of decreasing monitoring overhead by limiting the amount of information we might obtain from remote instances.

For instance, the set of collected events can be reduced in several ways: (1) can be limited arbitrarily by selecting a part of the system to be analyzed considerably reducing the necessary instrumentation [67, 2, 64, 42]. (2) Can be also distributed among multiple instances of a same application running on different machines, where each instance will be in charge of monitoring a different aspect of the analysis [10, 65]. (3) Can be determined probabilistically, where each instrumentation point runs only with a certain probability [7, 40, 14, 48]. (4) Can be limited by optimizing the placement of instrumentation in field executions [60, 53]. (5) And it can be also limited by collecting bursts of events, instead of monitoring full executions [35, 59].

In the runtime verification process the efficiency and unobtrusiveness of collecting data from field is a fundamental stage. By definition monitors should keep overhead low [6], especially when monitors are required to be executed in parallel with the system under analysis (e.g., online and synchronous monitors).

The problem of cost-effectively monitoring field data might be further complicated by the cost of saving the individual events. Most of the techniques cited above focus on tracing *simple information*, such as the name of an invoked method, but there is little research about cost-effectively tracing *expensive events*, such as the value of data structures and the state of the application, which may require saving runtime information about several objects and variables.

The main objective of this Ph.D. work is *to cost-effectively collect field data, which may include expensive information for some of the collected events, without affecting the end user experience*.

## 1.3 On the Perception of the Monitoring Overhead

It is clear that running analysis activities (e.g., profiling, testing, and program analysis) in parallel with user activities may annoy users [63]. Since the available resources are shared between all processes running in a same environment, the analysis processes may introduce slowdowns in the user processes, negatively affecting their experience. This is particularly true for *interactive applications*, that is, applications that continuously interact with users. Addressing the intrusiveness of monitoring implies getting a better and deeper understanding of when users get annoyed due to monitoring overhead.

There exist studies about the absolute perception of time [46, 70], about the capability to perceive small deviations in durations [46], and studies investigating the impact of slowdowns in specific situations, especially while browsing the Web [38, 54]. Unfortunately, *little is known about the actual perception of any slowdown introduced in a program*, and thus embedding analysis strategies in software programs without annoying the users can be challenging.

Identifying to what extent users may recognize slowdowns is extremely important because it can be used to quantify the amount of resources that can be consumed by any additional analysis process embedded in a program running in the field.

## 1.4 Main Contributions of this Thesis

This Ph.D. thesis provides three main contributions:

1. **A study of the impact of the overhead on the user experience**

   This Ph.D. thesis presents an empirical study specifically aimed at *quantifying if and to what extent the overhead introduced in an interactive application is perceived* by users. Since a user might execute different kinds of operations of different complexity and different duration while the system is slowed down, ranging from opening a menu to submitting a complex query, we expect that the perception of the overhead may depend on the nature of the operation that is executed and on its context.

   For this reason, we consider multiple *categories of operations* and sequences of operations executed in different *order*, investigating how both factors may influence the perception of monitoring overhead (Chapter 2).

2. **A strategy to sample executions to avoid exposing users to significant overhead**

   In this Ph.D. work we present a monitoring framework that can derive comprehensive runtime data without affecting the quality of the user experience.

   Instead of limiting the size and number of events observed from the field, the idea is to vary the timing of monitoring. For example, a monitor might be turned on and off several times during the program execution in order to collect *pieces* of traces, that might be merged in a second step to produce more complete knowledge about software behavior.

   We thus collect *bursts* of executions, that is, partial traces with no internal gaps, annotated with state information that captures the state of the monitored application at the beginning and end of the burst.

   Although bursts include only partial information about a monitored execution, they are complete when restricted to their time interval (i.e., every monitored event produced by the application after the first event of the trace fragment and before the last event of the trace fragment also occurs in the burst).

These bursts can be simply obtained by activating and deactivating a monitor that records every relevant event. Since the monitor is used intermittently, its impact on the user experience is limited.

The different collected bursts will be used to produce a Finite State Automaton that shows how the monitored application has been used in the field.

Since this approach, namely *Controlled Burst Recording*, does not record full executions we might loss the full ordering of field events, but still a good representation of the behavior of the application can be given.

We assess our approach in a restricted, although common, scenario, that is, *recording the sequence of function calls produced by an application, annotated with parameters and state information*.

Many techniques exploit this type of field data, such as techniques for reproducing failures [41], profiling users [27], Finite State Automaton synthesis from traces [50], and controlling software evolution [65] (Chapter 3).

3. **A strategy to effectively save the captured data**

   Since field monitoring may slowdown the reactiveness of an interactive application, our idea is to delay some operations of the monitoring process to idle time to not impact directly on the user experience.

   Since the process of saving the collected data to a persistent memory could be extremely expensive, the idle state of an application can be exploited to decrease the probability of generating excessive levels of overhead whenever the user is interacting with the monitored application. With this third contribution, we are recording very limited amount of information in parallel with user activity and further decreasing the impact on the user experience.

   Since this approach is based on postponing the time when data is saved, it may present the risk of losing data. However, the empirical results show that the problem is relatively frequent and it can be a reasonable penalty to pay compared to the benefit of recording applications in the field (Chapter 4).

# Chapter 2

# A User Study About the Impact of the Overhead on the Users

This chapter presents an empirical study specifically aimed at *quantifying if and to what extent the overhead introduced in an interactive application is perceived* by users. Since a user might execute different kinds of operations of different complexity and different duration while the system is slowed down, ranging from opening a menu to submitting a complex query, we expect that the perception of the overhead may depend on the nature of the operation that is executed and on its context. For this reason, we consider multiple *categories of operations* and sequences of operations executed in different *order*, investigating how both factors may influence the perception of the overhead.

The presentation of the study is organized as follows. Section 2.1 presents the categorization of the operations that we used in our study. Section 2.2 presents a preliminary study we conduct to investigate the possible overhead levels to use in the actual experiment. Sections 2.3 and 2.4 present the setting and the design of our experiment, respectively. Sections 2.5 and 2.6 discuss the obtained results and findings. Finally, Section 2.7 presents some final considerations about the user study.

## 2.1 Operation Categories: Perception of Delayed System Response Time

The study we carried out in this chapter places in the Human Computer Interaction area, in particular it focuses on the system response time [72, 71, 70] and its evaluation from the end-user point of view [23, 46, 13, 54, 57, 52, 38]. Even though in Chapter 5 we give more details about these concepts, we anticipate that our study as far as we could find in the literature, it is the first one set up in this way, that is understanding the users' reaction to the delay of the single actions in the system response time.

How the overhead is perceived is likely to be dependent on the kind of operation that is affected by the overhead. For instance, users may perceive differently a delay on the opening of a menu compared to a delay on the processing time of a complex query. For this reason we explicitly considered the type of operation as a factor in the study.

To classify operations we relied on existing studies from the human computer interaction community. In particular, there are some interesting studies [70, 71, 72] that correlate the nature of the operation to its expected system response time (SRT), that is, the time elapsed between the user request and the response of the application. These studies classify operations in four categories with minor differences on the quantification of the SRT associated with each category. For the purpose of our study, we used the categorization defined by Seow [70] because he designed his study considering the interaction between the users and the computer as a conversation, that is consistent with the behavior of interactive software applications like the ones we considered.

We thus used the following categories:

- *Instantaneous*: these are the most simple operations that can be performed on an application, such as entering inputs or navigating throughout menus (SRT: 200ms at most).

- *Immediate*: these are operations that are expected to generate acknowledgments or very simple outputs (SRT: 1s at most).

- *Continuous*: these are operations that are requested to produce results within a short time frame to not interrupt the dialog with the user (SRT: 5s at most).

- *Captive*: these are operations requiring some relevant processing for which users will wait for results, but will also give up if a response is not produced within a certain time (SRT: 10s at most).

This categorization is complete for our purpose because interactive applications seldom have operations requiring more than 10s to be completed and this is also true in our study.

## 2.2 Preliminary Study

We are interested in investigating how the overhead can affect the user experience.

As stated before, the impact on the user experience can directly depend on the kind operation that users perform. However, the particular overhead level introduced by a certain monitor can also be a determinant factor and thus, it could be interesting to understand *how progressively slowing down a system may affect the user experience*. We propose to investigate the correlation between the different overhead levels and the user experience through a preliminary study. The overhead levels will be useful for the human-subjects study presented in the Section 2.3.

For this preliminary study we selected seven widely used interactive programs of different sizes and complexity: MS Excel 2016, MS Outlook 2016, Notepad++ 6.9.2, Paint.NET 4.0.12, Winzip 20.5, and Adobe Reader DC 2015. The monitoring activity consisted in collecting sequences of function calls from the different applications, in particular we instrumented the software using a probe implemented with the Intel Pin Binary Instrumentation tool [39]. The probe can be configured to use buffers of different sizes to store data in memory before flushing data into a file.

To run each application, we have implemented *Sikulix* [36] test cases that can be automatically executed to cover a typical usage scenario.

Each test case includes from 11 to 32 user actions, with a mean of 16 user actions per test. To assess the impact of the monitor, we measured the *overhead* and we estimated its effect on the *user experience*. To accurately investigate both factors, we collected data at the granularity of the individual actions performed in the tests. That is, if a test case executes actions $a_1 \dots a_n$, we collect data about the overhead and its impact on the user experience for each action $a_i$ with $i = 1 \dots n$.

We collected data for both the application without the probe and the application instrumented with our probe configured with buffers of different sizes: 0MB (data is Immediately flushed to disk), 1MB, 25MB, 50MB, 75MB, 100MB, and 200MB. Experiments have been executed on a Windows 7 - 32bit machine equipped with 4GB of RAM. Each test has been repeated 5 times and mean values have been used to mitigate any effect due to the non-determinism of the execution environment. Overall, we collected near 4,000 samples about the execution time of the actions.

While the overhead can be measured as the additional time consumed by an application due to the presence of the monitor, it is important to discuss how we estimated the effect of the monitor on the user experience.

In principle, assessing if a given overhead may or may not annoy users requires direct user involvement. However, since the aim of this preliminary study is to get an intuition about the different overhead classes that we could use in the human-subjects study, we thus rely directly in the Seow's System Response Categorization (i.e., Instantaneous, Immediate, Continuous and Captive categories).

We attribute categories to actions based on their execution time when no overhead is introduced in the system. We use the lower limit of each category to this end. For instance, actions that take at most $100ms$ are classified as Instantaneous, while actions that take more than $100ms$ but less than $1s$ are classified as Immediate.

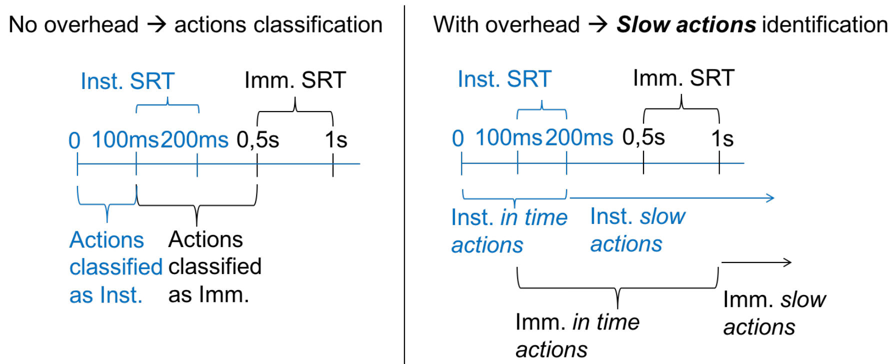



**Figure 2.1:** *Actions classifications: on the left, to classify actions with no overhead, on the right when monitoring is on. Blue for Instantaneous, black for Immediate.*

We thus estimated the expected impact of the overhead on the user experience by measuring the number of *slow actions*, that is, the actions that

exceed the upper limit of the response time for their category once affected by the overhead. According to this classification, we assumed that the response time of an action is acceptable by users as long as it is below the upper limit of the category the action belongs to. Thus, an overhead that increases the response time of an action without exceeding the upper limit of the category (e.g., an Instantaneous action that takes less than $200ms$ once affected by the overhead) would be hardly noticeable by users. On the contrary, if the overhead increases the response time of an action above the upper limit of the category (e.g., an Instantaneous action that takes more than $200ms$ once affected by the overhead), the execution time of the action would likely violate the user expectation, and the slowdown would be recognizable by the users. Figure 2.1 shows the previous example of classification.

### 2.2.1 Results of the Preliminary Study

In this section, we report the results that we obtained in our preliminary study.

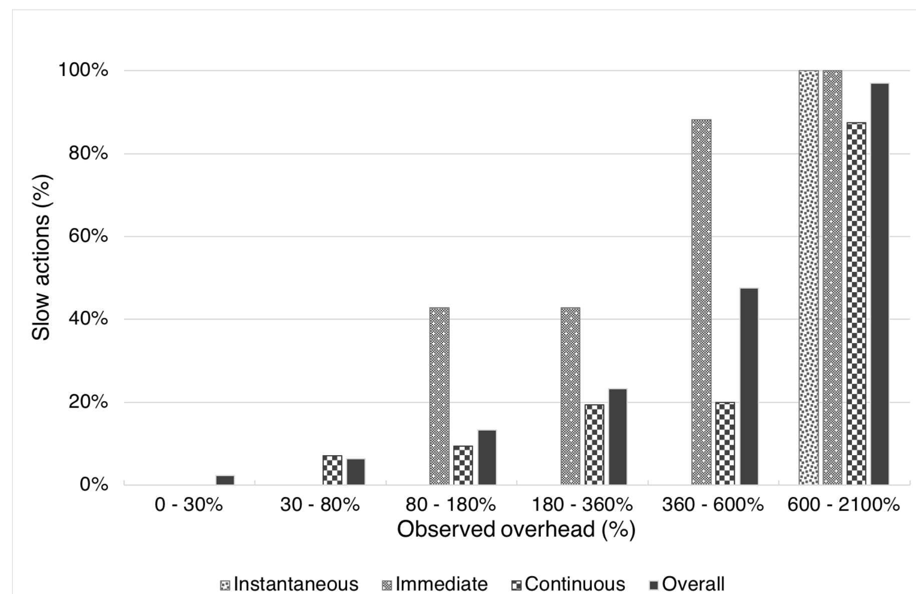



**Figure 2.2:** *Percentage of slow actions for different overhead intervals.*

We compute the percentage of slow actions distinguishing among Instantaneous, Immediate and Continuous actions. We plot these percent-

ages in Figure 2.2, considering overhead intervals that produce similar results. We also indicate the overall percentage of slow actions in each overhead interval. We do not plot the results for Captive actions because we had only few actions belonging to this category in the tests, thus the collected data are insufficient to produce relevant insights. However, the Captive actions have been included in the computation of the overall percentage of slow actions for each overhead interval.

We can first observe that an overhead up to $30\%$ never caused a noticeable slowdown for any kind of action. This result is partially in contrast with the intuition that only a very small overhead could be tolerated in the field.

An overhead in the range $30 - 80\%$ makes only a few actions exceed the maximum response time for their category. In particular, only Continuous actions show slowdowns that are likely to be perceived by users, while the overall percentage of slow actions is low. This suggests that simple computations can be safely monitored compared to complex operations, which require more attention.

An overhead in the range $80-180\%$ turns $13\%$ of the actions into slow actions in average. Almost all categories of actions are affected, again with a greater potential impact on the most complex operations. Higher overhead values ($> 180\%$) turns $20\%$ into slow actions. Note that the plot reports a significant effect on the Instantaneous actions only when the overhead is greater than $600\%$. This is probably due to the simplicity of the actions, which are still fast even for high percentage overhead.

These results show that a non trivial overhead (e.g., up to $30\%$) can likely be introduced in the field with little impact on the users. This result is coherent with the study described in [46], which shows that users are usually unable to identify time variations smaller than $20\%$.

In a nutshell, through the study we identified different intervals of overhead, where each interval has a certain effect on the perception of actions of the System Response Time Categorization.

## 2.3 Experiment Setting

Based on the operation categories and, the overhead values analyzed in the previous sections, we now present the human-subjects study aimed to understand how the different overhead levels and the order of actions of

the system response time may impact the user experience.

In this section, we provide detailed information about the experiment goals, experimental subjects and research questions, according to Juristo and Moreno [44]. The next sections discuss the specific design that we adopted.

### 2.3.1 Goal and Research Questions

The goal of this experiment is to evaluate if and when users perceive delays in the SRT for different operation categories, as discussed in Section 2.1. The study is conducted under the perspective of software developers and designers interested in investigating how much overhead can be introduced in the applications without annoying users. This quantification is useful to design appropriate monitoring, analysis and testing procedures running in the field.

In particular, our study is driven by two research questions.

**RQ1:** *Which overhead level may affect the user experience?*

This research question investigates if and when users recognize the overhead that affects a software application.

**RQ2:** *Can the order of execution of the functionalities offered by an application affect the user experience?*

This research question investigates whether the perception of the overhead depends on the order of execution of the functionalities.

### 2.3.2 Hypotheses, Factors and Treatment Levels

To answer RQ1, we analyze how users respond to *different levels of overhead* considering multiple *types of operations*.

We use the overhead values from the preliminary study presented in Section 2.2. Although the results were obtained without involving human subjects in the experiment, the study identifies $30\%$ and $80\%$ as interesting overhead values that may produce different reactions by users. Since Killeen and Weiss [46] states that users are usually unable to identify time variations smaller than $20\%$, we rather use $20\%$ instead of $30\%$ to be more aligned with the literature.

To have an additional point of reference we also consider an overhead of $140\%$, and we include the value $0\%$ in the study to have a baseline to compare to.

To cover different types of operations, we selected the four categories of operations described in Section 2.1.

RQ1 generates multiple null hypotheses to be tested: $H_{XY}$: *users tolerate equally all overheads when introduced into different operation categories*. While the alternative hypotheses are: $H_{\overline{XY}}$ :*users do not equally tolerate all overheads when introduced into different operation categories*. The values of Overheads ($X$) and Operation Categories ($Y$) are $\{0\%, 20\%, 80\%, 140\%\}$ and {Instantaneous, Immediate, Continuous and Captive}, respectively.

To answer RQ2 we considered how the order of execution of different functionalities may affect the perception of time. For instance, running a functionality that takes time followed by a functionality that terminates quickly may make the execution of the latter functionality to appear faster than it is in reality, and this may also affect the perception of the overhead.

Since the execution of a non-trivial functionality typically requires the user to perform a task consisting of a sequence of simple operations (e.g., to browse across windows) followed by an expensive operation, we studied this dependency in the context of short tasks ending with the execution of a meaningful functionality. Based on Seow's categorization (see Section 2.1), this implies working with tasks composed of Instantaneous and Immediate operations ending with either a Continuous or Captive operation.

The null hypothesis tested to address RQ2 is: $H_0$ :*The order of execution of tasks ending with Continuous and Captive operations does not influence the user perception of time.* While the alternative hypothesis is: $H_1$ :*The order of execution of tasks ending with Continuous and Captive operations influences the user perception of time.*

In addition to the factors discussed so far, the outcome of the experiment may depend on the specific tasks that are executed during the experimental sessions. For this reason, we also include the concrete task that is executed as a factor.

Table 2.1 summarizes the considered factors and treatment levels.

### 2.3.3 Response Variables and Metrics

The response variable represents the effect of the different factors involved in the experiment [44]. Both research questions RQ1 and RQ2 require a variable that can measure the user experience as response variable [44].

**Table 2.1:** *Factors and Treatment Levels*

| Factors | Treatment Levels |
|---|---|
| Overhead level | 0%, 20%, 80%, 140% |
| Operation Category | Instantaneous, Immediate, Continuous, Captive |
| Task Order | Continuous-Captive, Captive-Continuous |
| Concrete Task | The specific tasks used in the study |

We choose the *response time* as the variable to measure our research questions, since we are going to introduce slowdowns that affect directly the system response time. As we are analyzing how users perceive this property, we call it *Perceived Response Time* (*PRT*).

We measure *PRT* using a Likert scale, where participants can express their perception with respect to the system response time. This scale goes from "Too slow" to "Too fast" divided in five levels, with the third level corresponding to "Normal". To be sure that participants are able to evaluate if the application is responding differently, they should be aware of "which the normality is": for this reason, as described better later on, we selected an application used by the subjects during their academic course, so that they all have a reference SRT.

The response variable is measured after executing a task with the subject program, so that the users can actually express their opinion about the system they just interacted with.

### 2.3.4 Experimental subjects

The subjects of the experiment are bachelor students of the first year of Computer Science degree at the University of Milan-Bicocca. To replicate the case of a user who interacts with a *known* application affected by overhead, we selected Eclipse, since it is the IDE regularly used by the subjects during their classes.

We recruited a total of 48 students who completed a demographic questionnaire before of the actual experiment. Table 2.2 shows some characteristics and background of the participants.

Eclipse experience indicates the number of students who have already worked with Eclipse IDE, and specifically with the version we used for the experiment (Eclipse Mars). Note that almost the entire population of students have used Eclipse. The size of the developed project indicates the kind of projects students have already developed. All of them developed

**Table 2.2:** *Students' background and characteristics.*

| **Eclipse experience** | **Eclipse** | **Eclipse Mars** | | | | |
|---|---|---|---|---|---|---|
| Number of students | 47 | 31 | | | | |
| **Size of developed project (# classes)** | **0** | **1-10** | **11-30** | **30-50** | **50-100** | **>100** |
| Number of students | 0 | 0 | 32 | 15 | 1 | 0 |

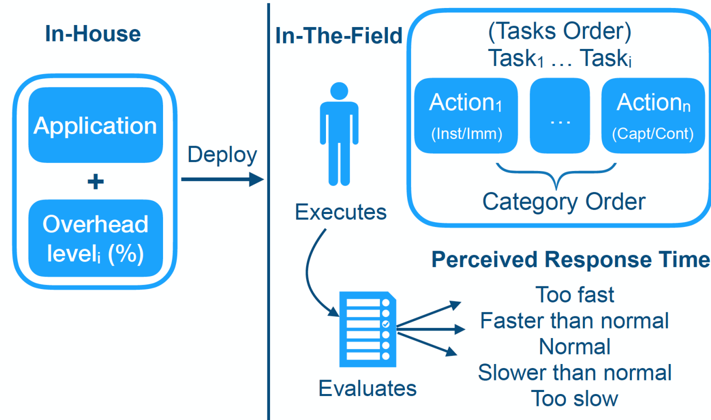



**Figure 2.3:** *Experiment overview*

projects with tens of classes, denoting some familiarity with the Java language.

The average age of the participants is 20.8 years and most of them declared to have more than 10 years of experience using interactive applications. According to these characteristics, the subjects can be considered to be representative of a population of young but experienced users of interactive software applications.

## 2.4 Experiment Design

Figure 2.3 shows the high-level structure of our experiment. Each participant interacts with an application performing some tasks. The application used by the participant is instrumented to respond according to configured overhead level (0% in the case of the control group). Once a task is completed, the participant evaluates the quality of the interaction in terms of the experienced SRT.

This high-level procedure can be instantiated in multiple ways. In the next sections we describe the full design and discuss why it is clearly infea-

sible to be adopted: then we describe the limitations to control the factors and their combinations until reaching the final adopted design.

### 2.4.1 Full Design

When defining the tasks that the subjects should complete, if we limit the design to tasks with one operation per SRT category (i.e., tasks with four operations), we should in principle consider enough tasks to cover every combination of SRT categories and order of operations inside the task, that is, $4! = 24$ possible tasks: this is prohibitive for multiple reasons. Using 24 tasks implies planning for very long laboratory sessions where a same subject has to complete a large number of operations, with the major risk of collecting inaccurate data once the level of attention starts dropping down. Alternatively, the tasks can be split among subjects, but this requires recruiting a very large number of participants to obtain statistically significant results.

Moreover, each task can be instantiated with multiple concrete tasks (for example, Eclipse implements multiple Continuous operations). Considering multiple ways of mapping an operation of a given SRT category into a concrete operation (that is, mapping a task into a concrete one), would make the overall set of concrete tasks to be used for the experiment to growth further. It is thus mandatory to work with a small set of tasks, so that the activity of the subjects involved in the experiment will be limited in time.

Besides studying the impact of a certain level of overhead on the user experience, it would be interesting to analyze the effect of variations on this factor, that is, considering the order of the overhead values a subject is exposed to as a factor. Since we consider 4 possible overhead values, all the possible overhead orders over a sequence of four operations is a set of $4^4 = 256$ combinations. Again, this number of combinations is impossible to study in practice.

In the next section, we present the design decisions we made to obtain a workable design.

### 2.4.2 Design Decisions

**Limiting categories order within a task**

Not all combinations of operations are relevant in practice. Users typically interact with applications by executing an initial sequence of short operations, such as clicking on menu items and browsing between windows, to reach a functionality of interest, followed by the execution of the target functionality, which is often a rather expensive operation. It is thus important to consider tasks composed of fast operations at the beginning and a slow operation at the end. This is also consistent with our practical experience on the definition of tasks in Eclipse where it was almost impossible to reach an expensive operation (a Continuous or Captive operation) without first executing a sequence of fast operations (an Instantaneous or Immediate operation) from any state of the system.

These considerations led us to define *two main types of tasks* to be used for our study. A task with some short operations at the beginning (both Instantaneous and Immediate) and a Continuous operation at the end, and another task also with short operations at the beginning but a Captive operation at the end. The number of short operations depends on the activity that must be performed to reach the functionality executed at the end of task.

**Limiting task instances**

The two types of tasks that we identified can be mapped into many concrete tasks by choosing different concrete operations of the right SRT category each time. To keep the study small, while considering more than one option for each SRT category, we consider *two concrete tasks for each type of task*, obtaining four concrete tasks.

**Fixing overhead order within a task**

Our tasks are composed of few operations executed sequentially, with a total duration of approximatively 10 seconds. While considering how subjects react to dynamically changing values of the overhead is interesting, it makes little sense to consider this in the scope of so short tasks. We thus decided to focus the study on the perception of a *constant percentage of the overhead* in the context of short tasks, leaving the investigation of

how a dynamically changing overhead may influence the user experience for future work.

#### Limiting task order

As discussed in Section 2.3.2, we are mainly interested in studying the effect of the order of execution of Tasks with Continuous and Captive operations on the perception of the overhead. Since we have four concrete tasks, two with Continuous operations and two with Captive operations, we can think of many different ways of arranging these four tasks one after the other. However, since the main focus is studying the switch from a Continuous to a Captive operation and vice versa, we focus on two main task orders: executing the tasks with Captive operations first and Continuous operations next, and vice versa. Furthermore, since the focus is on the switch from the task with the Captive operation to the task with the Continuous operation and vice versa, we do not consider multiple orders within the sequence of tasks of the same type, that is, we consider a same order for the two tasks with the Captive operations and a same order for the two tasks with the Continuous operations.

#### Fixing overhead order between tasks

To study overhead order, each subject has necessarily to execute multiple tasks. In principle, each task could be exposed to a different overhead. However, considering different overhead orders across tasks, in addition to producing an infeasible set of overhead orders to be studied, would interfere with one of the objectives of the experiment that is studying the effect of moving from tasks with more expensive functionalities (Captive) to less expensive functionalities (Continuous) and vice versa. In fact, when switching from a task to another, in addition to changing the category of the last operation that is executed, we would also change the overhead. For this reason, we keep the *overhead constant for all the tasks executed by a same subject*.

### 2.4.3 The Final Design

Based on the objective of the study and the design decisions discussed in Section 2.4.2, we have clear constraints on the experiment. In particular,

the study has to cover:

1. Four overhead values
2. Four standard SRT Categories
3. Two types of tasks (quick operations followed by a Continuous operation and quick operations followed by a Captive operation)
4. Two task instances per task
5. Two task orders (Executing two tasks ending with a Captive operation followed by two tasks ending with a Continuous operation, and vice versa)

**Table 2.3:** *Adopted design and setup.*

| **Task Order** | **Applied OH** | | | |
|---|---|---|---|---|
| | **0%** | **20%** | **80%** | **140%** |
| Tasks with Continuous, then Tasks with Captive | G1 | G3 | G5 | G7 |
| Tasks with Captive, then Tasks with Continuous | G2 | G4 | G6 | G8 |

The resulting design is shown in Table 2.3. The subjects are distributed in eight groups G1-G8: groups G1, G3, G5, G7 first perform the two tasks terminating with a Continuous operation and then the two tasks terminating with a Captive operation, while groups G2, G4, G6 and G8 do the opposite. This allows to study the impact of task order on the perception of the overhead. In addition to Continuous and Captive operations, the tasks include enough Instantaneous and Immediate operations (see description of the tasks in next subsection) to cover all combinations of SRT and overhead.

Each group works with a same overhead value for all the performed tasks. Since the design is *Between Overhead Values*, that is, each group will perform all the tasks (and consequently all the categories) with the same overhead, we might confuse the effect of the overhead with the effect of the group. To mitigate this risk we assigned people to groups randomly, avoiding any bias in the definition of the groups.

### 2.4.4 Experimental Objects

The experimental objects are the tasks that the subjects perform during the experiment, reflecting some of the different treatments proposed in the

design [44]. In our case, the experimental objects are the four concrete tasks that the participants to the experiment have to perform in different order.

To design these four tasks, we first identified the operations implemented in Eclipse that are either Continuous or Captive according to Seow's classification, and we also identified the shortest sequence of operations that must be performed to reach the identified operation and run it. Among these tasks, we finally selected the cases with the most balanced presence of operations of the other categories. In the ideal case, we would like to have one Instantaneous and one Immediate operation in each task. However, this was not always possible, and we had to tolerate the presence of two Instantaneous operations in two tasks.

The resulting four tasks are defined as follows:

- *Task1 - Clean&Build*: click on *Project* menu and wait for the sub menu being visualized (Instantaneous), click on *Clean* and open the dialog window (Immediate), and click on *Ok* to start the *Clean&Build* operation, and wait for the progress window to be automatically closed (Captive)
- *Task2 - Search*: open the search window by clicking on the keyboard *Ctrl + H* (Immediate), click on *Java Search* tab (Instantaneous), enter the string *C** (Instantaneous), and click on *Ok* to start the search and wait for the automatic closing of the dialog window (Captive)
- *Task3 - Type Hierarchy*: enlarge the project tree with a double click (Instantaneous), select the *src* package with one click (Immediate), and press *F4* to start the creation of the Type Hierarchy (Continuous)
- *Task4: Sort Members*: click on the arrow near the name of the project to expand its structure (Instantaneous), select the *working* package with a click (Immediate), click on the *Source* menu (Instantaneous), and click on *Sort Members* and wait for the end of the sorting operation (Continuous)

The total set of operations executed by performing the four tasks are six Instantaneous operations, four Immediate operations, two Continuous operations, and two Captive operations.

### 2.4.5 Experiment Procedure

The experimental procedure resembles the general structure represented in Figure 2.3 and the detailed design shown in Table 2.3. In particular,

all the participants to the study have been invited to join an experimental session in a lab that we prepared for the purpose of the experiment. To have a balanced number of people in each group, we asked the students to subscribe for the experimental session in advance. We then randomly distributed the students among the 8 groups to have 6 people per group.

The lab session started with a profiling questionnaire, whose main results are reported in Table 2.2. We then passed an instruction sheet giving general information about the structure of the experiment and a general description of the tasks that will be performed with Eclipse. Each participant performed the four tasks presented in Section 2.4.4 but in a different order (tasks 3, 4, 1, 2 for the odd groups and tasks 1, 2, 3, 4 for the even groups) and exposed to different overhead (see Table 2.3). Subjects were not told about the specific aim of the experiment, nor about the fact that tasks had overheads.

Each participant incrementally accessed the four sheets describing the specific tasks to be performed with Eclipse, that is, the sheet with the description of the next task was accessible only once the previous task has been completed. Each task is carefully described with text and screen-shots to avoid any misunderstanding. Once a task is completed and before moving to the next task, the participant evaluated the perceived response time of each operation that has been executed according to five possible levels: "Too slow", "Slower than Normal", "Normal", "Faster than Normal", "Too Fast".

The subjects finally compiled an exit questionnaire to check whether the performed operations were clearly explained.

To expose each instance of Eclipse to the right overhead, we used AspectJ and Equinox Weaving [26] to inject an aspect that altered the SRT of each executed operation based on the group of the participant.

### 2.4.6 Data Analysis

There is controversy about the correctness of analyzing data measured in Likert scale with parametric tests, as it is an ordinal scale [68]. However, recent studies show that they are sufficiently robust [58]. There are two options for analyzing a repeated measures design: a repeated measures ANOVA and a mixed linear procedure. We cannot use a repeated measures ANOVA because it requires all the tasks to have one operation for each SRT

category, which is not the case in our design. We thus used a *mixed linear procedure*.

A mixed linear procedure can deal with SRT Categories not being represented in all the tasks (Continuous operations in Tasks 1 and 2, and Captive operations in Tasks 3 and 4), but it cannot deal with multiple operations of the same SRT category in the same task, as happening for Instantaneous operations in Task 2 and 4. So, we ran two analyses using either the first or the second Instantaneous operation (only for tasks with multiple operations of the same SRT category), and we obtained exactly the same significant interactions in both cases: we thus considered the first operation only, discarding the second one.

When applying the mixed linear procedure, we examined six models: a random effects model with no repeated measures, in which we represent the grouping variable, in this case the subjects, as a random effect; and repeated measures models with the repeated effect Task and Operation Category, where we explored five different types of covariance: Identity, Compounded Symmetry, Diagonal, AR(1), Unstructured. We chose the model that best fits our data, as described later on.

The mixed model requires that the residuals are approximately normal distributed. The Kolmogorov-Smirnov and Shapiro-Wilk tests cannot be used due to our sample size: with large sample sizes ($> 100$), they easily get significant results for small deviations from normality. Therefore, a significant test does not necessarily tell us whether the deviation from normality is enough to bias any statistical procedures that we apply to the data [30]. So, we check normality with probability plots: if the residuals of the model fit the normal line, we can proceed assuming that the error terms are normally distributed.

The normal probability plot of the full factorial model (all four main effects plus all possible two, three, and four-way interactions) revealed severe normality issues.

In complex models with several factors and interactions, it has been observed that the normality issues can be corrected by removing some of the higher-order interactions. In this case, we decided to remove the fourth-order interaction, and to analyze data using a fractional factorial model.

We compared the considered over mentioned six models using the Akaike Information Criterion (AIC), which is an estimator of the relative quality of a statistic model for a given set of data. In particular, AIC is a measure of

fit that penalizes models for having additional variables [30]. Smaller AIC values indicate a better fit: we observe that the repeated-measures model with unstructured covariance is the model with lowest AIC, and thus the best model.

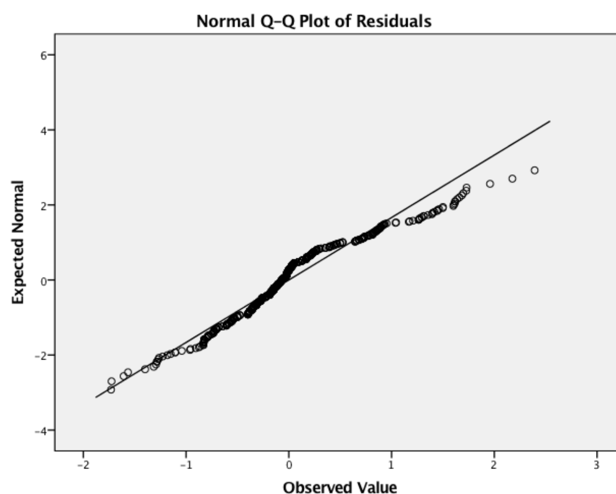



**Figure 2.4:** *Normality plot for fractional model*

We also checked the normality of the residuals for the selected model. The corresponding plot is shown in Figure 2.4, and we can see that residuals show a good fit to the normal line. Although the fitting is not perfect, it is adequate since the statistical test we are using is robust to slight deviations from normality [33].

We thus finally selected the *repeated-measures* model with *unstructured covariance*.

Besides analyzing the significance, we would like to study the existing strength (usually called *effect size*) between the given treatments. We measure the strength of a correlation using the Spearman's $\rho$ index, which returns a value between $0$ and $1$: values between $0 - 0.1$ indicate a negligible strength, values between $0.1 - 0.3$ indicate a small strength, values between $0.3 - 0.5$ indicate a medium strength, and values greater than $0.5$ indicate a high strength [16].

## 2.5 Experiment Results

Figure 2.5 shows how the participants have classified the SRT of each category of operation when exposed to different overhead levels. It is possible to visually notice how the SRT is generally perceived as normal for all the categories and overhead levels, with the exception of Captive operations that are classified as slower than normal already when they are exposed to no overhead, and their perceived SRT decreases for higher overhead levels.

**Figure 2.5:** *Mean PRT of each operation in each Task for different overhead levels*

We analyzed this visual trend and the rest of the factors controlled in our experiment distinguishing statistically significant effects and their effect size. We used IBM SPSS Version 24 [19] for the analysis of the data.

Table 2.4 shows the significance (column *p-value*) of each factor and their interactions (column *Source*).

The rows in gray correspond to interactions or main effects that are not significant (according to a significance level equals to $0.05$).

The table must be read from high-order interactions (i.e., the three-way interactions reported at the top of the table) to low-order interactions (i.e., the main effects reported at the bottom of the table). When a $n$-way interaction is included into a higher-order interaction, the $n$-way interaction can be ignored since it is subsumed by the higher-order interaction. In our case, we have three interactions (two three-way interactions and one two-way interaction) that subsume all the other interactions and effects. We indicated them in bold.

Table 2.4: *Type III Tests of Fixed Effects*

| Source | p-value |
|---|---|
| ***Overhead * Task * Task Order*** | ***0.008*** |
| ***Overhead * Task * Operation Category*** | ***0.018*** |
| Overhead * Task Order * Operation Category | 0.161 |
| Task * Task Order * Operation Category | 0.201 |
| ***Task Order * Operation Category*** | ***0.005*** |
| Task * Operation Category | 0.003 |
| Task * Task Order | 0.008 |
| Overhead * Operation Category | 0.195 |
| Overhead * Task Order | 0.003 |
| Overhead * Task | 0.000 |
| Operation Category | 0.000 |
| Task Order | 0.022 |
| Task | 0.001 |
| Overhead | 0.306 |

Table 2.4 shows that: (`Int-1`) the combination of the *Overhead*, the specific *Task* that is executed and the *Task Order*; (`Int-2`) the *Overhead*, the specific *Task* that is performed and the *Operation Category*; (`Int-3`) the combination of the *Task Order* and of the *Operation Category* of the operations that are executed, have an influence on the perceived response time. In this section, we refer to these significant interactions to present the *specific cases* that produce a *significant effect* on the response variable together with their *strength*, referring to the two research questions that we investigate with our study.

### 2.5.1 RQ1 - Impact of Overhead Levels

#### Captive Operations Are Particularly Sensitive to Overhead

Operations that require users to wait for the result are particularly critical and must be exposed to overhead carefully. In our experiments, this was true for Captive operations. Table 2.5 reports the cases where the operation is significant in the interaction between tasks, operations categories and overhead values (`Int-2`).

Figure 2.6 shows how operations from different tasks have been classified by users restricted to significant cases. Even when not affected by any overhead (yellow square), users classified the SRT of Captive operations below normal. This differs significantly from the evaluation of the other operations, such as the Instantaneous operations in task 1 and both Instantaneous and Immediate operations in task 2, classified as normal. We remind that Captive operations are present in tasks 1 and 2 only. This

**Table 2.5:** *Results interaction Task, Operation Category, and OH by comparing different Operation Categories.*

| Task | OH(%) | Op. Category (i) | Op. Category (j) | Sig. | Spearman's $\rho$ |
|---|---|---|---|---|---|
| 1 | 0 | Instantaneous | Captive | 0.020 | 0.187 |
| 1 | 80 | Instantaneous | Captive | 0.000 | 0.093 |
| 1 | 80 | Immediate | Captive | 0.000 | 0.057 |
| 1 | 140 | Instantaneous | Captive | 0.000 | 0.322 |
| 1 | 140 | Immediate | Captive | 0.000 | 0.000 |
| 2 | 0 | Instantaneous | Captive | 0.013 | 0.134 |
| 2 | 0 | Immediate | Captive | 0.028 | 0.289 |
| 2 | 140 | Immediate | Captive | 0.028 | 0.331 |
| 3 | 0 | Immediate | Instantaneous | 0.000 | 0.541 |

reveals a slightly bad attitude of the users in dealing with operations that naturally require more than 5 seconds to complete.

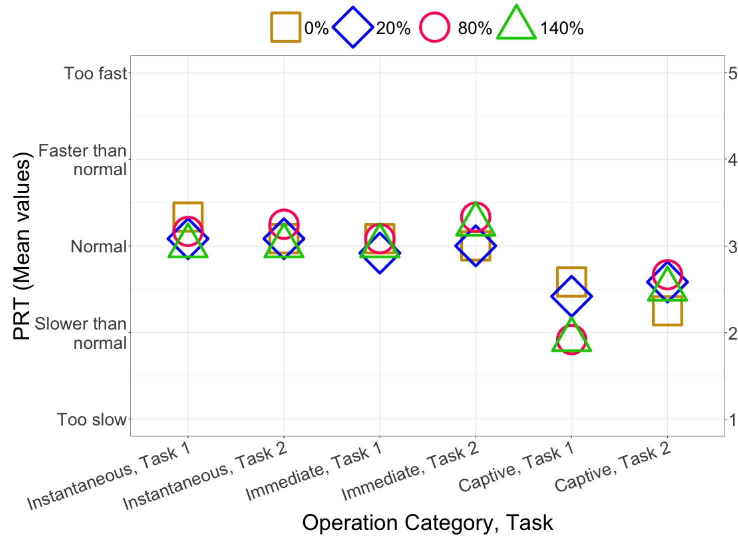



**Figure 2.6:** *Interaction between Operation Category and OH for Task 1 and 2.*

This difficulty in accepting that complex operations require some time to complete increases when a significant overhead is introduced. In fact, when the overhead is 80% (red circle) and 140% (green triangle), Captive operations are reported to be "Slower than normal", while Instantaneous and Immediate operations in tasks 1 and 2 remain normal. Although not every combination of Captive operations, tasks, and overhead value is significant, there is evidence that the overhead must be introduced carefully on Captive operations because the user is reluctant to accept functionalities that require more than 5 seconds to compute, even if this cost is justified.

**Table 2.6:** *Results interaction Task, Operation Category, and OH by comparing different OH levels.*

| Task | Op. Category | OH% (i) | OH% (j) | Sig. | Spearman's $\rho$ |
|---|---|---|---|---|---|
| 3 | Immediate | 80 | 140 | 0.045 | 0.460 |
| 4 | Immediate | 0 | 140 | 0.025 | 0.408 |

### Immediate Operations Are Sometime Sensitive to Overhead

We analyzed how users react to different overhead levels for the operations belonging to the various categories and tasks (`Int-2`). Figure 2.7 shows how users perceived a different SRT for the Immediate operations present in tasks 3 and 4 once exposed to an overhead of 140% (Table 2.6 reports the significant cases). Although Immediate operations typically consist of navigation actions, the users perceived the presence of an overhead when their SRT is more than doubled, while it was not the case for the other operation categories. We interpret this as a sign that introducing overhead in navigation actions can be done safely only up to a certain level, otherwise the decreased responsiveness of the application may annoy users.

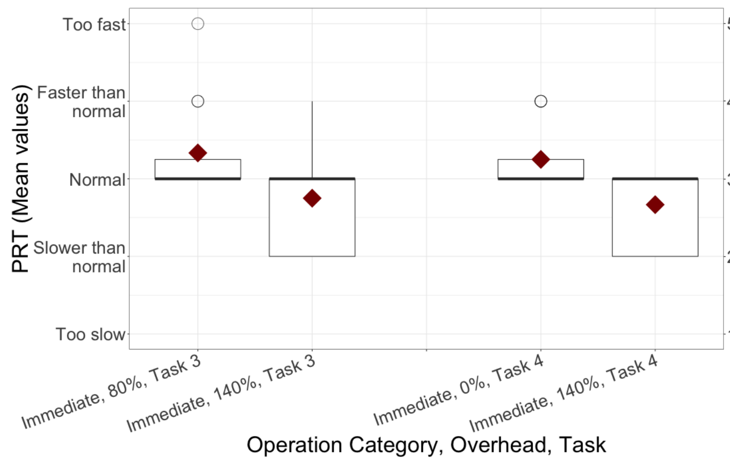



**Figure 2.7:** *Interaction between Operation Category and Task for different OHs.*

### The Specific Nature of an Operation Matters

Our study shows that there are specific situations that may break general rules, and thus the specific nature of the executed operation is often relevant.

For example, the SRT of the Instantaneous and Immediate operations in task 3 has been perceived differently even when the operations are not exposed to an overhead (see Table 2.5, last row). We attribute this behavior to the nature of the operation. The Instantaneous operation in task 3 corresponds to double-clicking on the Project menu, and finishes when the user sees the project tree expanded. The Immediate operation corresponds to clicking on the folder name that the user wants to analyze. In this case the operation finishes when the folder appears highlighted, which might be sometime difficult to recognize.

#### Users Hardly Perceive Important Overhead Levels if Introduced for a Limited Time

This is probably the most relevant result. Out of the many cases where the SRT was increased with an overhead up to 140%, only in few cases the users perceived an increased SRT. The statistical analysis revealed that only for two Immediate operations exposed to 140% overhead users reported a significant difference in the perception of the SRT (see Table 2.6). In addition to this specific case, there is a general negative attitude with Captive operations that is present already when there is no overhead, as illustrated in Figure 2.5. In all the other cases, the users evaluated the operations as running normally.

This suggests that, contrarily to the common belief that even little overhead may annoy users, in reality users seldom detect overhead up to 140%, as long as it is introduced for a limited number of operations. Indeed, our experiment considers tasks of up to 4 operations with only one operation causing a significant computation. We do not know how users would react to the same overhead level if preserved for a higher number of operations.

### 2.5.2 RQ2 - Order of Operations

#### The Context Influences the Perceived SRT

In the context of RQ2, we studied if the order of execution of Tasks ending with Continuous and Captive operations, which generally correspond to the execution of application domain functionalities, might have an impact on the PRT. The analysis of the interactions that include the *Task Order* as a factor (`Int-1` and `Int-3`) revealed multiple cases where the order

of execution of Continuous and Captive operations has an impact on the perceived SRT. In particular, the interaction between the *Operation Category* and the *Task Order* (`Int-3`) is significant for Continuous operations (p-value = $0.003$, $\rho = 0.062$). Although the strength is negligible, we quickly present this result because it is confirmed, with higher strength, for other operations and tasks as discussed later.

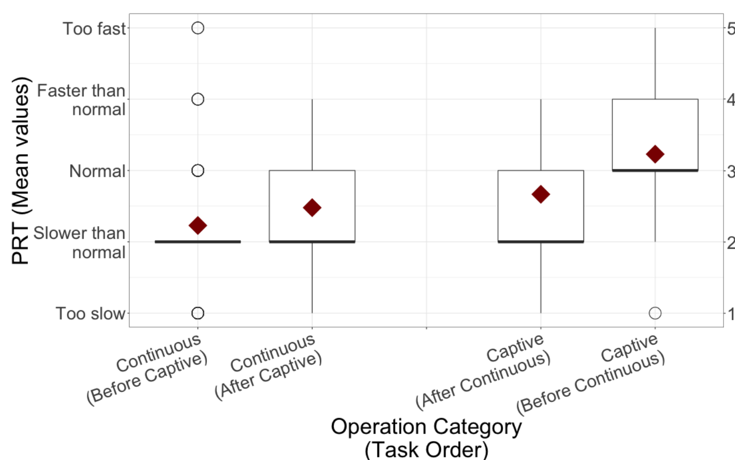



**Figure 2.8:** *Interaction between Operation Category and Task Order*

Figure 2.8 shows how Tasks ending with Continuous and Captive operations can be perceived differently if executed before or after the task including the operation of the other kind. In particular, a Continuous operation is perceived faster if executed after a task ended with a Captive operation (second box) rather then before (first box).

That is, the execution of a long operation generates a pessimistic expectation for the future, resulting in an optimistic perception of the response time if the functionality that is later executed is faster than the previous one. Although the effect has not been reported as significant, Captive operations are also perceived better if executed before a task ended with a Continuous operation (fourth box) rather than before (third box).

When considering the impact of Task Order on the three-way interaction `Int-1`, we found that the PRT is affected by the order of execution of the tasks in the following significant cases, also reported in Table 2.7: *i)* task 1 when the overhead is 0% and in Task 2 when the overhead is 20%, with negligible strength; *ii)* task 4 when the overhead is 20%, with small strength; *iii)* tasks 3 and 4 when the overhead is 80%, with small

and medium strength, respectively; *iv)* task 3 when the overhead is 140%, with medium strength.

**Table 2.7:** *Results interaction Overhead, Task, and Task Order.*

| Task | OH(%) | T. Order(i) | T. Order(j) | Sig. | Spearman's $\rho$ |
|---|---|---|---|---|---|
| 1 | 0 | Cont. then Capt. | Capt. then Cont. | 0.005 | 0.002 |
| 2 | 20 | Capt. then Cont. | Cont. then Capt. | 0.001 | 0.093 |
| 3 | 80 | Capt. then Cont. | Cont. then Capt. | 0.004 | 0.294 |
| 3 | 140 | Cont. then Capt. | Capt. then Cont. | 0.041 | 0.408 |
| 4 | 20 | Capt. then Cont. | Cont. then Capt. | 0.004 | 0.211 |
| 4 | 80 | Capt. then Cont. | Cont. then Capt. | 0.004 | 0.394 |

These results show that in several practical cases the perception of the SRT depends on the context, regardless of the overhead.

How users perceive the SRT in the cases with non-negligible strength is graphically illustrated in Figure 2.9.

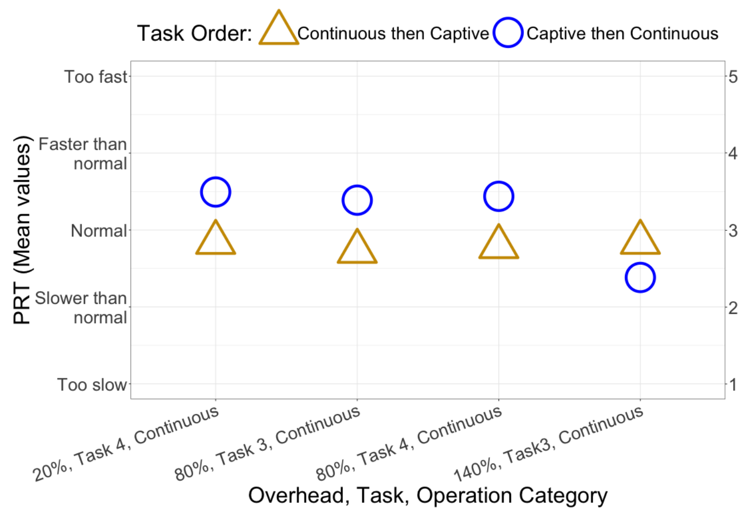



**Figure 2.9:** *Significant Interactions between Task, Task Order, and Overhead*

We can notice how the first operation that is executed (e.g., a Captive operation) systematically influences the long lasting operation executed in the following task (e.g., a Continuous operation) setting up an expectation that can be satisfied or violated depending on the specific order of execution. There is an exception for Task 3 with 140% overhead, where the evidence goes in the opposite direction. This is likely due to users who might be so annoyed by the bad responsiveness of the system that they run out of patience once exposed to two Captive operations with an overhead of 140%.

At this point, they might not be willing to accept any other long lasting operation, such as a Continuous operation with 140% overhead, regardless of its context of execution.

These results suggest that any overhead should be wisely introduced taking the context into account. For instance, if a Captive operation is followed by a Continuous operation, the user might be more likely to accept a higher overhead on the Continuous operation, as long as the overhead is not too high, because the Captive operation influenced the expectation of the user. Vice versa, if the order of the two operations is exchanged, the user might be unlikely to accept any overhead on the Captive operation executed after the Continuous one.

### 2.5.3 Threats to validity

We now discuss the threats to the validity of our study.

The main threat to *Conclusion Validity* concerns the possible lack of causality for the significant relationships that have been identified. To mitigate this risk we considered the strength of the reported evidence in the discussion of the results. Moreover, to mitigate the risk that an interference is produced by the organization in groups of the participants, we assigned people to groups randomly.

The main threats to *Construct Validity* concern the way the overhead has been introduced in the software and the clarity of the tasks performed by the subjects. We controlled the overhead introduced in the software by carefully designing the aspects based on the performance of the machines in the lab used for the experiment. Note that all the machines have exactly the same hardware and software. We tested the consistency of the behavior of our software on every machine of the lab. We also asked the participants to indicate if the activity they performed was clear in the exit questionnaire: 96.3% of the answers were positive and only 3.7% of the participants found some unclear aspects. We can thus assume that the values collected for the response variable were accurate.

The *Internal Validity* threats have been addressed in Section 2.4 with the careful discussion of the design decisions that motivated our final design.

The main threat to the *External Validity* is the representativeness of the participants and the application we used. The students we recruited acted

as regular computer users, so their student status is not likely to represent a threat on the external validity of the experiment. On the other hand, it is known that the perception of time can be different for young people, adults, and elderly people [23]. Our study is thus representative of the perception of young people, and we cannot generalize to other classes of users.

We studied the perceived SRT for interactive applications (e.g., Eclipse) covering multiple categories of operations. While the results are informative on how the overhead is perceived for computer applications, they are not guaranteed to hold for Web applications, where the network may play a relevant role on the SRT, and mobile applications, where the different hardware and interaction modalities may have an impact. Additional studies are necessary to extend our findings to these other domains.

## 2.6 Findings

In this section we summarize the our main findings and discuss their impact on analysis strategies working in the field.

**Users are unlikely to recognize important overhead levels if introduced for a limited number of interactions**. Indeed users have been able to perceive the variations in the SRT in a few cases only. This implies that *applications can be safely slowed down for as much as 80% and often up to 140% of their regular SRT* to support analyses running in the field. This however can be done for interactions of limited length (3 or 4 operations). We do not know if this overhead can be maintained longer without having the users noticing it. This result opens to the design of families of monitoring and analysis solutions that can *opportunistically consume significant resources* for a limited amount of time.

**Users might be quite sensitive to the overhead introduced in long lasting operations**. Users have been reported to be quite sensitive to Captive operations, even when they are not exposed to any overhead. This introduces a significant risk of annoying the user if the operation that is slowed down requires between 5 and 10 seconds to complete. As a consequence, monitoring and analysis solutions working in the field would be better if activated *selectively*, *avoiding to overlap with captive* operations as much as possible.

**Significantly reducing the reactiveness of the system may impact on the perceived SRT**. In case of 140% overhead, users perceived

for some of the Immediate operations a reduced SRT. This means that navigation and browsing operations, which can be normally completed quickly, can be exposed to relevant overhead levels (e.g., 80%) but might be problematic if exposed to higher levels (e.g., 140%). Again, this calls for the design of monitoring and analysis solutions working in the field that can be *selectively* activated based on the nature of the operation.

**The context of execution impacts on the perceived SRT**. Based on our study, the recent history of execution influences the perceived SRT. In particular, we study this phenomenon for the operations that correspond to domain functionalities and we discovered that Continuous operations are perceived as faster if executed after Captive operations, that is, if executed after a slower operation, while not exposed to excessive overhead levels (e.g., 140%). This introduces the challenge of having monitoring and analysis strategies that *keep track of the recent history* of the execution and exploit the context to *opportunistically increase or reduce their activity*.

**The specific nature of an operation matters**. There are of course exceptions to these general good practices inferred from our study. This means that any activity running in parallel with an application should be *carefully validated and refined* to consider the specific characteristics of the application.

## 2.7 Discussion

Development and operation are nowadays tightly integrated phases that are sometime hard to distinguish one from the other. The operational environment is not only designed to provide services to users but is also conceived as a monitoring, analysis and testing platform that can generate valuable data about the quality of the application [69, 40, 48, 5, 27, 60, 41, 15].

So far the research focused on how to enrich the operational environment paying limited attention on how the generated overhead may impact on the user experience. In this chapter we presented an empirical study about how users perceive delay introduced in the applications that they use. The main findings we distilled from this experience can be useful to design and tune solutions that operate in the field. For instance, we discovered that users did not perceive significant differences for an overhead of 80% and seldom perceived an overhead of 140%, that users are more sen-

sitive to delays experienced on operations of specific categories, and that the operations that have been recently executed may influence the user perception.

Even though our results were obtained for understanding the impact of monitoring overhead on the user experience, the results and findings exposed in this chapter could be also used for understanding the impact of any response delay in interactive applications, even if the delay is not related to the monitoring overhead.

Particularly, these findings will be useful for the design and implementation of the monitoring strategies presented in Chapter 3 and 4. Since monitors should not impact on the user experience while running in the field, it is thus very important to guarantee that the overhead introduced for each type of action respects the guidelines discovered in this human-subjects study.

# Chapter 3

# Controlled Burst Recording

This chapter presents *Controlled Burst Recording*, a sampling technique for monitoring applications and for inferring knowledge from field executions with little impact on the user experience. This solution has been designed based on the results obtained with the user study we conducted for understanding the impact of monitoring on the user experience. This chapter presents also an empirical evaluation carried out on a real world application for assessing the efficiency and the quality of the approach, compared to other sampling-based techniques.

The presentation of the chapter is organized as follows. Section 3.1 presents an overview of the technique. Section 3.2 presents the *Controlled Burst Recording* Framework and its different steps. Section 3.3 presents the empirical validation we conducted to assess our technique. Finally, Sections 3.4 and 3.5 discuss the obtained results and findings respectively.

## 3.1 Inferring Knowledge from Field Executions

*Controlled Burst Recording*, or *CBR* for short, is a technique to collect partial traces from the field. These traces could be used to produce a comprehensive knowledge about how an *object-oriented* application is used by different users in a wide range of possible scenarios and configurations. In general, our goal is to efficiently monitor interactive applications, that

is, applications that continuously interact with users. Since *CBR* is *user-oriented* we aim to monitor traces that start and finish with a user interaction.

Sampling-based techniques are useful to lower the monitoring overhead, since the amount of samples gathered from field are strongly reduced in a probabilistic way. In literature, sampling techniques have been widely used for isolating bugs [48, 40], runtime verification [7, 43, 4] and bursty monitors [35] (as described in Chapter 5).

*CBR* has been designed based on the results obtained in the user studies presented in Chapter 2. Interestingly, we discovered that users hardly perceive important overhead levels if introduced for a limited time (see Section 2.5.1), in particular, we observed that even for overhead levels of 140% users struggle to recognize variations in the system response time with the exception of some long lasting operations.

*CBR* leverages these results: since a significant overhead can often be tolerated by users, traces of non-trivial length might be feasibly collected from the field, but at the same time, since the impact of the monitoring activity changes with the operation that is executed, monitoring must be cleverly limited to prevent the introduction of an excessive overhead for specific operations.

Given *EV* as the set of every possible event that can be produced by a monitored application, a trace $T = \langle e_1, e_2, ..., e_n \rangle$, with $e_i \in$ ***EV***, $i = 1 \ldots n$ is an ordered sequence of events observed in the field.

In order to obtain a comprehensive information about the sequentiality of the traces, we might collect state information at the moment the execution traces are collected in the field.

All this knowledge can be represented as a Finite State Automaton, or *FSA* for short, where the states represent the state of the monitored application and the transitions represent how the state of the application changes as a consequence of the execution of a user operation.

Since several monitoring techniques in the literature collect method calls, eventually annotated with parameters or state information [61, 41, 42, 59, 17], we also focus on the collection of this data type.

In a nutshell, *Controlled Burst Recording* is an approach that combines chunks of execution recorded at different times to produce insights about the field behavior of the application. With this approach we might loss full ordering of the events in the *FSA* model, but since we are going to record

partial traces at different times of the execution we expect not to impact on the user experience.

In the following section, we present an overview of *Controlled Burst Recording* and the main steps of the approach.

### 3.1.1 Overview of the Technique

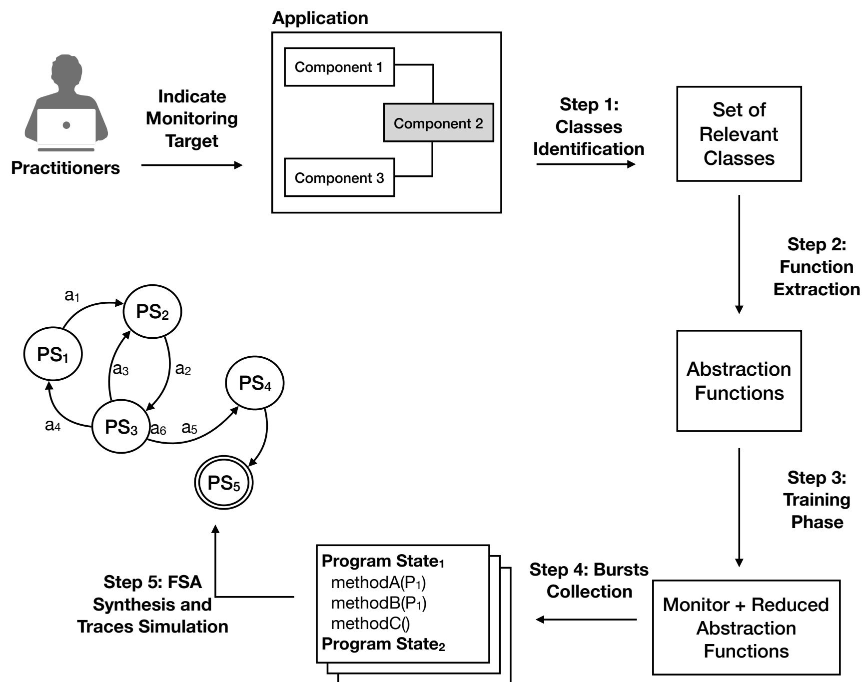



**Figure 3.1:** *Overview of Controlled Burst Recording approach.*

Figure 3.1 shows the steps that must be performed to apply *CBR*.

To start, practitioners indicate to *CBR* the components they would like to monitor in the field. To extract data from the chosen components, *CBR* identifies all the classes that may influence the usage of the components (step 1).

This analysis is done by first selecting a subset of the system (e.g., component) that has been defined as the monitoring goal and then by retrieving all the *relevant classes* and their respective methods that exist inside the selected component, which is the result of step 1.

Given that *CBR* has been designed to *sample* executions from the field, in order to decrease monitoring overhead, we can use program state information to give a temporal context to the different traces by calculating the state of the program before and after the set of events observed in the field.

This state information should be accurate enough to recognize different application states between user inputs, and inexpensive enough to be obtained at runtime so it does not add too much monitoring overhead.

One way to represent the state of an application is to consider the set of values of all the variables at a certain moment $t$ during the execution. Certainly, acquiring this information is too expensive to be done at runtime. An efficient way to perform this operation, is to subset the number of variables and the amount of information *CBR* should save for each variable while monitoring.

In the Function Extraction step (see Section 3.2.2), *CBR* analyzes the relevant classes to distill the knowledge about the functions that may influence their execution, and thus influence the interaction with the monitored components. In particular, *CBR* identifies a set of functions to be used to derive an abstract representation of the current state: we call the set of functions the *Abstraction Functions*, or *AF* for short.

The most intuitive way to derive the *AF*, is to extract from the application's code the functions that the application uses to define its execution path according to the runtime value of the variables. *CBR* will only consider those functions that helps to differentiate efficiently between different states at runtime.

For instance, let us assume that the monitored components control the interaction with the file system: if a method `write` may actually write to a variable `file` only if its boolean variable `open`, which represents if the file has been already open or not, is true, the Function Extraction step would output the function `file.open == true` to indicate that the evaluation of this function may influence the interaction with the monitored components. In general, the *AF* can capture important state information that could be useful to understand how the application under monitoring changes its state according to user inputs.

The *AF* are systematically applied to concrete states of the application to produce an abstract representation of the program state to which they were applied: these representations are called *Abstract Program States*, or *APS* for short.

The *AF* should have a proper balance in the abstraction level: too concrete functions can have bad implications, for instance checking all the field values of the program variables would be too expensive to be assessed at runtime (*performance*). On the other side, too abstract functions could generate poor program states that are not useful to distinguish different states of the application while running in the field (*precision*).

After having defined the *AF* through an automatic extraction process (see Section 3.2.2), the monitor is then assembled: the monitoring component consists of the State Monitor and the Event Monitor. The State Monitor produces concrete values of the *AF* at the beginning and at the end of each stream of events, and the Event Monitor collects the events associated with the component under monitoring.

To start using the State Monitor, we need to pass through a Training Phase (step 3) to produce the optimal set of functions for estimating a proper representation of the state of the program. These steps are described in details in Section 3.2.3.

Then, in the field, *CBR* uses the Abstraction Functions to produce a compact and relevant representation of the state of the program, immediately before and after a stream of events is recorded (step 4).

A *burst* is a stream of consecutive events collected in the field enriched with the state information at the beginning and at the end of each user action. In particular, a burst $B$ is given by a tuple $B = \langle label, S_1\, T\, S_2 \rangle$, where $label$ is the user action represented in $B$, $S_1$ and $S_2$ are two *APS* produced by the *AF* and the trace $T$ is the sequence of events collected in the field.

Since *CBR* is implemented as a sampling technique, the different bursts are collected probabilistically by turning on and off the monitor while running in the field.

The evaluation of the Abstraction Functions to a concrete state of the program during runtime (i.e., the *APS*) is used within bursts to represent the state of the application at a certain point of the execution.

The collected bursts are then analyzed offline. The *APS* at the beginning and at the end of each burst abstractly represent the state of the system at the time the trace was recorded. Since a program state directly depends on the conditions actually computed by the program while interacting with the monitored components, the *APS* is representative of how the execution may proceed, specifically in terms of the interaction with the monitored

components.

In principle, a set of bursts might be used to form a Finite State Automaton, to show real uses of the application (step 5), where the edges of the *FSA* represent the different user actions and the nodes represent the possible program states. Finite state automaton-based abstractions of software behavior are popular since they can be used as the basis for automated verification and validation techniques [75], such as detection of anomalous behaviors, protocol verification, testing, among others.

The literature on synthesizing *FSAs* that describe the behavior of a system is vast, some particularly relevant works include Wil van der Aalst's [74] book, entirely devoted to mining processes from observed data. Also, Lorenzoli et al. [50] worked on *GK-Tail*, a technique for automatically generating extended finite state machines from interaction traces. The technique focuses on the relations between constraints on data values on component interactions, retrieving more accurate models for analysis and testing techniques. In the same direction, Krka et al. [47] developed a *FSA* inference technique that uses not only execution traces but also program invariants in order to obtain relevant internal state information of the program variables and to produce more reliable models. Recently, Walkinshaw et al. [76] proposed an approach for inferring extended *FSAs* from software executions, this work focuses on addressing the non-determinism and the inflexibility problem of automatically generated *FSAs*.

Usually the functions that represent the values of the states of a *FSA* are obtained automatically (e.g., *Daikon* [28]) or manually (e.g., written by practitioners). *Controlled Burst Recording* introduces a new strategy for function extraction that is based on the actual software source code and should produce more precise and personalized *FSA* models.

To create the *FSA*, we merge the different bursts collected in the field that have a common state, intuitively a burst that ends with a certain state can be concatenated with a burst that starts with the same abstract program state, because it represents two consecutive actions performed by the user in the application.

The main reason for generating these *FSA* is representing the possible executions of an application, which may be useful to analyze the scenarios that may occur in the field.

## 3.2 *Controlled Burst Recording* Framework

The most important activity performed by *CBR* is the identification of the functions that are later used to represent the Abstract Program State, that is, a compact and abstract representation of the state at the time the collection of a burst is started or stopped.

In this section, we use a small example to present the different steps of *Controlled Burst Recording* Framework.

Consider a Java program that includes the `Cart` class (see Listing 3.1), which represents a shopping cart of an online store. The `Cart` class implements four different methods and the `Product` inner class: the `addItem` method adds a `Product` object to the shopping cart, which is represented by an array of `Product` type, the `emptyCart` method empties the shopping cart, the `calculateTotal` method estimates the total price of all the products in the cart and the `applyDiscount` method applies a discount to the total price.

Let us assume that we want to collect data about how a certain program uses the `Cart` class, focusing on the sequences of method calls produced by the program. Capturing every call to every method of the `Cart` class might be expensive for operations that extensively use the public methods implemented by the `Cart` class.

```
1  public class Cart {
2
3          static int PRICE = 1000;
4          static double DISCOUNT = 0.8;
5          static double TAX_PERCENTAGE = 0.22;
6          static int CART_SIZE = 30;
7
8          Product[] products;
9          int nProducts = 0;
10         double total = 0;
11
12         public void addItem(Product product) {
13                 if (nProducts == 0) {
14                         products = new Product[CART_SIZE];
15                 }
16                 products[nProducts] = product;
17                 nProducts++;
18         }
19
20         public void emptyCart() {
21                 if (nProducts > 0)
22                         products = new Product[CART_SIZE];
23         }
24
25         public void applyDiscount() {
26                 for (int i = 0; i < nProducts; i++)
27                         if (products[i].value < PRICE)
28                                 return;
29                 total = total * DISCOUNT;
30         }
31
32         public double calculateTotal() {
33                 for (Product p : products) {
34                         double pTaxes = 0;
```

```
35                      if (p.taxFree)
36                              pTaxes = 0;
37                      else
38                              pTaxes = p.value * TAX_PERCENTAGE;
39                      total = total + p.value + pTaxes;
40              }
41              return total;
42      }
43
44      public class Product {
45              int value;
46              boolean taxFree;
47      }
48
49  }
```

**Listing 3.1:** *Shopping Cart Class*

### 3.2.1 Step 1: Classes Identification

The *Classes Identification* step produces the set of classes that directly or indirectly determine the sequence of invocations to the methods in the monitored component. To derive the set of classes, practitioners need to indicate the component(s) of interest within the application under monitoring. Since one of the goals of *CBR* is to generate as little monitoring overhead as possible, it is important to include under the monitoring objective only those classes that are directly involved within the functionalities being monitored.

The set of classes is determined statically by extracting from the monitored component all the classes that exist inside the program. Then, with the help of WALA [22] static analysis tool, *CBR* constructs the Class Hierarchy of the program under monitoring and extracts all the classes that match the scope of the monitoring.

Later, we save the meta-data of all methods existing inside the classes we already gathered through the Class Hierarchy.

In our example, we are interested in understanding the behavior of the class `Cart`, so *CBR* considers the `Cart` and its `Product` inner class for the analysis.

### 3.2.2 Step 2: Function Extraction

*CBR* uses the classes produced during the step 1 to extract the **Abstraction Functions** that represent how the value of the state variables that may influence the execution of the monitored program. These functions are

exploited at runtime to record state information efficiently, that is, representing the state of the application with a proper abstraction level so we avoid performance and precision issues as stated in Section 3.1.1.

The Abstraction Functions computed by the monitored program, and in particular the ones computed in the relevant classes, are a set of functions that can be naturally exploited to define an efficient abstraction strategy that can be applied at runtime to generate the program state.

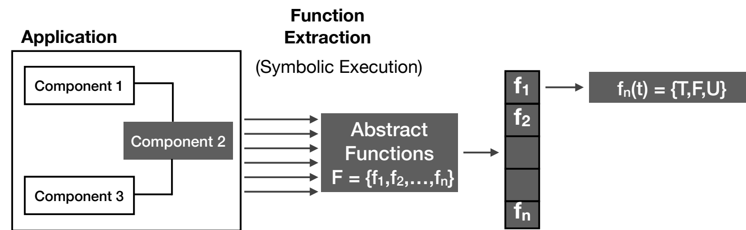



**Figure 3.2:** *Function Extraction process.*

To extract the Abstraction Functions from the code (see Figure 3.2), we symbolically execute each method in each selected class and use the *path conditions* resulting from the analysis of each method as functions for state abstraction. A *path condition* is a condition on the state variables of a program, and the Abstraction Functions are the application of these conditions to the program state.

We use the JBSE [11] symbolic executor to compute the Abstraction Functions from the relevant classes, bounding the exploration of loops.

Note that a *path condition* represents how the inputs to a method, that is, the values of the parameters and the values of state variables, relate to the execution of a specific path inside the method. *Path conditions* are a good abstraction because they show how applications use inputs and state variables.

For instance, the Abstraction Functions could be composed by a particular function that checks if the number of products in a shopping cart is greater than zero to decide how to use the shopping cart, in this case it would be enough to record whether the condition `nrProducts > 0` is true

or not when saving state information, with no need of recording the actual number of products, which would be irrelevant to determine how the execution may proceed.

The Abstraction Functions capture relevant state properties and represent the different execution paths, that in turn may produce different sequences of invocations to the monitored classes. Each evaluation of these functions at runtime produces the program state of the application.

From our `Cart` class example, the resulting Abstraction Functions include the following functions for each method:

```
1  Function 1
2  /* corresponds to the if condition in line 13, false case */
3
4  Cart.nProducts != 0 && Cart.products.length >= 0
5
6  Function 2
7  /* corresponds to line 14 */
8
9  Cart.nProducts == 0 && Cart.CART_SIZE >= 0 && Cart.nProducts < Cart.CART_SIZE
10
11 Function 3
12 /* corresponds to line 14 */
13
14 Cart.nProducts == 0 && Cart.CART_SIZE >= 0 && Cart.nProducts >= Cart.CART_SIZE
15
16 Function 4
17 /* corresponds to the if condition in line 13, true case */
18
19 Cart.nProducts == 0 && Cart.products.length == 0
```

**Listing 3.2:** *Functions for method addItem*

```
1  Function 1
2  /* corresponds to the condition for entering into the for loop in line 26,
       false case */
3
4  Cart.nProducts > 0 && Cart.products.length > 0
5
6  Function 2
7  /* corresponds to the condition for entering into the for loop in line 26,
       true case */
8
9  Cart.nProducts > 0 && Cart.products.length == 0
10
11 Function 3
12 /* corresponds to the if condition in line 27, false case */
13
14 Cart.nProducts > 0 && Cart.products.length > 0 && Cart.products.[0].value >=
       Cart.PRICE
15
16 Function 4
17 /* corresponds to the if condition in line 27, true case */
18
19 Cart.nProducts > 0 && Cart.products.length > 0 && Cart.products.[0].value <
       Cart.PRICE
```

**Listing 3.3:** *Functions for method applyDiscount*

```
1  Function 1
2  /* corresponds to line 35 when taxFree is true */
3
4  Cart.products.length > 0 && Cart.products.[0].taxFree == true
5
6  Function 2
```

```
7   /* corresponds to line 37 when taxFree is false */
8
9   Cart.products.length > 0 && Cart.products.[0].taxFree == false
10
11  Function 3
12  /* corresponds to the condition for entering into the for loop in line 33,
        true case */
13
14  Cart.products.length > 0
15
16  Function 4
17  /* corresponds to the condition for entering into the for loop in line 33,
        false case */
18
19  Cart.products.length == 0
```

**Listing 3.4:** *Functions for method calculateTotal*

```
1   Function 1
2   /* corresponds to the if condition in line 21, false case */
3
4   Cart.nProducts <= 0
5
6   Function 2
7   /* corresponds to if condition in line 21, true case */
8
9   Cart.nProducts > 0 && Cart.CART_SIZE >= 0
```

**Listing 3.5:** *Functions for method emptyCart*

Applying the Abstraction Functions to a certain program state produces a compact vector of ternary values: the size of the vector depends on the number of functions that are evaluated and the values are the result of the evaluation of a function to the current program state. The result of the evaluation of a function can be True (T), False (F), and Unknown (U). The vector of ternary values is called the Abstract Program State.

A single function evaluates to true if the values of the program variables at the time the function is evaluated satisfy the function. Similarly, a function evaluates to false if the values of the program variables at the time the function is evaluated do not satisfy the function. If some of the elements that appear in a function cannot be evaluated, for instance because an object with an attribute that should be considered in the evaluation of a function is null or even non existing, the function evaluates to unknown.

From our example, the Function 1 in Listing 3.2 for the method `addItem`:

```
Cart.nProducts != 0 && Cart.products.length >= 0
```

evaluates to true when there is more than one `Product` in the shopping cart, while it evaluates to false if the shopping cart is empty. Finally, it evaluates to unknown if the `Cart` object is not available.

In the other hand, for example the Function 4 in Listing 3.3 for the method `applyDiscount` :

```
Cart.nProducts > 0 && Cart.products.length > 0 &&
Cart.products.[0].value < Cart.PRICE
```

evaluates to true when the cart is not empty, and the price of the first `Product` in the cart is higher than the `PRICE` constant, evaluates to false whether the cart is empty or the price of the first `Product` is lower than the `PRICE` value, and evaluates to unknown when the object Cart does not exist in memory.

When the program under monitoring is executed, *CBR* produces a number of bursts consisting of an evaluation of the extracted functions on the state of the program, a sequence of events (each event is a method invocation), and again an evaluation of the functions on the state of the program.

The organization `state - burst - state` is useful, because allows *CBR* to understand the sequentiality of bursts. Since the states give information about the current program state, it is possible to correlate different bursts collected at different times and compare their states' values, to obtain a more comprehensive knowledge rather than considering only one burst.

Considering the previous example of the program exploiting the `Cart` class, a burst of its execution could take the form:

```
(U, U, F, T) addItem(product) calculateTotal() ...emptyCart() (T,
T, F, T)
```

where the part between parentheses is the Abstract Program State, which includes the evaluation of every function identified in the function extraction analysis (T, F, U stand for true, false, and unknown, respectively), and the sequence of labels between the two instances of the Abstract Program State is the sequence of events collected at runtime.

Considering the initial Abstract Program State (i.e., the first *APS* of the burst), let us assume that the first value refers to the mentioned Function 1 on the method `addItem`: the U in the program state means that, when the monitor started that trace recording, that function was evaluated as unknown, while at the end of the trace it turned to be true.

### 3.2.3 Step 3: Training Phase

The symbolic executor produces a large number of path conditions since it tries to assess every possible execution path for every method under analysis. Including all path conditions to represent the *AFs* is not practical,

since evaluating a high number of functions at every user interaction might introduce infeasible levels of overhead, and thus affecting the user experience. For this reason, we need to filter out the path conditions produced by the symbolic executor to reach a good compromise between the accuracy of the state information that is traced and the cost of producing such a state information.

From the original set of path conditions, we are interested in those giving unique information about the behavior of the application, that is, information that is not subsumed by the information checked by other functions. To find the optimal set of conditions, we need to go through a training phase that consists in empirically finding all the *AFs* that are useful for distinguishing the different logical states of the program as identified by the functions themselves. In other words, we run the application with a monitor that assesses the complete set of *AFs* returned by the symbolic executor to obtain the most representative result for the *AF*.

In this training phase, we evaluate the *AF* for every method call performed by the application, to clearly distinguish all different Abstract Program States at runtime.

For each user execution $l$ we generate a matrix $E^l_{iJ}$, with $i = 1 \dots k_l$ being $i$ a sample observed in the field with $k_l$ being the number of samples observed in the $l$-th execution, and $J = 1 \dots n_{af}$ being $J$ the abstraction functions under observation and $n_{af}$ the total number of *AFs*. For instance, in Figure 3.3 we show two examples of executions, with $n_{af} = 7$, collected in the field: Execution 1 ($E^1$) with $k_1 = 3$ and Execution 2 ($E^2$) with $k_2 = 2$.

Execution 1

| Sample | AF1 | AF2 | AF3 | AF4 | AF5 | AF6 | AF7 |
|---|---|---|---|---|---|---|---|
| 1 | U | U | T | U | U | U | F |
| 2 | U | F | T | T | F | F | F |
| 3 | U | F | T | T | F | F | F |

Execution 2

| Sample | AF1 | AF2 | AF3 | AF4 | AF5 | AF6 | AF7 |
|---|---|---|---|---|---|---|---|
| 1 | U | F | T | T | F | F | T |
| 2 | U | F | T | U | T | F | T |

**Figure 3.3:** *Example of two executions collected in the field.*

After collecting information for each execution, we organize the several $\{E_{ij}^1, E_{ij}^2, \ldots, E_{ij}^L\}$ with $L$ the total number of executions, into a matrix $M_{ij}$, where $i$ (rows) ranges from 1 to the sum of all samples $\sum_{l=1}^{L} k_l$ gathered at runtime, while $j$ (columns) goes from 1 to $n_{af}$. The matrix $M_{ij}$ is obtained as $M = \{E^1 \diamond E^2 \diamond \ldots \diamond E^L\}$, where the $\diamond$ operator represents the vertical concatenation of the $E_{iJ}^l$ matrices. For instance, considering $E^1$ and $E^2$ the executions of our example, the operation $E^1 \diamond E^2$ would generate the matrix presented in Figure 3.4.

| Sample | AF1 | AF2 | AF3 | AF4 | AF5 | AF6 | AF7 |
|---|---|---|---|---|---|---|---|
| 1 | U | U | T | U | U | U | F |
| 2 | U | F | T | T | F | F | F |
| 3 | U | F | T | T | F | F | F |
| 4 | U | F | T | T | F | F | T |
| 5 | U | F | T | U | T | F | T |

**Figure 3.4:** *Vertical concatenation of executions $E^1$ and $E^2$.*

During the filtering process, we can apply four type of operations to the matrix $M$ in order to remove samples or ***AFs***. Particularly, we identify four operations: *Duplicated Sample*, *Non-Discriminating Abstraction Function*, *Equivalent Abstraction Function* and the *Redundant Abstraction Function* operation. In Figure 3.5 we show an example of the filtering process and its several steps until we arrive to the reduced matrix, the example starts with the matrix generated in Figure 3.4.

*Duplicated Sample* is the case in which two samples obtained the same value for all the ***AFs***. Formally, two samples $M_{I_1 j}$ and $M_{I_2 j}$ are duplicated if $M_{I_1 j} = M_{I_2 j}$ for all $j = 1 \ldots N$.

In our example, in the first matrix we identified Duplicated Samples, in fact Sample 2 and Sample 3 are identical $S_2 = S_3 = \{U, F, T, T, F, F, F\}$, in general we remove the second occurrence of a repeated column/row, so in this case we remove Sample 3.

*Non-Discriminating Abstraction Function* are those abstraction functions that do not change their values throughout the different samples and thus are not useful to distinguish the program states at runtime. More for-

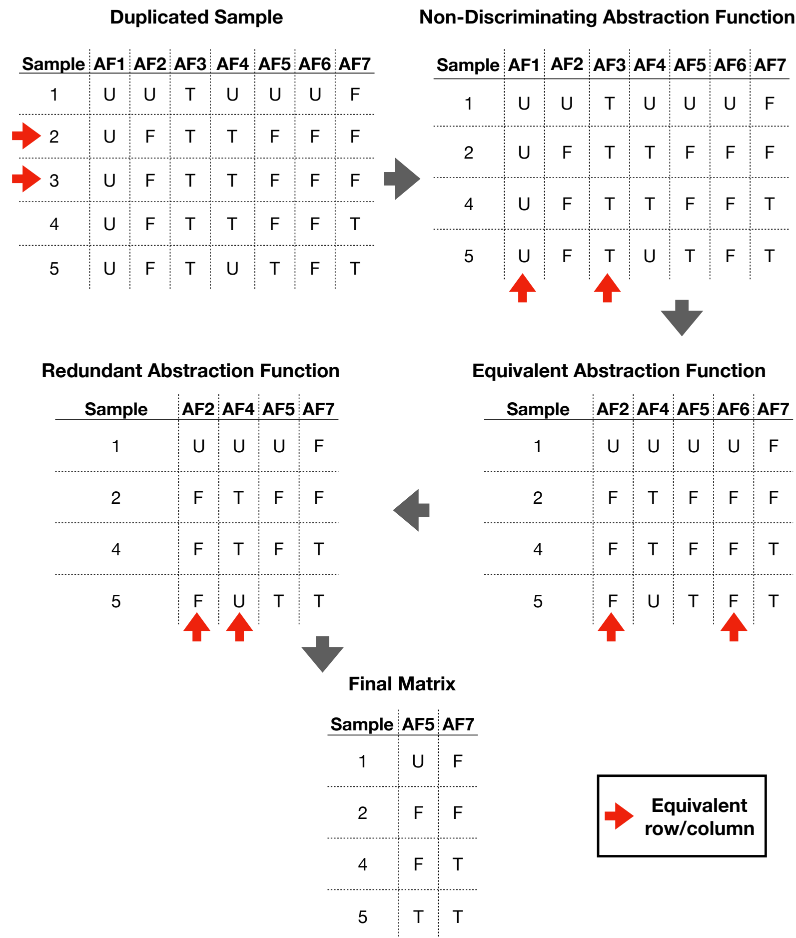



**Figure 3.5:** *Example of filtering process of AF.*

mally, an abstraction function $af_J$ is non-discriminating if $M_{1J} = M_{2J} = \ldots = M_{IJ}$

In the second matrix, it is possible to recognize two Non-Discriminating Abstraction Functions, in this case $AF_1$ and $AF_3$, because they always generated the values $U$ and $T$, respectively.

*Equivalent Abstraction Function* is the case in which there are two abstraction functions always produce the same set of abstract values. More formally, two functions $af_I$ and $af_J$ are equivalent if their columns $M_{iI}$ and $M_{iJ}$ are the same, that is, $M_{iI} = M_{iJ}$ for all $i = 1...M$.

In the example, it is possible to recognize two equivalent abstraction function, since the abstraction function values represented in column $AF_6$

are already present in column $AF_2$ ($AF_2 = AF_6 = \{U, F, F, F\}$), in this case we remove column $AF_6$.

Finally, the *Redundant Abstraction Function* reduction is a heuristic step that aims to remove those ***AFs*** that are not useful for identifying unique program states. The reduction procedure works by removing one abstraction function at a time and checking if the function was necessary to distinguish a unique program state. The procedure consists of removing one column $j$ of the matrix $M$ and applying the *Duplicated Sample* procedure: if the number of rows of the new matrix is equivalent to the original matrix $M$, it means that the *removed abstraction function j* is not useful for recognizing a unique program state, otherwise, if the number of rows of the new matrix is less than the matrix $M$ it means that the *removed abstraction function* is actually useful and we then reincorporate it to the matrix $M$.

In our example, ***CBR*** attempts first to remove the column $AF_2$. Since $AF_2$ is not useful, $AF_2$ is removed. Along with column $AF_2$, ***CBR*** also removes $AF_4$.

***CBR*** performs these operations iteratively until it arrives to a minimal representation of the executions with respect to the different ***AFs*** given by the symbolic executor: in this example the minimal set is composed by abstraction functions $AF_5, AF_7$ and executions $1, 2, 4$ and $5$. These ***AF*** are used to generate a monitor that is able to collect bursts during program executions.

### 3.2.4 Step 4: Bursts Collection

One possible way to decrease the overall monitoring overhead in the field is to collect data in a probabilistic way, that is, to record information about an event only under a certain probability.

From our human-subjects study presented in Chapter 2, we discovered that for a few actions it is actually possible to monitor applications in the field without impacting the user experience. So it makes sense to collect data occasionally when a user performs an action. Also, the monitoring of each burst starts by recording as soon as the user interacts with the application and it stops when the software finishes processing the user request.

The collection of bursts is done probabilistically: since the technique traces bursts of the execution signed with program state information, we

are not forced to record sequentially all bursts that occur in the field, that would correspond to continuously collect all the program execution interfering with the user activities. The program state information helps us to acquire a logical order of the different independent bursts collected at different times during different executions under monitoring.

The sampling probability of *CBR* will be then estimated experimentally, testing with different probabilities to search for the best cost-benefit compromise in terms of quality of the solution.

Let $Abstraction$ be the *AF* that applied to a certain concrete state $S$ of a program produces an Abstract Program State $S_a = Abstraction(S)$.

Following our example, some of the recorded bursts may for example look like:

```
(U, F) cart.applyDiscount() cart.calculateTotal() ...(F, F)
```

When the action *pay total* is performed, *CBR* saves the *APS* $(U, F)$ plus the target methods of the analysis, then *CBR* saves again a new representation of the *APS* equal to $(F, F)$.

### 3.2.5 Step 5: *FSA* Synthesis and Traces Simulation

At this point, we already gathered several independent bursts from the field, where each burst explains the effect of each user action on the program state. The issue is that each burst by itself provides partial information about a full execution. To overcome this issue, *CBR* puts together all the bursts in a common structure.

The right structure to assemble the different bursts is a *Finite State Automaton*: in fact, a *FSA* is a natural representation for traces with states and transitions. In our case, the *FSA* will show how the application's state changes according to different users' inputs.

To decide how to synthesize the *FSA*, *CBR* exploits state information contained in the different bursts. That is, if a burst $B_1$ ends with an abstract program state $S_2$ that matches the abstract program state $S_1$ reported at the beginning of another burst $B_2$, then the two bursts $B_1$ and $B_2$ can be concatenated by a node containing the state information in common between both bursts.

The Finite State Automaton is a tuple $G = (S, M, s_0, s_F)$, where $S$ is a finite non-empty set of nodes representing the different *APS* identified at the beginning and at the end of each burst, $M$ is a finite set of transitions

between states in $S$ representing the different user inputs monitored at runtime, $s_0 \in S$ is the initial state, and $s_F \subset S$ is the set of final states.

So, for each monitored burst $B = \langle label, S_1\,T\,S_2 \rangle$, ***CBR*** creates a transition $m \in M$ starting from state $S_1$ to $S_2$ with a specific label and a trace $T$. Since it is totally possible that different users perform the same action in different scenarios, ***CBR*** enriches the transition $m$ with more traces $T$.

The procedure of concatenating bursts is done iteratively for all bursts collected in the field. Certainly, as more bursts are used to create the graph, an increased number of program behaviors will be represented in it.

To simulate possible traces of the application we again exploit state information. Since we already built the ***FSA*** as the tuple $G = (S, M, s_0, s_F)$, the procedure is straightforward: consider two transitions $m_1, m_2 \in M$ connected by a common state $S_1 \in S$, ***CBR*** generates a new burst $B'$ with (1) the initial state $S_i$ of transition $m_1$, (2) a new trace $T'$ made by the concatenation of the trace $T_1$ from $m_1$ with the trace $T_2$ from $m_2$ and (3) the final state $S_f$ of transition $m_2$. The new burst $B' = \langle label, S_i\,T'\,S_f \rangle$ contains the simulated trace represented by $T'$.

The idea is to repeat this process until a larger trace is created.

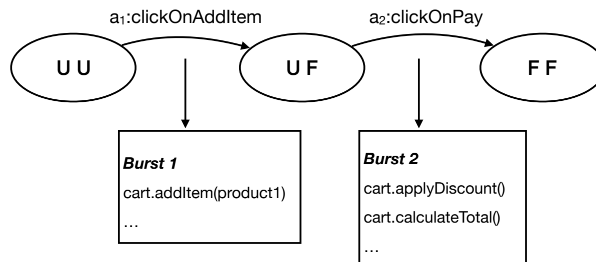



**Figure 3.6:** *Example of simulation of traces.*

In Figure 3.6, for our example we have two transitions: actions *clickOn-AddItem* and *clickOnPay*, that have in common the ***APS*** `(U, F)`. In this case, we might concatenate Bursts 1 and 2 and produce the following trace:

```
(U, U) cart.addItem(product1) ...cart.applyDiscount()
cart.calculateTotal() ...(F, F)
```

The generated traces could represent possible behaviors of the application, nevertheless if the quality of the model is not good, the procedure

could derived unfeasible traces. This aspect will be covered in the next section.

In a nutshell, *CBR* provides useful information at two different levels: (1) shows how the application is used in the field (*FSA*) and (2) through the traces simulated by the *FSA* shows different possible uses of the component under monitoring.

### *CBR* model validity

For evaluating the validity of the *CBR* model we consider two aspects: *precision* and *trace-level recall* of the traces generated by *CBR*.

The building process and the semantics of the *FSA* has a direct relationship with the precision of the technique. In some cases, it is possible that the generated traces are behaviors that do not occur in the field, which is the reason why we need to assess the precision of the generated *FSA*.

Particularly, to evaluate precision we check if the traces produced by the *FSA* model are possible outcomes of the application, in other words, we check if the derived traces from the *FSA* model exists in the original trace.

The precision is evaluated first locally (i.e., *node precision*) and then globally (i.e., *overall node precision*), in other words, we first assess if the local decisions taken in each node of the *FSA* are right. The assumption is that if on average the local decisions taken for each node are right, so the decisions taken globally by an execution derived from the model is also probably right.

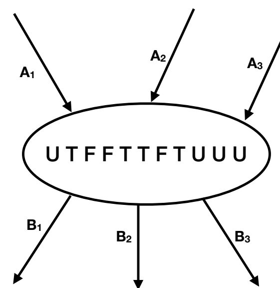



**Figure 3.7:** *FSA node with state information and incoming and outgoing transitions.*

To assess *node precision*, for each node present in the *FSA* model we check that every possible sequence is a possible program behavior or not.

For example, for the node in Figure 3.7 we first generate all the possible combinations of sequence of actions, that is, $A_1 \rightarrow B_1$, $A_1 \rightarrow B_2$, ..., $A_3 \rightarrow B_3$, then we verify if the traces produced by these sequences are present in the reworded traces, that is, traces recorded with a monitor that collects all the events sequentially.

The node precision for a particular node $i$ is estimated with the following formula:

$$Precision(node_i) = \frac{CS_i}{TS_i} \tag{3.1}$$

where $CS_i$ is the amount of correct sequences for the node $i$ and $TS_i$ is the total of possible combinations of sequences for the node $i$. The *overall node precision* is obtained by estimating the mean *node precision* of the *FSA* model, for simplicity in the rest of the document we refer to *overall node precision* simply as *precision*.

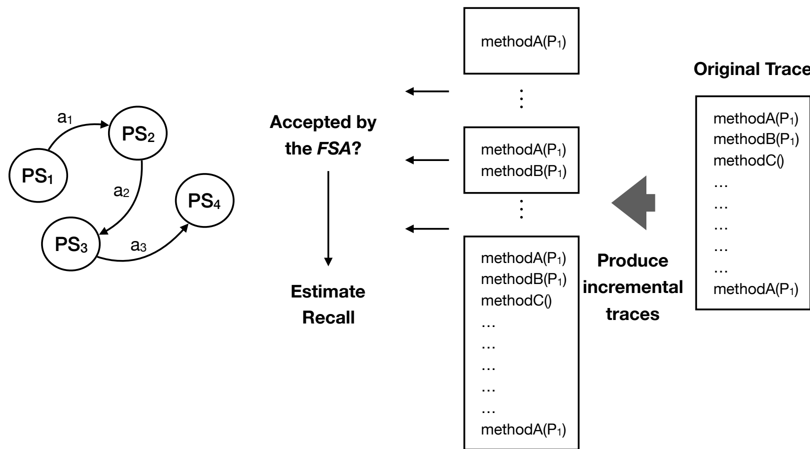



**Figure 3.8:** *Recall validation process.*

The *trace-level recall* is assessed by estimating how complete are the traces generated by the technique with respect to the original trace. In the rest of the document we refer to *trace-level recall* simply as *recall*.

In particular, we incrementally consider the single events contained in the original trace, checking if the incremental trace is accepted by the *FSA* (see Figure 3.8). We thus estimate the recall of the longest trace accepted by the *FSA* model.

Particularly, the recall for a trace $j$ is estimated with the following for-

mula:

$$Recall(trace_j) = \frac{E_j}{EO_j} \tag{3.2}$$

where $E_j$ is the number of events contained in the generated trace $j$, and $EO_j$ is the number of events in the original trace representing the execution of trace $j$.

### Finite State Automaton Example

Since the final step of the technique consists in generating a *FSA* and then producing simulations of possible field executions, we show an example of a possible *FSA* and how we assess the validity of the model.

Consider the GUI application for the shopping cart example of the previous sections that enables users to perform operations such as: adding items to the shopping cart, paying for all the items in the cart and starting a new session (emptying the shopping cart). If we monitor this application using *CBR* with the *Abstraction Functions* derived in the previous steps, we can for example collect the following bursts during an user execution (we show only methods related to the `Cart` class):

```
1  PS_0: (U, U)
2  ...
3  M: addItem(Product)
4  P: Product = [id:1, type:'chair']
5  ...
6  PS_1: (U, F)
```

**Listing 3.6:** *Burst 1: clickOnAddItem*

```
1  PS_1: (U, F)
2  ...
3  M: addItem(Product)
4  P: Product = [id:5, type:'desk']
5  ...
6  PS_1: (U, F)
```

**Listing 3.7:** *Burst 2: clickOnAddItem*

```
1  PS_1: (U, F)
2  ...
3  M: applyDiscount()
4  ...
5  M: calculateTotal()
6  ...
7  PS_2: (T, T)
```

**Listing 3.8:** *Burst 3: clickOnPay*

```
PS_2: (T, T)
...
M: emptyCart()
...
PS_0: (U, U)
```

**Listing 3.9:** *Burst 4: clickOnStartNewSession*

```
PS_1: (U, F)
...
M: emptyCart()
...
PS_0: (U, U)
```

**Listing 3.10:** *Burst 5: clickOnStartNewSession*

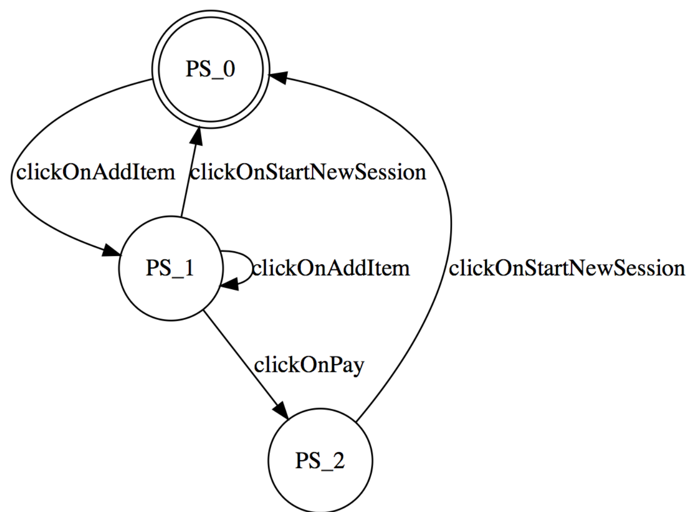



**Figure 3.9:** *Example of resulting FSA*

By following the procedure for joining the collected bursts in a *FSA*, we could obtain the model shown in Figure 3.9.

```
...
M: addItem(Product)
P: Product = [id:1, type:'chair']
...
M: addItem(Product)
P: Product = [id:5, type:'pencil']
...
M: addItem(Product)
P: Product = [id:10, type:'glass']
...
M: applyDiscount()
...
M: calculateTotal()
...
M: emptyCart()
...
```

**Listing 3.11:** *Trace 1*

```
1  ...
2  M: addItem(Product)
3  P: Product = [id:1, type:'chair']
4  ...
5  M: emptyCart()
6  ...
7  M: addItem(Product)
8  P: Product = [id:15, type:'desk']
9  ...
10 M: applyDiscount()
11 ...
12 M: calculateTotal()
13 ...
14 M: emptyCart()
15 ...
```

**Listing 3.12:** *Trace 2*

```
1  ...
2  M: addItem(Product)
3  P: Product = [id:1, type:'chair']
4  ...
5  M: addItem(Product)
6  P: Product = [id:5, type:'pencil']
7  ...
8  M: addItem(Product)
9  P: Product = [id:10, type:'glass']
10 ...
11 M: applyDiscount()
12 ...
13 M: calculateTotal()
14 ...
```

**Listing 3.13:** *Trace 3*

We assess the validity of the model against traces 1, 2 and 3, represented respectively in Listings 3.11, 3.12 and 3.13. These traces has been observed by a monitor that collects all the events in the field, so we highlight only the methods related to the `Cart` class.

As described previously, to assess the validity of the model we first need to evaluate the precision of the *FSA*. We use Equation 3.1 to estimate the precision for nodes $PS_0$, $PS_1$ and $PS_2$ in the *FSA*. In particular, we check that all possible traces combination from nodes actually exist in traces 1, 2 or 3.

**Table 3.1:** *Precision for each node in the FSA*

| Node | Node precision |
|---|---|
| $PS_0$ | 100% |
| $PS_1$ | 100% |
| $PS_2$ | 100% |

The reported precision for each node is shown in Table 3.1. The average precision of the *FSA* is $100\%$.

We then use Equation 3.2 to evaluate the recall of the traces produced through the *FSA*. In other words, make sure that traces 1, 2 and 3 are a possible outcome of the model.

**Table 3.2:** *Recall of each trace generated by the FSA*

| **Trace** | **Recall** |
|---|---|
| $Trace_1$ | 100% |
| $Trace_2$ | 100% |
| $Trace_3$ | 100% |

The recall of each trace is reported in Table 3.2. The average recall of the traces produced from the *FSA* is $100\%$.

Considering the values we achieved for *precision* and *recall*, we conclude that the *FSA* presented in Figure 3.9 will produce trustful traces of high quality for the example.

However, if *CBR* would identified a *clickOnAddItem* action with a transition going from node $PS_2$ to $PS_0$, the node precision of node $PS_2$ would be dropped to $50\%$, since the sequence of actions *clickOnPay* followed by *clickOnAddItem* was never observed in the field.

## 3.3 Empirical Validation

This Section presents the empirical validation we conducted for *Controlled Burst Recording*, in order to assess the quality of the solution with respect to the performance, recall and precision in comparison to sampling techniques. We provide information about the details of the experiments, the investigated aspects and the results we obtained for *CBR*.

### 3.3.1 Goal and Research Questions

The goal of the experiment is to evaluate the performance, the precision and the recall of the traces produced by *Controlled Burst Recording*, by comparing results with different approaches present in the literature.

With this experiment we aim to answer the following research questions:

- **RQ1: What is the overhead introduced by *Controlled Burst Recording*?** In this research question, we measure the performance of our technique with respect to other sampling techniques.

- **RQ2: What is the precision of the model generated by *Controlled Burst Recording*?** In this research question, we measure the precision of the *FSA* model with respect to the traces collected with monitoring.
- **RQ3: What is the recall of the traces produced by *Controlled Burst Recording*?** In this research question, we measure the recall of the technique with respect to the monitored traces.

### 3.3.2 Experiment Design

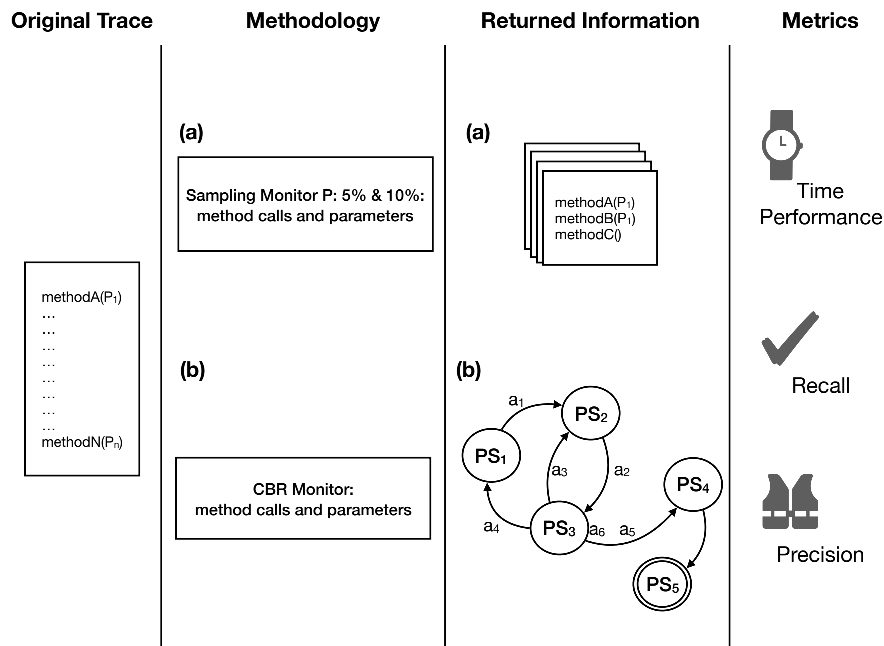



**Figure 3.10:** *Controlled Burst Recording Validation.*

Figure 3.10 shows the high-level work-flow of the experiment. The main idea of the experiment is to compare *CBR* to other sampling techniques, with respect to the performance, precision and recall of the technique itself. The monitoring objective is to collect the method calls related to a certain execution, including the runtime value of the parameters associated to each method call.

Sampling monitors have been used widely for monitoring applications in the field [40, 49, 43, 4]: this type of monitors decrease the overall monitor-

ing overhead by observing events only under a certain probability. When analyzing interactive applications, each time the user interacts with the application the sampling monitor records with a certain probability. When the monitor is activated, it registers a defined amount of events.

For the experimentation, we implemented two sampling monitors, one with a sampling probability of $5\%$ and a second one with a $10\%$ of probability. Both sampling monitors trace a fixed number of events, in this case we fixed the amount to $30$ events per trace: a similar configuration setup has been used in [35] for a sampling monitor.

For assessing the *performance* we measure execution times of a program being monitored with the three approaches compared to a program execution without any monitoring. The granularity of the monitoring is the single user action, that is to say measuring the execution time from the moment the user interacts with the application to the moment the program has finished processing the user request and gives a feedback to the user. Since we care about how monitoring impacts on different functionalities, we analyze overhead with respect to the different categories of operations of the system response time categorization (see Chapter 2).

To address the validity of the *CBR* model we consider the procedure specified in Section 3.2.5: we assess *precision* by applying Equation 3.1 to every single node of the *FSA* produced from the traces generated during the experimentation. Instead, we assess *recall* by estimating how complete are the traces generated by each approach with respect to the original traces, in particular, we use Equation 3.2 to calculate the Recall indicator of a certain trace.

### 3.3.3 Experimental Subject

To empirically answer the research questions, we use *Controlled Burst Recording* to monitor ArgoUML [73]. ArgoUML is a UML diagramming application written in Java (389,952 lines of code): within the features of ArgoUML is possible to design Use Case, Class, Sequence, Collaboration, State-chart, Activity and Deployment diagrams. We selected ArgoUML because is a representative program of the target applications we are studying here: an user-interactive software application of medium-big size.

We considered the package `org.argouml.uml.diagram.activity` as the target of monitoring, in particular, this package manages all the function-

alities related to the design and management of activity diagrams in ArgoUML. We chose this package since activity diagrams are a very common way to represent behavioral information about software systems, so it is easy to make examples of this type of diagrams.

### Definition of the Abstract Program State

To define the Abstract Program State we start by doing static analysis of the component to be analyzed: Table 3.3 offers some information about this process. To start, static analysis process discovered 16 classes and 199 methods inside component `org.argouml.uml.diagram.activity`, then the discovered methods are used as inputs to the symbolic executor, which processes each method and produce the corresponding abstraction functions. The symbolic executor produced 5,732 abstraction functions.

**Table 3.3:** *Monitoring Target and Symbolic Execution Outcome*

| **Monitoring Target** | `org.argouml.uml.diagram.activity` |
|---|---|
| **Packages Analyzed** | `org.argouml.uml.diagram.activity` |
| | `org.argouml.uml.diagram.activity.layout` |
| | `org.argouml.uml.diagram.activity.ui` |
| **Classes Analyzed** | 16 |
| **Methods Analyzed** | 199 |
| **Abstraction Functions** | 5,732 |

In order to recreate proper executions of ArgoUML we have written 30 automatically executable test cases that encode typical usage scenarios for `org.argouml.uml.diagram.activity` component: each test case consists in simulating the drawing of one activity diagram on the application (e.g. diagrams such as Figure 3.11). These 30 test cases are used along the calibration and experimentation process. To avoid non-determinism and for reproducibility of the test cases we implemented them with the Sikulix [36] capture and replay tool.

To train and produce the optimal representation of the program state, we run the application with the monitor using the original 5,732 abstraction functions. Once we execute the 30 test cases, we produce a single log file with all the possible program states detected across the simulations. As specified in Section 3.2.3, our technique filters out all the *Non-Discriminating*, *Equivalent* and *Redundant* Abstraction Functions.

The result of the filtering process is shown in Table 3.4: the procedure

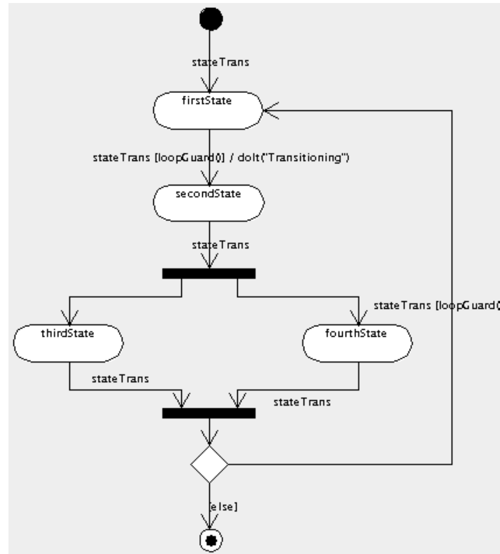



**Figure 3.11:** *Activity Diagram example in ArgoUML*

**Table 3.4:** *Details from the abstraction function filtering process.*

| | **Number of Abstraction Functions** |
|---|---|
| Original Set from JBSE | 5,732 |
| *Non-Discriminating Abstraction Function* | 1,631 |
| *Equivalent Abstraction Function* | 2,646 |
| *Redundant Abstraction Function* | 1,298 |
| **Filtered Set** | **157** |

filtered out 5,575 functions, leaving 157 *AF* as the optimal representation for *APS*. In particular, the *Non-Discriminating Abstraction Functions* were 1,631, the *Equivalent Abstraction Functions* were 2,646 and the *Redundant Abstraction Functions* were 1,298.

## 3.4 Results

In this section we present the results of the experiment we conducted to validate *Controlled Burst Recording*. All the experiments were executed on a computer running macOS version 10.13.6 with a 3.1 GHz Intel Core i7 processor and 16 GB of RAM.

The following sections describe the results obtained to answer to the different research questions.

### 3.4.1 Performance Results

To answer the research question **RQ1: What is the overhead introduced by *Controlled Burst Recording*?** we assessed the impact of the monitors, specifically we measured the overhead by comparing the execution times of the monitored version with those without monitoring. Then, we ran our set of 30 test cases, each test has been repeated 3 times and mean values have been used to mitigate any effect due to the non-determinism of the execution environment. Overall, we collected near 3,459 samples about the execution time of actions.

In addition to measuring execution times, we collect the execution traces that will be useful for answering the research questions **RQ2** and **RQ3**.

Since this work is focused on understanding the impact of the monitoring on the user experience, we classify user actions according to its system response time category, in particular we used the categorization presented in Chapter 2 (i.e., Instantaneous, Immediate, Continuous and Captive actions). Along the 30 test cases, we identified 915 actions of the Instantaneous category, 25 actions from the Immediate category and 213 actions from the Continuous category. Given the nature of the functionalities we tested, we did not identify Captive actions.

**Table 3.5:** *Performance results with respect to system response time categories.*

| | **Monitoring Overhead [%]** | | |
|---|---|---|---|
| **Monitoring Technique** | **Instantaneous** | **Immediate** | **Continuous** |
| *CBR* | 123.78 | 40.70 | 0.71 |
| Sampling 1 (P: 10%) | 17.03 | 4.65 | 1.37 |
| Sampling 2 (P: 5%) | 7.12 | 2.88 | 0.28 |

The results for performance can be found in Table 3.5: specifically, we present overhead data for the different monitors used in the experimentation with respect to the different system response time categories.

In the table, *CBR* represents *Controlled Burst Recording*, *Sampling 1* the monitor that traces in a sampling way with 10% probability and *Sampling 2* the monitor that samples with 5% probability.

The first thing we can notice is that for the actions categories that employ less than one second, i.e., Instantaneous and Immediate actions, *CBR* introduces considerable more overhead than the other techniques, 123.78% of *CBR* compared to 17.03% of Sampling 1, 7.12% of Sampling 2 for Instantaneous actions, or the 40.70% of *CBR* compared to the 4.65% and 2.88% of Sampling 1 and 2 respectively for Immediate actions.

The increased overhead for actions $< 1$ second is explained by the fact that *CBR*, besides tracing events, needs to evaluate 157 conditions at the beginning and at the end of each action, plus the fact that Instantaneous and Immediate actions are volatile and very sensitive to any additional time.

The second thing we notice is that contrary to Instantaneous and Immediate actions, *CBR* performs equally for actions $> 1$ second (i.e., Continuous actions), in fact *CBR* introduced 0.71% of overhead, which is similar to the performance of Sampling 1 (1.37%) and Sampling 2 (0.28%). Since these actions are less volatile, the task of evaluating program state does not impact on the final overhead.

However, from our human-subject studies results (see Section 2.5) we discovered that users do not perceive high monitoring overhead levels for Instantaneous actions, and that users perceive a degradation of the system response time only at 140% for Immediate actions, meaning that in practice, users would not perceive monitoring overhead when *Controlled Burst Recording* is activated.

Moreover, Sampling 1 and 2 techniques are not useful for simulating possible executions from the field, since they are observing only small percentages from the execution and do not provide contextual information, while *CBR* is capable of representing comprehensive behaviors from the traces signed with program state information.

### 3.4.2 Precision Results

For answering the research question **RQ2: What is the precision of the model generated by *Controlled Burst Recording*?** we measured the precision of each node when simulating possible traces from the *FSA* generated by *Controlled Burst Recording*.

We show results only for *CBR*, since it is the only technique in this work that simulates possible execution traces, meaning that the precision of the traces recorded by sampling monitors is 100%.

Table 3.6 and Figure 3.12 show information about precision of *CBR* given the sampling probability and the amount of runs of the application.

As explained in Section 3.2.4, *CBR* works in a probabilistic way, meaning that the monitoring of each interaction between the user and the application occurs only under a certain probability. If *CBR* is deployed to

**Table 3.6:** *Precision results for Controlled Burst Recording.*

| | 1 run | 5 runs | 10 runs | 15 runs | 20 runs |
|---|---|---|---|---|---|
| P [%] | Precision [%] | Precision [%] | Precision [%] | Precision [%] | Precision [%] |
| 5 | 0 | 66.67 | 68.42 | 70.21 | 68.11 |
| 10 | 50.00 | 68.42 | 63.39 | 72.51 | 74.22 |
| 15 | 100 | 70.73 | 72.36 | 76.62 | 78.83 |
| 20 | 66.00 | 65.24 | 79.98 | 77.80 | 79.70 |
| 25 | 55.35 | 66.67 | 80.27 | 79.97 | 80.43 |
| 30 | 47.83 | 72.10 | 81.15 | 80.34 | 80.55 |
| 35 | 38.89 | 79.61 | 80.83 | 81.22 | 80.25 |
| 40 | 72.73 | 75.89 | 79.84 | 80.52 | 81.02 |
| 45 | 64.62 | 79.95 | 80.55 | 80.62 | 79.88 |
| 50 | 72.92 | 80.89 | 80.75 | 80.57 | 81.12 |
| 55 | 65.87 | 80.16 | 80.66 | 80.62 | 80.39 |
| 60 | 65.88 | 80.00 | 80.29 | 81.17 | 80.20 |
| 65 | 66.82 | 80.98 | 80.52 | 80.43 | 79.92 |
| 70 | 71.55 | 80.38 | 80.57 | 80.66 | 80.34 |
| 75 | 78.14 | 80.41 | 80.89 | 80.75 | 80.48 |
| 80 | 74.44 | 80.43 | 80.48 | 81.23 | 80.56 |
| 85 | 75.23 | 81.23 | 81.07 | 80.48 | 80.41 |
| 90 | 78.69 | 80.98 | 80.66 | 79.98 | 80.70 |
| 95 | 79.52 | 80.70 | 80.75 | 81.02 | 80.49 |
| 100 | 80.71 | 80.70 | 80.70 | 80.66 | 81.21 |

monitor data with a probability lower than 100%, it is necessary to run several times the same functionality in order to observe the whole behavior in the field. Since in practice a program is executed several times in different instances, we would observe several repeated observations of the same functionality throughout the monitoring process, and by increasing the number of runs of the monitor fewer samples will be needed in order to derive accurate results.

In Table 3.6 we present how the overall precision changes according to the number of observed runs of the application, in this case, we experimented with 1, 5, 10, 15 and 20 runs.

First, we observe that the maximum precision of the technique is around 81%, this means that for every node in the *FSA*, 81% of the possible combinations of bursts are a feasible behavior of the application, while the remaining 19% are traces that we never observed in the traces recorded when activating *CBR* all the times.

However, not all the 19% are unfeasible traces: a subset of the traces represented by the remaining 19% could still be possible behaviors of the software (e.g., we do not developed all the necessary test cases to cover the remaining 19%).

Second, we note that with 5 runs of the application we achieve the maximum of the precision for our technique with a sampling probability of 50%. Instead, when considering 10, 15 or even 20 runs, the maximum precision

Table 3.7: *Recall results for Controlled Burst Recording.*

| | 1 run | | 5 runs | | 10 runs | | 15 runs | | 20 runs | |
|---|---|---|---|---|---|---|---|---|---|---|
| **P [%]** | **RB [%]** | **RE [%]** | **RB [%]** | **RE [%]** | **RB [%]** | **RE [%]** | **RB [%]** | **RE [%]** | **RB [%]** | **RE [%]** |
| 5 | 4.68 | 3.95 | 9.70 | 8.99 | 13.53 | 12.18 | 16.66 | 16.58 | 22.62 | 20.78 |
| 10 | 5.41 | 4.78 | 13.06 | 11.50 | 21.82 | 21.25 | 28.48 | 28.30 | 38.47 | 41.32 |
| 15 | 7.24 | 6.13 | 18.34 | 16.61 | 31.57 | 32.34 | 47.33 | 47.91 | 61.56 | 63.85 |
| 20 | 8.53 | 5.83 | 22.86 | 21.93 | 42.17 | 43.43 | 63.36 | 64.35 | 84.64 | 86.36 |
| 25 | 9.37 | 7.36 | 29.16 | 27.73 | 49.41 | 51.47 | 75.33 | 78.44 | 95.93 | 96.97 |
| 30 | 11.74 | 10.20 | 32.25 | 33.05 | 67.65 | 70.82 | 92.02 | 92.14 | 96.89 | 96.89 |
| 35 | 11.46 | 9.71 | 39.47 | 42.39 | 86.66 | 88.36 | 100 | 100 | 100 | 100 |
| 40 | 14.32 | 12.21 | 42.65 | 44.97 | 84.80 | 85.45 | 100 | 100 | 100 | 100 |
| 45 | 15.12 | 11.34 | 67.06 | 68.73 | 98.70 | 98.75 | 100 | 100 | 100 | 100 |
| 50 | 15.59 | 14.31 | 61.44 | 62.82 | 98.70 | 99.19 | 100 | 100 | 100 | 100 |
| 55 | 17.56 | 17.44 | 87.95 | 88.89 | 98.59 | 99.14 | 100 | 100 | 100 | 100 |
| 60 | 19.29 | 19.55 | 77.51 | 78.87 | 100 | 100 | 100 | 100 | 100 | 100 |
| 65 | 23.89 | 23.60 | 90.84 | 91.05 | 100 | 100 | 100 | 100 | 100 | 100 |
| 70 | 26.21 | 25.84 | 98.78 | 99.01 | 100 | 100 | 100 | 100 | 100 | 100 |
| 75 | 28.31 | 27.08 | 98.99 | 99.23 | 100 | 100 | 100 | 100 | 100 | 100 |
| 80 | 31.80 | 33.93 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 85 | 31.80 | 33.93 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 90 | 36.60 | 39.44 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 95 | 57.47 | 59.96 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |

can be achieved with a 25% sampling probability, which seems to be the more balanced result for all measurements.

Nevertheless, considering 5, 10, 15 or 20 runs are a very small number of executions in comparison to running the application thousands of times across the several instances in the field. Hence, taking into account thousands of samples for a same user action would allow us to considerably reduce the *CBR* sampling probability.

### 3.4.3 Recall Results

In this section, we present results for the research question **RQ3: What is the recall of the traces produced by *Controlled Burst Recording*?**. As stated in Section 3.3.2, to estimate the recall we measure the percentage covered by traces obtained with each monitor in comparison to those obtained with *CBR* monitor activated all the time.

For *Controlled Burst Recording* we present both in Table 3.7 and Figure 3.13 the recall of the technique according to the sampling probability and how it changes according to the number of runs of the application. The recall is presented in two measures, the **RB** and **RE** indicators: the **RB** indicator shows the recall percentage with respect to the number of bursts that can be identified within the original traces in comparison to the simulated trace. Instead, the **RE** indicator shows the recall percentage with respect to the number of events that can be identified in the original traces

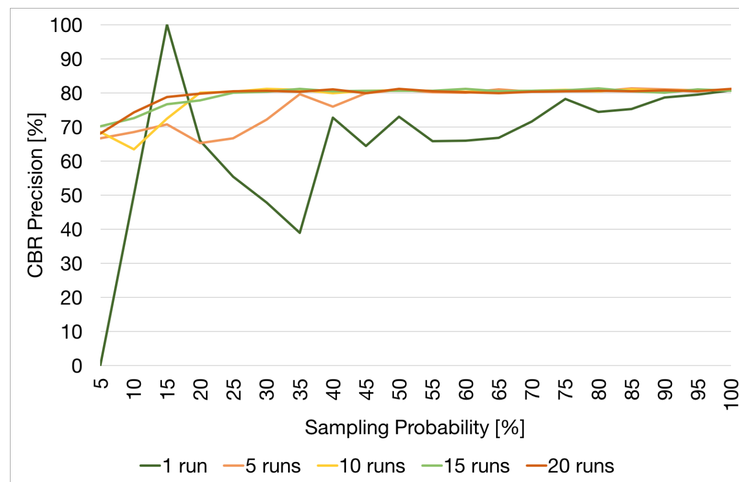



**Figure 3.12:** *Precision results with respect to different runs of the application.*

in comparison to the trace simulated by the *FSA*.

We notice that considering only one run of the application we only achieve 100% recall with sampling strategy at 100%, which means having the monitor always on. When we decrease the sampling probability to 90%, the recall of the technique decreases rapidly to 36%.

Given that we are considering only one run, it is very likely that our *FSA* is missing several transitions and thus producing very short and incomplete traces. In fact, when considering 5 runs of the application it is possible to achieve a good recall percentage with 65% of sampling probability.

Now, if we consider 10 runs of the application we obtain approximately $90\%$ of recall at a 45% of sampling probability. Going further, for 20 runs of the applications we already obtain a good recall level for 25%. Which is also a similar result obtained for precision in Section 3.4.2.

On the other side, for sampling monitors we obtained a recall value of 4.15%, with a maximum recall of 7.77%, meaning that the traces collected by this approach contains on average 4.15% events from the total events contained in the traces obtained when activating *CBR* all the times.

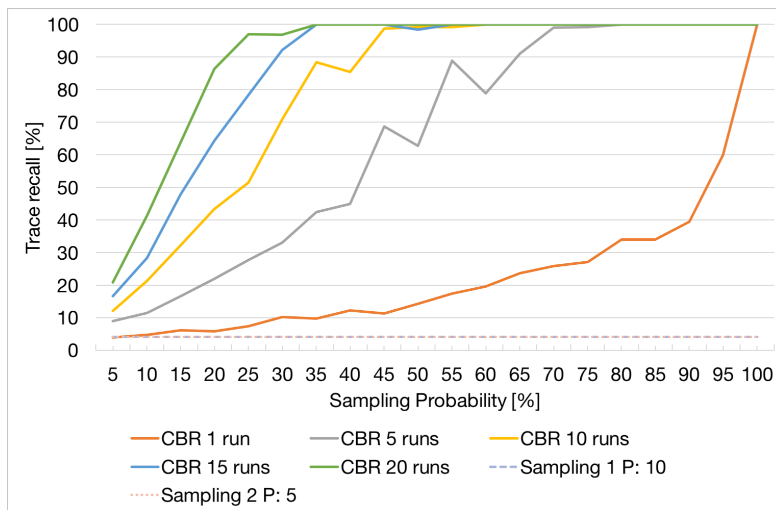



**Figure 3.13:** *Recall results with respect to different runs of the application.*

## 3.5 Discussion

We have carried out an experimental evaluation for assessing the quality of *CBR* in terms of monitoring *performance* with respect to other two sampling approaches.

Since *CBR* works in a probabilistic way and does not trace full executions, we also assessed the *precision* and the *recall* of the traces simulated by *CBR*, to understand the usefulness of the sampling approach in an attempt to decrease monitoring overhead and to reduce the impact on the user environment.

When assessing performance, we discovered that the monitoring overhead introduced by *CBR* with respect to other sampling approaches was higher for user actions employing less than one second, and similar for those user actions employing more than one second.

However, by considering our earlier results obtained in Chapter 2 about the impact of monitoring overhead on the user experience, we conclude that *CBR* would not be perceived by users, since the overhead introduced by the technique is always lower than 140%, which is the limit for tolerating monitoring overhead in the users' environment, while allowing the technique to obtain traces with enriched information for generating more comprehensive knowledge of the application usage.

About the quality of the traces produced by *CBR*, we discovered that the

precision of the simulations we produced were 81% with respect to the original traces. Unfortunately, our experimentation was not enough to prove if the remaining 19% were actually infeasible behaviors or they were simply unobserved behaviors of the application.

On the other side, *CBR* was able to simulate traces that were 100% complete with respect to the original traces. A very different result compared to the 4% recall obtained by the other sampling approaches.

Nevertheless, we noticed that by using a sampling rate fixed to 25% was possible for *CBR* to reproduce safe traces that represent reliable behaviors of field executions. We are also aware that by increasing the amount of users, that is, increasing the monitored executions, we might considerably decrease the sampling rate to make *CBR* a less expensive solution.

We also discovered the importance of adding state information to traces: as a matter of fact by using the *CBR* monitoring approach we were able to observe complete user executions, while other sampling approaches not employing state information only observed small percentages of the user execution.

In the overall, *CBR* is a good monitoring approach that can be used for constructing and simulating knowledge from field executions without impacting on the user experience.

# Chapter 4

# Delayed Saving

This chapter presents Delayed Saving, a technique for delaying the saving of events to file during field monitoring. Even though Delayed Saving should not be mandatorily used, it is very useful when used to avoid monitoring to interfere with the user experience. This chapter presents also an empirical evaluation carried out on a real world application for assessing the efficiency and the quality of the approach, compared to immediate recording.

The presentation of the chapter is organized as follows. Section 4.1 overview the Delayed Saving technique. Section 4.2 discusses the parameters that may influence the behavior of the technique. Section 4.3 presents the Delayed Saving process and its steps. Section 4.4 presents the empirical validation we conducted to assess our technique. Finally, Sections 4.5 and 4.6 discuss the obtained results and findings, respectively.

## 4.1 Prioritizing User Interactions

We already discussed how monitors slow down the performance of applications. We also acknowledge that the interaction of users with software applications is the most critical aspect of monitoring, since, in some cases, users are very sensitive to delays that may occur in the system response time, according to the results reported in Chapter 2. Consequently, a monitoring strategy should limit the activity performed in parallel with users operations, in order to reduce its impact on the user experience.

In interactive applications it is possible to distinguish two different

phases of the interaction between the user and the application: *working* time and *idle* time. *Working* time is the interval of time from the time the user produced an input for the application and to the time the application responds to that input. On the contrary, *idle* time is the time from a response to the next user input.

Saving data during the monitoring phase could be extremely expensive. To initially quantify the impact of the process that saves data to a file compared to saving data in memory, we implemented a small program in Java that creates objects of the `java.awt.Point` class and saves them following two different procedures: (a) the first procedure saves the objects in a collection of type `java.util.ArrayList`, while (b) the second procedure saves the created objects directly to a file (in this case we use the `toString` method to generate the information to be saved for each object). We made six different executions, making some variations in the amount of objects created (see Table 4.1), and the amount of information we save through the `toString` method. In particular, we measured the time consumption for both procedures to assess the impact of saving data in a file in comparison to saving data to the memory. The outcome of this small experimentation is shown in Table 4.1 and in Figure 4.1.

**Table 4.1:** *Time performance comparison when saving to memory/file.*

| Example | `toString` modified? | Number of created objects | Time required to save data in mem. [s] | Time required to save data in file [s] | Time rate between both saving methods |
|---|---|---|---|---|---|
| 1 | No | $10^7$ | 0.26 | 9.52 | 37× |
| 2 | No | $2.5 \times 10^7$ | 0.69 | 27.27 | 39× |
| 3 | No | $5 \times 10^7$ | 1.47 | 48.39 | 33× |
| 4 | Yes | $10^7$ | 0.25 | 11.25 | 45× |
| 5 | Yes | $2.5 \times 10^7$ | 0.71 | 28.23 | 40× |
| 6 | Yes | $5 \times 10^7$ | 6.36 | 322.06 | 51× |

We can notice that the time employed to save the objects to file is huge in comparison to saving data to the memory. Since directly saving the collected data is extremely expensive, it might be a good approach to postpone the storing of the events to phases in which the application is in idle state, in other words, *when it is not being actively used by users*. We expect that by limiting the amount of resources used simultaneously with user operations we can decrease the impact of monitoring overhead on the system response time.

The CPU usage level could be an useful indicator to recognize if the application is under heavy use or not, since its values relate proportionally

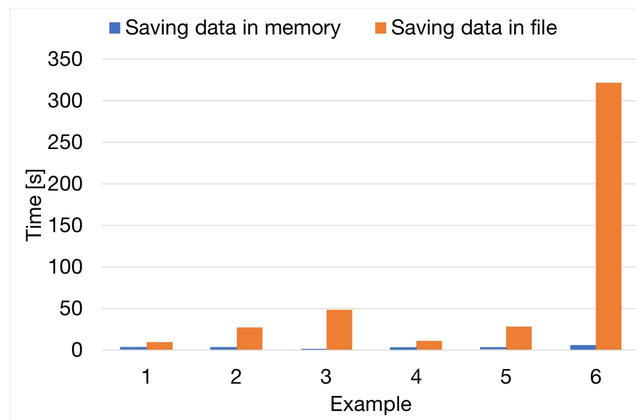



**Figure 4.1:** *Time performance comparison when saving to memory / file.*

with the amount of processing requested by a certain running application. So, among the different resources on a computer, we rely on the CPU to determine when the application moves from working to idle state.

In Figure 4.2 we introduce an example of the CPU usage level of the execution of a certain application over time. While the execution goes forward, it is possible to identify periods in which the CPU usage is high, because the application is processing user requests, and others where the CPU usage is almost negligible, since the application has finished processing user requests.

In the same figure, we show the CPU usage level for the same execution with monitoring added, represented by the dotted line. In this case the monitor works by intercepting all user events and saving them directly to files: from now on this monitoring strategy will be known as the *Direct Saving* monitor.

From Figure 4.2 we can distinguish a *working* phase, represented by the gray area, in which any additional processing time might have an impact on the system response time, and a *idle* phase, represented by the white area, a period in which the user does not actually interact with the application, the program has finished processing the user request and the user preparing for the next interaction.

Our idea is to propose a monitoring strategy with delayed saving of the events observed in the field that could behave similarly to the dashed line in Figure 4.2. Particularly, we note that this strategy would reduce the impact on the system response time by shifting the load of storing events

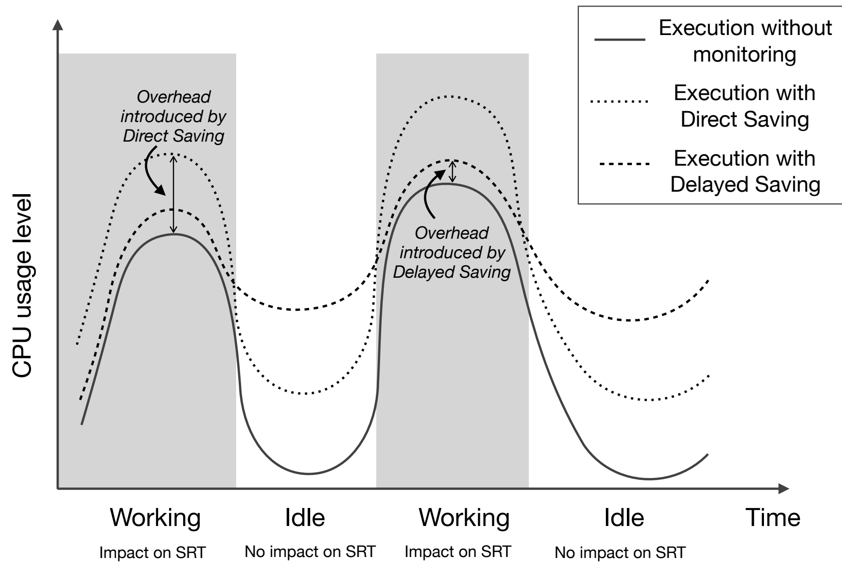



**Figure 4.2:** *Comparison of performances for different monitoring approaches.*

to file to the idle phases of the execution. Since the application is using few resources, the load of saving data to file is likely to not be perceived by users. This monitoring strategy is called *Delayed Saving*, or *DS* for short.

While the application is being monitored, the intercepted events are saved on a buffer, and then recorded in files only when an idle phase is detected. In the buffer *DS* saves an identifier for the event (i.e., the method call) and the references to the objects passed as parameters. The actual values of the parameters are retrieved only when the application comes back to an *idle* state.

However, this strategy could represent a risk for the accuracy of data: in the moment *DS* actually saves the data to files from the buffer, the data associated with the event (i.e., the objects passed as parameters) could (1) have changed or (2) have been removed from the memory.

Still, if the percentage of saved data that differs from the data recorded with Direct Saving is low, then the problem might be negligible with respect to the opportunity of monitoring applications with low impact on the user experience.

Delayed Saving recognizes the passage from the working state to the idle state using a CPU usage level threshold, that is, if the CPU usage

level goes below the threshold this means that the application is in an *idle* phase. On the contrary, Delayed Saving identifies a *working* phase each time a new user request is detected (e.g., the user clicks on a button) to be reasonably sure that we are not interfering with the user experience.

Using a very low threshold allows to minimize the probability of interfering with the user experience. In this way, the most expensive monitoring activities would be performed only when the application is under light use.

## 4.2 Delayed Saving Parameters

The impact of Delayed Saving on the system response time depends directly on the efficiency of the technique to identify idle states of the application: in this section we explain the set of parameters that impact on the capability of the technique to determine the state of the application under monitoring. Especially, we identified the CPU Usage Percentage, the Sampling Frequency of the CPU and the Threshold Level of the CPU.

### 4.2.1 CPU Usage Percentage

The CPU Time is the time taken by the CPU to execute instructions related to a process. The CPU Time does not include the time spent waiting for I/O operations nor the execution of other processes [66]. In general, the CPU Time is obtained as the sum of the *User Time* and the *System Time*.

The User Time is the time that the CPU spends in executing the operations related to a certain program. Instead, the System Time is the time taken by the CPU to elaborate operations related to the operative system triggered by a certain program.

The **CPU usage percentage** of a program during an interval $[t_1, t_2]$ can be determined as follows:

$$cpuUsage = \frac{cpuTime_2 - cpuTime_1}{t_2 - t_1} \times 100 \qquad (4.1)$$

Where $cpuTime_t$ is the observed CPU time in at instant of time $t$. Thus, CPU Usage Percentage ranges from 0% to 100% per number of cores. In our case, it ranges from 0% to 600% because we use a machine with 6 cores for the experiments.

### 4.2.2 Sampling Frequency of the CPU

The second parameter of Delayed Saving is the *Sampling Frequency of the CPU*, that is the frequency *DS* checks the CPU value to distinguish idle and working phases. As a general rule, if we increase the sampling frequency we also increase accuracy in determining the program behavior, and as a result the monitor is activated with higher precision.

On one side, if the sampling frequency is too high there exist the risk that the procedure itself to reveal the state of the CPU value introduces a harmful overhead into the user experience.

On the other side, if the sampling frequency is too low there exist the risk that the revealed behavior is not consistent with the real behavior of the application observed in the field. The main drawback is that the monitor might be activated at the wrong time, and therefore impacting negatively on the user experience.

There is a trade-off between low and high values of the sampling frequency, since a high frequencies can lead Delayed Saving to more precise results, but with higher levels of overheads, and too low frequencies may lead Delayed Saving to low overhead levels, but also less precise results.

### 4.2.3 Threshold level of the CPU

The third parameter is the value of the threshold that we use to determine an idle phase during runtime.

Using a very low threshold allows to minimize the probability of interfering with the user experience. In fact, in this case, the most expensive monitoring activities would be performed only when the program is actually doing very little or no work.

On one side, there is a risk that the process of emptying the buffer never gets activated, since the use of the CPU by the program could remain in most cases always higher than the threshold level. On the other side, a too high threshold level would allow to immediately save the information of the status of the program, but it would increase the risk of annoying the user. For these reasons, a fine tuning on each application under monitoring is required to select the correct threshold to be used.

Equivalent to other parameters, it is necessary to establish a fair compromise between the level of accuracy of the saved information and the possible overhead introduced by the technique itself.

## 4.3 The Delayed Saving Process

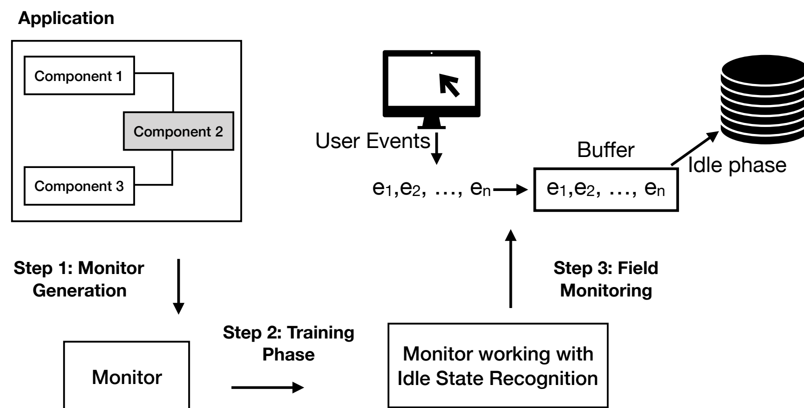



**Figure 4.3:** *Delayed Saving Process*

In this section, we present the process behind the Delayed Saving technique. The process is composed of three steps shown in Figure 4.3: the Monitor Generation, the Training and the Field Monitoring step.

### 4.3.1 Step 1: Monitor Generation

The first step of Delayed Saving Process consists of specifying the monitoring target. As monitoring targets, it is possible to consider packages, components or set of classes of the application under monitoring.

The current implementation of Delayed Saving is based on Aspect-Oriented Programming [45], that is basically a programming paradigm for modularizing cross-cutting concerns. It does so by adding additional behavior to existing code, an *advice*, without modifying the code itself, while separately specifying which code is modified via *pointcuts* specification.

We exploit Aspect-Oriented Programming to define rules that enable the Delayed Saving monitor to intercept all the events, in this case method calls, related to the component selected as a target.

### 4.3.2 Step 2: Training

The second step is about training and calibrating the technique. Especially, Delayed Saving needs to find the right threshold level of the CPU for the application under monitoring. Choosing the right threshold for deciding whether to activate or not *DS* is one of the most sensitive aspect of this technique, since a poorly chosen threshold may lead to elevated overhead values. We propose two approaches for empirically choosing the monitoring threshold value: the *Timing Classification* and the *Behavioral Classification*.

The Timing Classification approach treats the problem of searching the right threshold value as an optimization problem, in which we maximize the amount of CPU sample values classified correctly, *according to whether they occurred in working or idle phase*.

Instead, the Behavioral Classification searches the right threshold value by representing the ideal behavior of a monitor into a string, and then searching for the threshold that represents the most similar string compared to the ideal one. In other words, *this procedure looks for the threshold that makes DS to be activated correctly only when the application is in idle phase*.

Since each approach studies a different aspect of the behavior of the application under monitoring, we decide to study both before choosing the right threshold value.

Below, we give specific details about each training procedure for choosing the threshold level of the CPU.

#### Timing Classification

The Timing Classification is an approach for defining the right CPU threshold value that helps our technique to decide whether to activate or not the delayed saving of the events.

The approach focuses on classifying the different CPU samples as correctly or incorrectly classified: if a CPU sample is classified as a sample that belongs to an idle phase, but in reality the sample was taken when the application was processing user requests then the sample is **incorrectly classified**, oppositely if the application was actually doing nothing and waiting for a new user input then the sample is **correctly classified**.

This optimization problem aims to find the threshold value that maxi-

mizes the amount of **correctly classified** values. It works by systematically proving different threshold values in an incremental way (e.g., covering the range $0 - 600\%$) until it finds the optimal CPU threshold level for the application under analysis.

The Timing Classification optimization problem receives as parameters the sets of CPU samples idle and working: (1) the set of idle CPU samples are obtained by running the application without any monitoring, and it represents the ideal CPU usage level when the application is not processing user requests, they are recorded between the end and the beginning of two actions. (2) The set of working CPU samples are obtained by running the application with the Direct Saving monitor, and it represents the CPU usage level that is annoying for users, these CPU values are recorded between the beginning and the end of an action.

The sets of CPU samples idle and working should be collected by performing representative usages of the application, that is, consistent usages in line with those performed in the field.

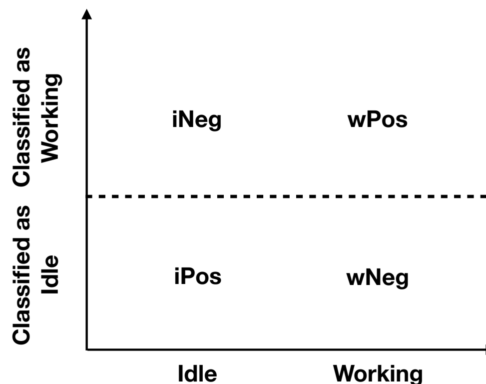



**Figure 4.4:** *Timing Classification problem.*

The procedure establishes a threshold value to begin with, then picks up a CPU sample from the idle/working set and according to its CPU usage level and the selected threshold value, the CPU sample is classified as idle or working. Since we already know if the CPU sample was obtained in idle or working phase, we can assess whether is *correctly* or *incorrectly* classified.

We introduce four indicators to quantify the occurrences of each type

of CPU sample (see Figure 4.4): we call *iPos* to the number of correctly classified CPU samples taken from the original idle set and *iNeg* to the number of incorrectly classified samples. Similarly, we call *wPos* to the number of CPU samples from the working set classified correctly, and *wNeg* to the number of CPU samples from the working set classified incorrectly.

The Timing Classification percentage for a certain threshold value is estimated by the following formula:

$$timingClass(idle, working) = \left(\frac{iPos}{iPos + iNeg} \times \frac{wPos}{wPos + wNeg}\right) \times 100$$

The procedure is repeated with a different threshold value until it finds the optimal threshold for the application under monitoring.

#### Behavioral Classification

The Behavioral Classification is another approach for defining the right CPU threshold to be used at runtime. While the Timing Classification focuses on the *timing* of the CPU sampling and the correctness of the classification of these samples, the Behavioral Classification works by selecting the CPU threshold value according to the *behavior* of the monitor with respect to the idle and working phases. In particular, the aim of this procedure is to find the CPU threshold that could represent the **ideal behavior** of a monitored application, that is, activating the delayed saving of the events only when the application is in idle phase (i.e., the **real behavior** is the one we actually recognize from a monitor using a certain threshold level).

An option to represent the behavior of a monitored application is to use strings. For example, assume having a sequence of two CPU samples associated to a working phase and then four CPU samples associated to an idle phase, the ideal behavior of the monitor should be **001111**, where **0** indicates the monitor is not saving events to file based on the value of the current sample, while **1** indicates the monitor is writing events to file.

The idea of this step is to search for the CPU threshold values that produces the most similar behaviors between the ideal sequence and the observed one. The procedure receives as parameters the sets of CPU samples idle and working, the semantic and the procedure to obtain these sets is almost similar to the one presented in Section 4.3.2, the only difference

is that the working set is recorded without any monitoring, because we are looking for the ideal behavior that should have the monitor when working in the field.

The equation we use to estimate the similarity between both behaviors is the following one:

$$behavioralClass(idle, working) = \frac{1}{2} \times \left(\frac{iEqual}{iLength} + \frac{wEqual}{wLength}\right)$$

Where *iEqual* is the number of equivalent characters associated to the idle phase when comparing the real behavior with the ideal one, *iLength* is the total number of characters during the idle phase. *wEqual* is the number of equivalent characters associated to the working phase when contrasting the two strings, and finally *wLength* is the total number of characters related to the working phase. The maximum value we can obtain for *behavioralClass* is $1$ and represents a total similarity between the case observed in the field and the ideal one. Oppositely, a value of $0$ represents no likeness between both them.

For instance, let the string **111000111011** be the ideal behavior of the monitor, and **100010110101** the behavior observed during monitoring for a threshold value $th_i$. In this case, the *behavioralClass* is $0.5$ since $iEqual = 4$, $iLength = 8$, $wEqual = 2$ and $wLength = 4$.

The main challenge of Delayed Saving is the precision of the collected data, either in terms of missing data (references that do not longer exist in memory) or unsound data (data that has changed its value between the time of collection and the time data is saved).

### 4.3.3 Step 3: Field Monitoring

We deploy the application under monitoring with Delayed Saving incorporated, now with the ability of determining when the application passes from a working to an idle phase.

The main mechanism of Delayed Saving is based on the introduction of a buffer component for saving temporarily the events intercepted by the monitor, as shown in Figure 4.5.

Contrary to a monitor that stores the events immediately, all the events are first saved in the buffer and then, when the monitor detects an idle

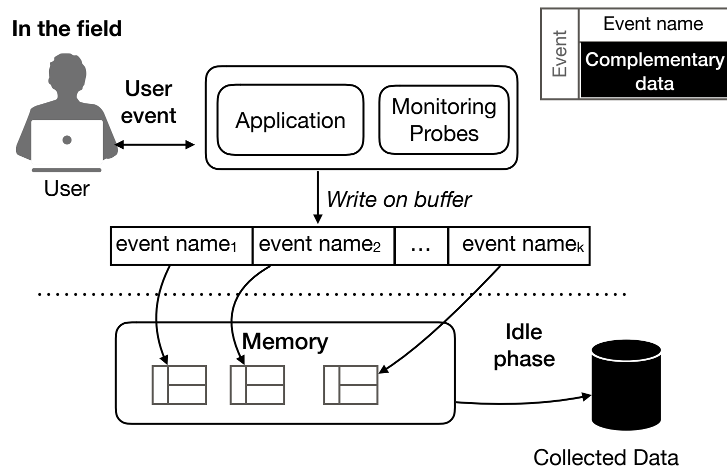



**Figure 4.5:** *Using Delayed Saving in the field.*

state of the application, the monitor starts emptying the buffer and saving each event to a permanent memory.

The ability of the monitor to empty the buffer is directly related to user interactions, meaning that if a user input is detected during monitoring, the buffer emptying process comes blocked until a new idle state is acquired.

We consider the monitoring of tracing method calls and the runtime value of its parameters.

In this case, in the buffer *DS* saves the *method call and references to the objects passed as parameters* (i.e. complementary data). The actual values of the parameters are retrieved *only when the application is in idle state*.

## 4.4 Empirical Validation

This section presents the empirical evaluation we performed to assess Delayed Saving, with respect to Direct Saving monitoring approach.

We provide information about the details of the experiments, the research questions and the results we obtained for Delayed Saving.

### 4.4.1 Goal and Research Questions

With this experimental evaluation we aim to answer the following research questions:

- **RQ1: Does Delayed Saving introduce a smaller overhead compared to *Direct Saving*?** In this research question, we measure the performance of our technique with respect to a technique that does not delay the saving of the events to file.
- **RQ2: Is the data collected by Delayed Saving accurate?** With this research question we aim to investigate if the traces produced by Delayed Saving are reliable from practitioners point of view.

### 4.4.2 Experiment Design

The main idea of the experiment is to contrast Delayed Saving with Direct Saving, which traces and records events at the time they occur. We make a comparison between both techniques with respect to the performance and quality of data.

For assessing *performance*, we measure execution time of a program monitored with both approaches compared to a program executed without monitoring. The overhead is measured with respect to the user actions, meaning that the measure starts with the user interaction and ends when the program has finished processing the request.

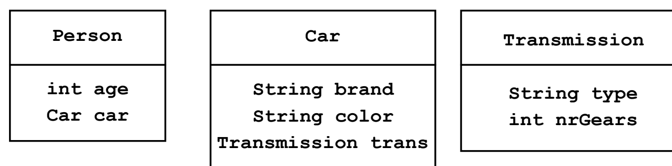
Consistently with the experimentation in Chapter 3, we analyze the overhead with respect to the different categories of operations of the system response time categorization (see Section 3.3.2).

To determine the difference between the data obtained with Delayed Saving and Direct Saving, we defined three indicators for assessing data quality: Recorded, Missing and Unsound. **Recorded** indicates the percentage of data collected with Delayed Saving that is in common with the data collected through Direct Saving. The **Missing** indicator measures the percentage of data that is lost when monitoring with Delayed, in comparison to Direct Saving. Finally, **Unsound** measures the percentage of data monitored with Delayed that has been modified in comparison to the traces monitored with Direct Saving.

The monitoring objective for both monitors is to collect the method calls

related to a certain execution, including the runtime value of the parameters associated with each method call.

We have seen that the usefulness of field data on post-deployment analyses depends on the level of detail of data [41]. Therefore, we decided to implement Delayed Saving with the ability to trace three different levels of collected data with respect to the referenced objects in the single events. The three levels are: the Objects, the Objects + Attributes and the Objects + Attributes Recursively.




**Objects level**

Person = [age: 39; car = Car]

**Objects + Attributes level**

Person = [age: 39; car = [brand = Fiat; color = white; trans = Transmission]]

**Objects + Attributes Recursively level**

Person = [age: 39; car = [brand = Fiat; color = white;
trans = [type = manual; nrGears = 5]]]

**Figure 4.6:** *Example of DS traces.*

In Figure 4.6 we present an example of the traces produced by Delayed Saving for each level. Let the classes `Person`, `Car` and `Transmission` be part of the domain of a certain application. If the Delayed Saving monitor trace an event that relates to an object of type `Person`, then we might expect three different outputs according to the level of collected data. For instance, if we are in the *Objects level*, then the trace contains attributes information only of the original object of type `Person`.

Later, in the *Objects + Attributes level* Delayed Saving saves the values of the *car* attribute, since the idea of this level is to trace information about the attributes of the objects in Objects level.

The last level, the *Objects + Attributes Recursively level*, goes a step forward and traces information about the attributes of the attributes of

the objects in the Objects + Attributes level. If our object *car* contains an attribute of Transmission type, we should trace the value of the attributes of this particular object.

As explained in Section 4.2.2, to assess the sampling frequency that ensures the best cost-benefit relation between performance and quality of data, we experiment three distinct sampling rates: **20**, **200** and **1000** milliseconds.

### 4.4.3 Experimental Subject

To answer the research questions, we use the Eclipse IDE [24] as subject of the experimentation. We considered the *JDT Plugin* as the target of monitoring, in particular, this plugin provides all the necessary tools for Java development. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools.

We have written an automatically executable test case that encodes a typical usage scenario for Eclipse, representing common activities for Java developers such as instantiating new classes, performing searches, cleaning and refactoring of code within the IDE.

In particular, the test case consists of the creation of a new Java Project with three classes. Then, for some of the classes are generated automatically getters, setters, constructor, `toString` method, and main method. Next, two searches are performed inside the project, together with the refactoring and cleaning functionality. Finally, it generates the documentation of the project along with a last refactoring which is a modification of the project's name.

To avoid non-determinism and for reproducibility of the test case we implemented it with the Sikulix [36] capture and replay tool.

### 4.4.4 Training Step

Since Delayed Saving is a technique whose performance depends on the selected threshold level of the CPU, we apply the training step explained in Section 4.3.2, that is, to use the *Timing Classification* and *Behavioral Classification* procedures for finding the right threshold value.

### Using the Timing Classification Procedure

**Table 4.2:** *Selection of optimal threshold level of the CPU using timing classification procedure.*

| Phase | Size | Mean | Median | 1st Quartile | 3rd Quartile | Upper Whisker |
|---|---|---|---|---|---|---|
| Idle | 71,236 | 45.07% | 10.74% | 0.00% | 67.22% | 168.04% |
| Working | 1,944 | 166.00% | 150.07% | 97.95% | 241.23% | 441.71% |

First, we attempt to find the right threshold value applying the Timing Classification procedure explained in Section 4.3.2. To use this optimization procedure we need both set of CPU samples for the working and the idle phases. To obtain these sets, we ran the test case and classified the samples depending if they were obtained during the execution of a functionality (working) or if they were obtained when no functionality was being executed on the application (idle). In Table 4.2 we report some descriptive data about each sample group, in particular we report 71,236 samples for the idle phase with a mean value of 45.07%, and 1,944 samples for the working phase with a mean value of 166.00%.

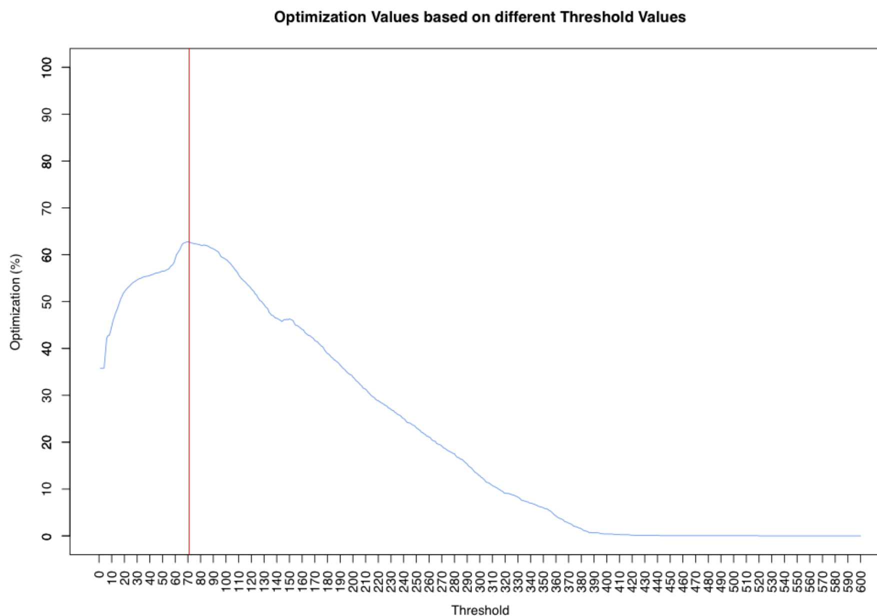



**Figure 4.7:** *Threshold selection using the Timing Classification procedure.*

Then, to find the right value we tested 600 threshold values going from 0% to 600%. In Figure 4.7, we show the different optimization values we

found for the threshold values of the CPU. Additionally, we show with a red line the value that maximizes the amount of *correctly classified* values for the corresponding threshold, especially we observe that the highest optimization value is 62.78% that reflects on a threshold value of 71%.

### Using the Behavioral Classification Procedure

**Table 4.3:** *Selection of optimal threshold level of the CPU using Behavioral Classification procedure.*

| **Iteration** | **Threshold** | **Optimization** |
|---|---|---|
| Execution 1 | 69 | 74.27% |
| Execution 2 | 66 | 74.09% |
| Execution 3 | 66 | 74.10% |
| Execution 4 | 71 | 73.89% |
| Execution 5 | 81 | 73.95% |

As explained in Section 4.3.2, this approach is based on the representation of an execution of a monitored program as a string, said string contains the information about the status of the monitor. The idea is to find the threshold value that maximizes the similarity between the *ideal* behavior of a monitor and the string produced according to the selected threshold value.

In Table 4.3 we show for each execution of the task the threshold value that optimized the string similarity. Especially, we report a mean threshold value of 71% with a 62.78% of optimization.

Since both procedures, Timing and Behavioral Classification, deliver a similar result, we proceed to use the 71% as the calibrated threshold value for the CPU to use in our experimental setup.

## 4.5 Results

In this Section we present the results of the experiment we conducted to evaluate Delayed Saving. All the experiments were executed on a computer running macOS version 10.12.6 with a 2.7 GHz Intel Core i5 processor and 16 GB of RAM.

### 4.5.1 Performance Results

To answer the research question **RQ1: Does Delayed Saving introduce a smaller overhead compared to *Direct Saving*?** we assessed the impact of both monitors in terms of the introduced monitoring overhead, in particular we measured the overhead introduced by each approach compared to the program executed without monitors. We ran our test case five times and estimate mean values to mitigate any effect due to non-expected variations of the execution time within the execution environment.

Overall, we gathered near 780 samples through the five executions of the test case for each setup. Additionally, we also saved the execution traces that will be useful for answering the research question **RQ2**.

As we did in Chapters 2 and 3 we classified user actions according to its system response time category, i.e. Instantaneous, Immediate, Continuous and Captive actions. We distinguished 35 Instantaneous, 12 Immediate and 1 Continuous actions. During the experimentation we did not identify Captive actions.

Below, we present the performance results for both monitoring approaches with respect to the different system response time categories. We divide results also according to the level of collected data.

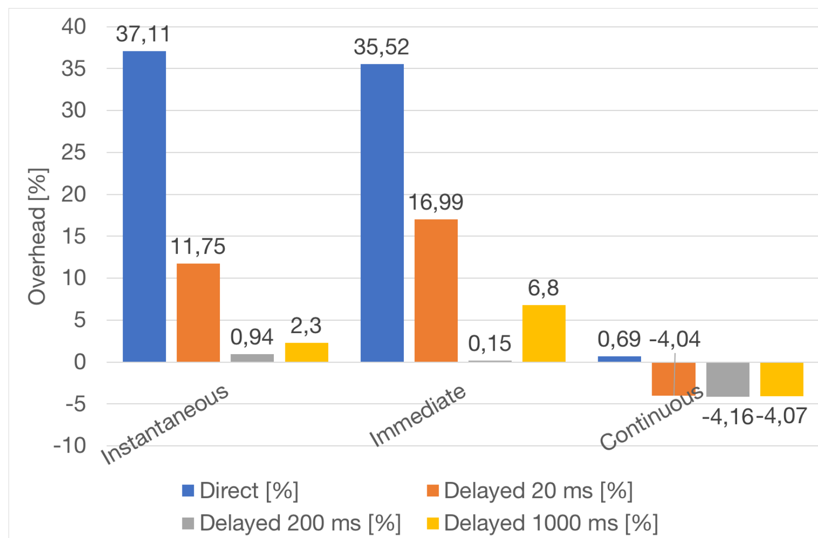



**Figure 4.8:** *Performance results for Objects level.*

For the Objects level, shown in Figure 4.8, the first thing we notice is that Delayed Saving improves considerably the performance over Direct

Saving with respect to actions that employs less than 1 second in the SRT Category, in particular we observe an improvement of approximately 32% for Instantaneous actions and a 15% for Immediate actions.

The second noticeable fact is that for actions taking longer than 1 second, or Continuous actions, the performance is almost comparable between Delayed Saving and Direct Saving.

When it comes to the sampling rate, we noticed that for Instantaneous actions the lowest overhead was introduced at 200 milliseconds (0.94%). We observed a similar behavior for Immediate actions (0.15% of overhead at 200 milliseconds) and for Continuous actions (-4.16% of overhead at that frequency).

We noticed negative monitoring overhead values for Continuous actions, the reason of these values might be related to the fact that there is only one action for this category, we would need to identify more Continuous actions to make these results more stable.

For this level, the traces were of approximately 300 MB each.

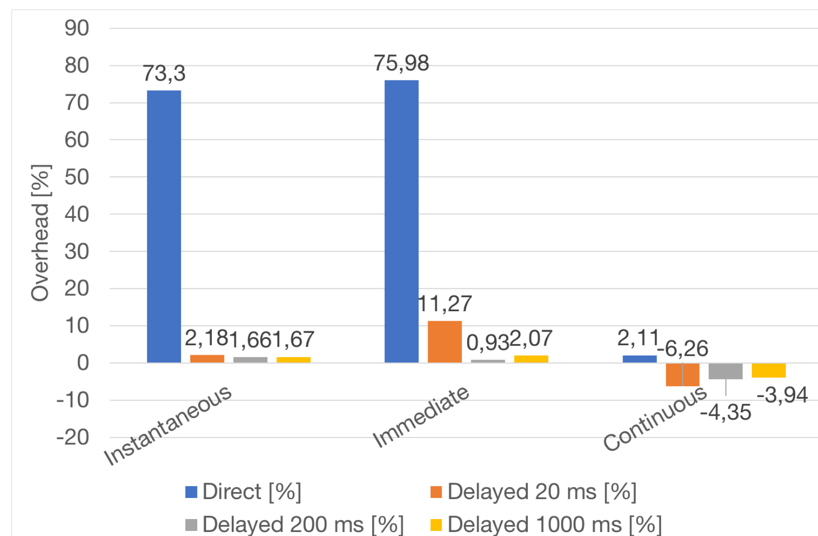



**Figure 4.9:** *Performance results for Objects + Attributes level.*

The performance results for the Objects + Attributes level (see Figure 4.9) shows that Delayed introduced again a noticeable improvement with respect to the Direct Saving technique. The improvement for Instantaneous actions was about 71%, for Immediate actions was 68%, and for Continuous actions was around 5%. On the overall for this level, the best performance

observed for the sampling rate was obtained again for 200 milliseconds.

In this level, the traces were of approximately 700 MB each.

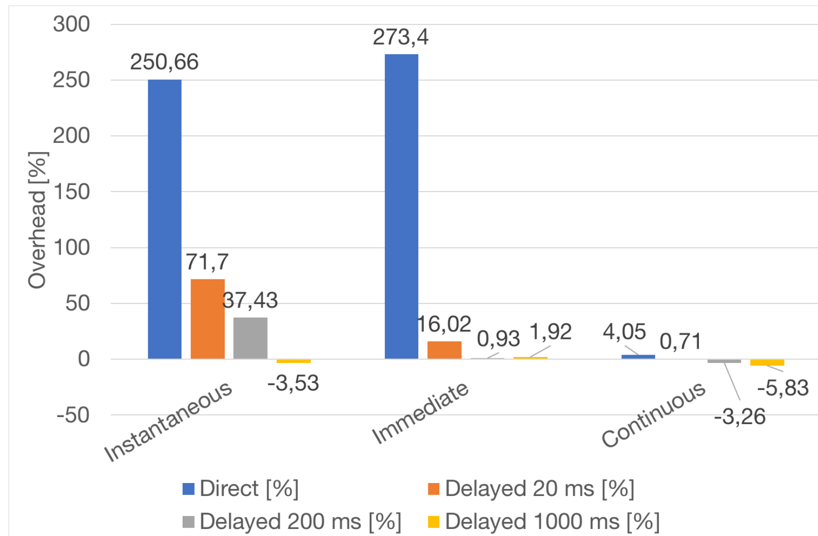



**Figure 4.10:** *Performance results for Objects + Attributes Recursively level.*

In the Objects + Attributes Recursively level (Figure 4.10) we again observe a big difference between Direct Saving and Delayed Saving results. In particular, for Instantaneous and Immediate actions we observe a difference of approximately $> 180\%$ between both approaches, instead for Continuous actions we noticed a difference of about 2% between Direct Saving and Delayed approaches. Regarding the sampling rate, with exception of Immediate actions, we obtained the best performance when monitoring with 1000 millisecond sampling rate, for Immediate actions the best performance was with 200 milliseconds (0.93%).

For this level, the traces were of approximately 3 GB each.

About the optimal choice of the sampling rate for performance, we can conclude that it does not depend on the amount of collected data during monitoring. In all three levels the optimal choice was 200 milliseconds, only with the exception of Instantaneous actions for the deepest level. For the Objects level, the 200 milliseconds configuration had an improvement of approximately 10% for Instantaneous and Immediate actions with respect to the 20 milliseconds setup. Then, for the Objects + Attributes level the 200 milliseconds approach had a 10% of performance improvement with respect to the 20 milliseconds setup of Immediate actions. Finally, in the

Objects + Attributes Recursively level the 200 milliseconds configuration had a 15% improvement with respect to the 20 milliseconds approach for Immediate actions.

### 4.5.2 Quality of Data Results

For answering the research question **RQ2: Is the data collected by Delayed Saving accurate?** we measured the three indicators for data quality: Recorded, Missing and Unsound. We divide results according to the level of collected data.

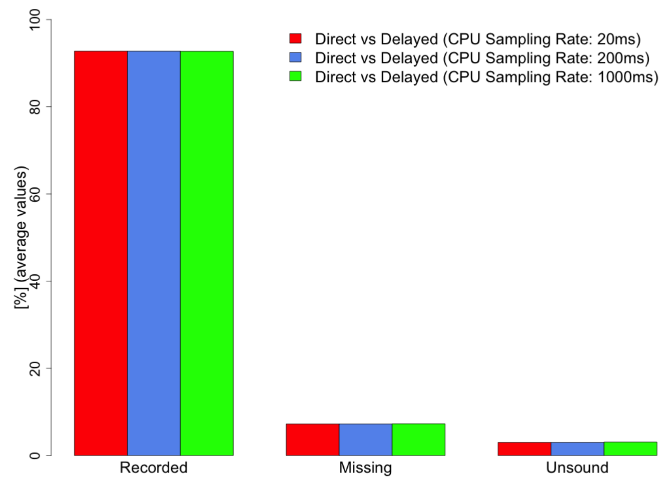



**Figure 4.11:** *Quality traces comparison for Objects level.*

When considering the Objects level (see Figure 4.11), we notice that results do not vary too much between the different sampling frequencies, whether for Recorded, Missing and Unsound indicators. However, we observe a high percentage of Recorded data (greater than 90%) and a very low percentage of Missing and Unsound data (less than 10%). Meaning that the traces captured by Delayed Saving are accurate, even when data related to events are saved in a second step.

When analyzing the second level, or the Objects + Attributes level in Figure 4.12, the first thing we note is that the level of Recorded drops quickly from 90% to approximately 80%. And that Missing and Unsound increases approximately 15% and 12% with respect to the Objects level.

Then in the third level, or the Objects + Attributes Recursively level in

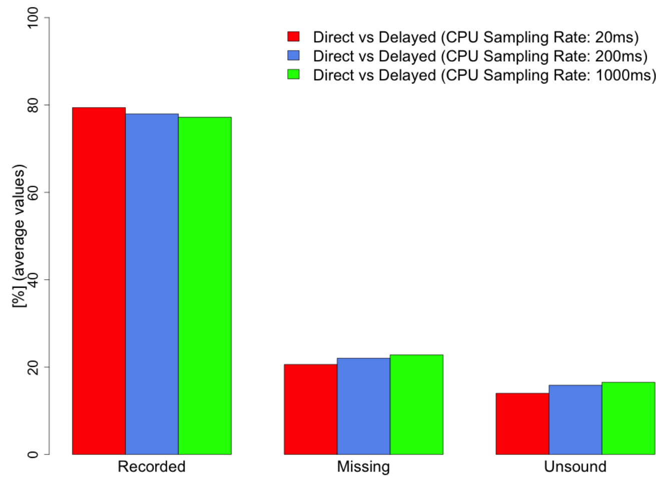



**Figure 4.12:** *Quality traces comparison for Objects + Attributes level.*

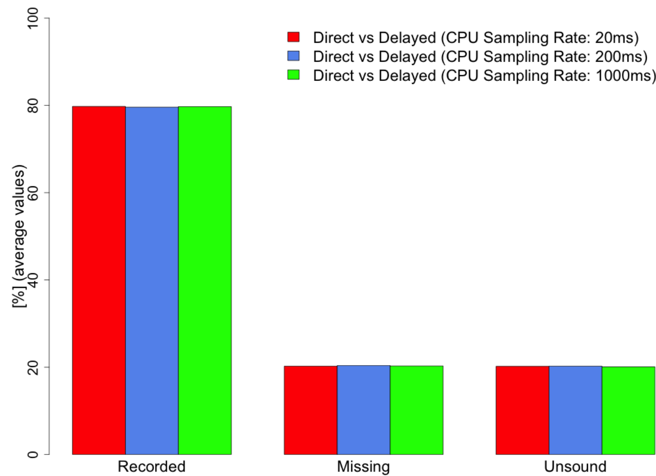



**Figure 4.13:** *Quality traces comparison for Objects + Attributes Recursively level.*

Figure 4.13, we notice that again results do not change much between the different sampling frequencies of the CPU, but that in general indicators remain on average pretty similar to the Objects + Attributes level.

Regarding the sampling frequency and the different levels of collected data, we can say that results are pretty similar. The only exception is when considering the Objects + Attributes level. In this case, the best results were obtained considering a sampling frequency of 20 milliseconds, while

the worst results were obtained when we sampled the CPU each 1000 milliseconds.

In general, we notice that the **Unsound** indicator increases as the level of collected data goes deeper, this is an expected result, since considering a deeper level of data means more details about a certain object, and more data that needs to be compared.

## 4.6 Discussion

We performed an experimental evaluation for evaluating the quality of Delayed Saving in terms of monitoring *performance* with respect to Direct Saving.

When assessing performance, we discovered that the monitoring overhead introduced by Delayed Saving was in all cases better than Direct Saving. In particular, we observed an improvement for Instantaneous actions of 36%, 71% and 213% for each level of collected data, respectively. Then, for Immediate actions we observed an improvement of 34%, 75% and 272% for the three levels of level of data collection. Finally, for Continuous actions we observed a slight improvement of 4%, 6% and 7% for the three levels of collected data, respectively.

Given that Delayed Saving works by delaying the recording of the objects related to the monitored events, it is possible that the data associated with the object might have changed its value or just being removed from the memory. This was the reason to also study the *accuracy* of the traces produced by Delayed Saving.

The quality of data produced by Delayed Saving depends directly on the level of precision of data collected from field, we discovered that the accuracy of the technique was approximately 92%, 77% and 79% respectively for each level of collected data. Additionally, we found that the percentage of unsound data was 3%, 15% and 20% for each level respectively. Even though we are considerably increasing the amount of collected data from level Objects + Attributes ($\sim$ 700 MB) to level Objects + Attributes Recursively ($\sim$ 3 GB), the accuracy of the technique remains unchanged, stabilizing near 77%. The imprecision we observed in the experimental results might not be tolerated by all the techniques working with field data, but it could probably be accepted by techniques that work on static models and

mining techniques, where absolute precision is not the most important aspect.

In the overall, *DS* presents a good compromise between performance and quality of data: on the one hand, the results suggest that strategy succeeds to decrease the monitoring overhead with respect to the actions of the system response time categories, on the other hand the results suggest a good quality of data since the percentage of saved data with Delayed Saving that differs from data recorded with Direct Saving is low. Delayed Saving could be useful for practitioners at the time of choosing a monitoring technique for analyzing instances running in the field.

# Chapter 5

# Software Monitoring

In this chapter, we frame our work in the context of related areas.

In particular, Section 5.1 discusses how this thesis relates to other studies about the perception of the system response time, and Section 5.2 discusses how this Ph.D. work relates to existing approaches to field monitoring and analysis.

## 5.1 Perception of the System Response Time

When performing analysis activities (e.g., profiling, testing, program analysis, among others) in the field, we are actually modifying the system response time of user interactions. Since the resources on a computer machine are shared between all processes, the analysis processes may introduce slowdowns to user processes, affecting the overall user experience.

Investigating how the overhead introduced by monitoring may affects the user experience implies understanding how users perceive the system response time (i.e., which is the expectation that a user has regarding the response time of a certain action) and how they react to variations in the reactivity of the system.

In computer science and Human Computer Interaction literature, or *HCI* for short, there are two main studies regarding the system response time and its evaluation by the end user, the one by Shneiderman ([71, 72], 1987 and 2009) and the one by Seow ([70], 2008). In the first one, the evaluation of the foreseen SRT is task oriented, so, for simple tasks (mainly typing and moving the mouse), the system must respond essentially as quickly

as possible, while as task complexity increases, users should better tolerate the delay in system response. So, the complexity of the task rules in determining the appropriate range of system response times.

In the second classification for the SRT, Seow proposed that interactions with a computer can be assigned to categories based on user expectancies, in a "conversation like" interaction. So, considering what the user is asking the system (the requested action), he will likely tolerate different SRT.

Anyway, the resulting classifications are quite similar, as shown in Table 5.1: since the Shneiderman is a little more recent (2009), it reflects the improvements in computer performances (converted in a quite less tolerant expected response time), but mainly the categories are the same four.

**Table 5.1:** *Shneiderman's and Seow's SRT categorization*

| **Shneiderman** | | **Seow** | |
|---|---|---|---|
| **Task** | **SRT** | **Category** | **SRT** |
| Typing, mouse movement | 50 - 150 ms | Instantaneous | 100 - 200 ms |
| Simple frequent tasks | 1 s | Immediate | 0.5 - 1 s |
| Common tasks | 2 - 4 s | Continuous | 2 - 5 s |
| Complex tasks | 8 - 12 s | Captive | 7 - 10 s |

Nevertheless, these two classifications of the SRT both show limitations: the Seow one is no longer updated with respect to new hardware evolutions and user capabilities (so, the expected time could not be so reliable), and furthermore it is not always simple to classify an action (and again, we cannot be sure that the user is completely aware of the complexity of the action), while the Shneiderman one does not clearly state how to define the task complexity (and anyway, it suffers of the same time estimation of the Seow's one, since it was updated ten years ago).

About the *perception of the SRT*, there are multiple studies in the context of the research in *HCI* and Psychology. For instance, the perception of time is a complex subject of study in psychology and several researchers investigated it.

Relevant to our experiment, Duncan et al. [23] studied how the perception of time changes with age discovering that young people, adults and elderly people perceive time differently.

This is why we selected subjects of similar age. Our study is thus representative of how young people perceive the overhead, but we do not know if the obtained results are valid for adults and elderly people.

In the *HCI* area, there are a number of studies about the satisfaction of the user concerning the SRT of an application.

For instance, an interesting result from the *HCI* ([46]), performed in 1987 but still valid (since it was a study of the human physical ability to estimate weight), shows that users are unlikely to be able to identify time variations inferior to 20% of the original value: this information could be useful to forecast if the user will tolerate the delay or not, if in the 20% range.

Then, Ceaparu et al. [13] studied how interactions with a personal computer may cause frustration. In the experiment, users were asked to describe, in written form, frustrating experiences with computers. The participants of the experiment declared that applications not responding in an appropriate amount of time and Web pages taking long time to process to be the main sources of frustration in the computer interaction. Although this experiment is different in both the design and the aim from ours, the results show that SRT delays (e.g., a less responsive Web browser) might be the cause of a bad user experience.

Some studies stressed the tolerance of the users in specific settings. For example, Nah et al. [54] investigated how long users are willing to wait for a Web page to be downloaded. Results show that users start noticing the slowdowns after 2 seconds delays and that do not tolerate a slow down of more than 15 seconds. A threshold of 15 seconds has been reported as the maximum that can be tolerated before perceiving an interruption in a conversation with an application also in other studies [57, 52].

The study conducted by Hoxmeier and Di Cesare [38] is similar to ours, both in the settings (100 subjects performing tasks on applications with different slowdowns) and in aims (to evaluate the user appreciation and perception of web applications in terms of ease of use and satisfaction experiencing different slowdowns). The experiment presents some differences with our study, mainly in the variables: (1) we are focused on studying the user tolerance of different overheads with respect to the the four types of operations, so we added a controlled overhead to each single operation, while in that study a fixed time overhead (3, 6, 9 and 12 seconds) is imposed every time a user move from a page to another (so, single operation versus moving through applications pages, fixed overhead versus percentage overhead on expected SRT) (2) we use the Seow's classification, while in that study authors refer to Shneiderman [71], (3) we focus on desktop applica-

tion, known to subjects, while in that study authors uses Web applications developed ad-hoc.

Their results show that 12 seconds seem to be the limit for the user tolerance, and that a linear relationship exists between response time and user satisfaction. This is partially coherent with our results, but they must be analyzed and compared considering the differences in the settings, and above all in the different expected time performances, since the study was conducted in year 2,000 over web applications, and the speed of computers and above all of the Internet connections is completely different today, so, the expected system response time is completely different too.

While these findings are interesting, they focus on complementary aspects compared to our experiment. In fact, we are not interested in identifying the maximum overhead that a user can tolerate, but we are interested in the overhead that users cannot event recognize. In other words, we are not interested in stressing up to the limit the users, but rather to seamlessly introduce analysis and monitoring routines in software applications. Moreover, our study covers different classes of operations, including operations to move across windows and menus and also the execution of domain functionalities, instead of focusing on specific operations (e.g., transitions between Web pages).

## 5.2 Field Monitoring

Monitoring a software consists of dynamically collecting information about its behavior. The problem of monitoring can be divided in two sub-problems: (1) obtaining data from executions and (2) defining the scope of monitoring.

Tackling the first problem is purely technical and consists of adding probes into programs to gather the data. Adding probes is not a challenge and a number of techniques can be used to add probes manually, statically or dynamically. Manual addition of probes consists of implementing the tracing code, e.g., using logging technology [1, 62]. Static instrumentation techniques automatically modify the software source or binary code by adding probes in the places specified by developers [39, 25]. Dynamic instrumentation is the same than static instrumentation, but probes are placed into applications at load-time [25].

The second problem is conceptual and focuses on the choice of the ele-

ments to be monitored. Techniques working on this topic are known as *field monitoring and analysis solutions*, and can significantly improve verification and validation methods since they can work with production data and can exploit the knowledge of the real interaction patterns between users and applications [40, 48, 27, 60, 37, 67, 41, 15]. Although these solutions can be beneficial for the quality of the software, they may degrade the performance of the applications and consequently may affect the quality of the user experience if not carefully designed.

In general, techniques that exploit and collect field data have been widely studied, and many approaches have been reported in literature.

For instance, Delgado et al. [20] developed a taxonomy to analyze and differentiate monitoring tools that focus on identifying field failures. The taxonomy categorizes field monitoring research by classifying contributions according to the elements that are considered essential for building a monitoring system, such as the monitoring mechanism that supervises the program's execution and the event handlers that intercept and save data from field. Under this taxonomy, our tool *CBR* can be classified as a monitor with: *automatic monitoring points* (i.e., dynamic probes placement), *inline placement* (i.e., that is embedded in the target code), *software platform* (i.e., that uses code to observe and analyze the values of monitored variables) and *single process implementation* (i.e., that the monitor executes in the same process as the target program).

The problem of *field monitoring* can be further defined in the following way: given ***EV*** as the set of every possible event that can be produced by a monitored application, a field execution $E = \langle e_1, e_2, ..., e_n \rangle$, with $e_i \in$ ***EV***, $i = 1 \ldots n$ is an ordered sequence of events. Depending on the objective of the monitoring activity, we can identify a set of relevant events that must be collected. We represent this set of interesting events (i.e., monitoring objective) as $O = \{o_1, o_2, ..., o_m\} \subseteq$ ***EV***.

Given a field execution $E$ and a monitoring objective $O$, the set of events that are ideally captured is defined as

$T_O(E) =< \bar{e_1}, \bar{e_2}, ..., \bar{e_k} >$, where:

1. $T_O(E)$ is a subsequence of $E$, that is, $T_O(E)$ can be obtained from $E$ by deleting a possibly empty set of events (*soundness*);

2. all the relevant events are recorded, that is, if an event in $O$ occurs in $E$, it also occurs in $T_O(E)$ (*completeness*);

3. only the events from $O$ occurs in $T_O(E)$ (*succinctness*).

Our work focuses particularly on the cost-effectiveness of monitoring, aimed to collect field data in a non-intrusive way without losing relevant information. In this line, some techniques have been specifically designed to limit the monitoring overhead introduced in the target application. Current approaches dealing with effective field monitoring can be classified in three mainstreams: *Distributive Monitoring*, *Probabilistic Monitoring* and *State-Based Monitoring*.

Below, we give a description for each field monitoring approach, and a comparison with our work.

### 5.2.1 Distributive Monitoring

Distributive Monitoring is an approach that takes the monitoring objective $O = \{o_1, o_2, ..., o_m\}$ and assign a subset $S \subset O$ to different computers, in a way that the union of all subsets $S$ satisfies the original monitoring objective $O$. This strategy decreases monitoring overhead because it limits the amount of behaviors observed on a certain computer.

This technique has been developed by Orso et al. [65, 10] with their tool *Gamma System* which divides a monitoring task into a set of subtasks and assigns them to individual instances of the software to be monitored, in order to minimize runtime overhead. Additionally, the technique optimizes the placement of probes across several instances for further overhead reduction.

The data collected in each instance across the field is independent from other locations, and thus is extremely hard to reconstruct field executions and having more comprehensive information about the application from the individual traces. Contrarily, *CBR* collects traces from different locations and summarizes field data into a comprehensive knowledge such as a Finite State Automaton, exploiting a representation of the program state that can be used to merge independently collected traces.

A similar approach that exploits this concept is the one by Briola et al. [12], which implements a framework for distributed runtime verification of Multi-Agent Systems (MAS). The approach is mainly in charge of monitoring interaction protocols in JADE MASs [9] by transforming agent messages into Prolog programs and predicates suitable for runtime verification.

In the same direction, Ferrando et al. [29] worked on an algorithm for decentralizing the monitoring in MASs. Since having one centralized monitor in growing MAS systems may not scale well, especially when considering complex systems, Ferrando et al. proposed to distribute and simplify the load of monitoring by partitioning the agents in several groups with one monitor per partition. The partitioning system guarantees that distributed monitoring detects all the protocol violations such as a centralized monitoring system would.

### 5.2.2 Probabilistic Monitoring

The second stream is Probabilistic Monitoring, which lowers the overhead by monitoring each instrumentation point with a certain probability, thus collecting random subsets of traces. This means that from the set of relevant events $O_1, \ldots O_k$, each event $O_m$ is observed only with a probability $P \in [0, 1]$ .

Liblit et al. [48] have taken advantage of Probabilistic Monitoring in particular to isolate bugs by profiling a large, distributed user community and using logistic regression to find the important predicates (a logical expression that evaluates to *true* or *false* that directs the execution path) that could be the faulty one.

In the same way, Jin et al. [40] presented a monitoring framework called *Cooperative Crug (Concurrency Bugs) Isolation* to diagnose production run failures caused by concurrency bugs: the framework introduces the necessary instrumentation to check whether accesses to a memory location were made by the same thread or by different threads. Moreover, the tool relies on sampling to keep the overhead low as our tool *CBR* does, the results of the study shows that the CCI framework incurs only in $2-7\%$ of monitoring overhead. This technique is focused in observing very specific concurrency problems such as races and atomicity violations, and especially is not intended for collecting comprehensive information about full executions, contrarily as *CBR* does through the generation of traces from the *FSA* model.

A work similar to Probabilistic Monitoring, but with a different domain (Runtime Verification), is the one of Bartocci et al. [7]: they presented *Adaptive Runtime Verification*, a monitoring tool that controls the overhead by enabling and disabling runtime verification of events according to overhead target levels. This framework determines statistically the probability

of an application property (based on observable actions of the monitored system) of being violated, and based on this number a higher or lower level of overhead for that property is assigned. When is not possible to verify a certain property (e.g., due to a high overhead level) the framework estimates the probability that a property will be satisfied, instead of monitoring the actual property, and thus reducing the load of runtime verification.

Runtime Verification approaches mostly assume the existence of complete execution traces, but often systems produce incomplete traces due to the use of Probabilistic Monitoring. For this reason some researchers have been working on techniques for the runtime verification of incomplete traces [43, 4]. On the one hand, Joshi et al. [43] have been developing an algorithm that identifies whether a certain property can be soundly checked in the presence of a partial trace: if the property can not be analyzed with the available data, the algorithm waits until there are enough samples from the field. In the other hand, Babaee et al. [4] worked on a prediction model to detect satisfaction or violation of a property based on incomplete executions basically, if a property can not be satisfied, the monitor estimates the probability of that property being satisfied based on the possible execution paths of the application according to the actual state.

Another type of Probabilistic Monitors is Bursty Monitors, an approach developed by Hirzel et al. in [35]. Bursty Monitors are known for collecting subsequences of events with ad-hoc strategies to construct a temporal program profile.

The strategy implemented by Bursty Monitors is the closest to our approach *Controlled Burst Recording*, because the monitoring procedure can be reduced to collect streams of consecutive events at runtime, instead of tracing full executions. The main differences with our approach are related to the fact that (1) the reconstruction of streams are made at two different levels: *CBR* is user-interaction oriented, while Bursty Monitors are procedure oriented, and (2) Bursty Monitors does not take in consideration the impact of the monitor on user operations, contrarily to the *Controlled Burst Recording* framework. We already discussed in Chapter 2 how users are more sensitive to delays experienced on operations of specific categories, so in general the monitoring overhead should be always considered with respect to the system response time classification.

In general, Probabilistic Monitoring is designed to collect partial information about the execution, while in this Ph.D. thesis we focus on the re-

construction of fairly larger traces.

### 5.2.3 State-Based Monitoring

The program state $PS$ of a certain application on a instant $t$ can be defined as $PS(t) = \{pv_1, pv_2, \ldots, pv_r\}$, where $pv_r$ represents the runtime value of a program variable on a moment $t$.

State-Based Monitoring approaches focuses on using a small subset of program variables $pv$ to represent a program state while the application is running in the field. These techniques are often used either for replaying field executions or for trace analysis (e.g., debugging).

For instance, Orso et al. [64] worked on a technique for a selective capture and replay of program executions. This technique allows to select a subsystem of interest, capture at runtime all the interactions of the applications with its subsystem and then replay the recorded interactions in a controlled environment. For each interaction of the application, the technique captures a minimal subset of the application's state and environment required to replay the execution.

Similar to *Controlled Burst Recording*, this framework exploits the idea of tracing just the entities defined inside the subsystem, that is, defining a monitoring scope in order to lower the overhead introduced by monitoring. However, this framework is designed to be used for analyzing partial executions instead of complete executions, as we do.

Diep et al. [21] presented a technique for analyzing traces produced by field applications, in particular to identify and delete irrelevant events from traces that do not offer interesting information for offline analyses.

Before deploying the application, practitioners select a subset of program variables to be used to represent the program state $PS$, then while the application is running in the field, the state of these variables is regularly saved before and after each monitored event.

In a second step (i.e., offline time), the technique divides the full trace in several pieces using the variables state as splitting points. After this operation, the technique deletes all the events that do not change the program state, and those events that whose occurrence can be re-ordered without affecting the program state, leaving in the trace only the most relevant information for understanding the program behavior.

Contrary to *CBR*, this technique does not take into account the monitor-

ing overhead introduced by the action of saving regularly state information, besides the fact that the representation of $PS$ is done manually in comparison to *CBR* that performs this operation in a automated way.

# Chapter 6

# Summary of Contributions and Open Research Directions

Fully assessing the correctness of software applications in-house is infeasible. Indeed a number of relevant cases are missed by verification and validation procedures due to lack of resources or the impossibility to replicate the scenarios that occur in the field. It is thus extremely important to be able to continue the validation activity in the field while applications are operational.

Monitoring and validation activities inevitably consume resources and, depending on their extensiveness, may significantly slow down software systems, interfering with the user activity. There is thus a challenging trade-off between monitoring and validation applications in their operational environment and preventing any degradation of the user experience. The lack of studies about *when* users perceive an overhead introduced in an application makes extremely difficult to fine tune techniques working in the field.

In a nutshell, this Ph.D. thesis presents two main challenges: (1) to study how the user perceives monitoring overhead and, (2) to investigate how to collect data in a non-intrusive way, without losing too much information.

To tackle challenge (1) this thesis work presents an empirical study

aimed at *quantifying if and to what extent the overhead introduced in an interactive application is perceived* by users. This study considers multiple *categories of operations* and sequences of operations executed in different *order*, investigating how both factors may influence the perception of monitoring overhead.

For assessing challenge (2), this thesis presents two different frameworks: first, a framework that collects data for short periods and extracts knowledge from field executions, and second, a framework that focuses on tracing users executions without actually interfering with them.

Both frameworks focuses in collecting expensive information from the field, without impacting on the user experience. We additionally carried out experimental evaluations to assess the *performance* of the techniques with respect to the system response time and the *accuracy* of the traces produced by both approaches.

## 6.1 Contributions

This thesis advances the state of the art of field monitoring of interactive applications by:

- Presenting an empirical study with human subject studies about how users perceive monitoring overhead. The study produced interesting findings that can be exploited to carefully design analysis procedures running in the field. We discovered that users did not perceive significant differences for an overhead of 80% and seldom perceived an overhead of 140%, that users are more sensitive to delays experienced on operations of specific categories, and that the execution of a long operation generates a pessimistic expectation for the future, resulting in an optimistic perception of the response time if the functionality that is later executed is faster than the previous one.

- Proposing a technique for extracting knowledge about field executions with little impact on the user experience [59, 17]. The technique generates a finite state automaton that models the relationships between events observed in the field and the different program states of the application. From the model it is possible to extract accurate and comprehensive traces that could be useful for post-deployment analysis

such as debugging, field failures reproduction, protocol verification, among others.

- Describing a technique for delaying the saving of events to file during field monitoring. The technique performs monitoring by limiting the activity performed in parallel with users operations: the approach considerably decreases the impact of monitoring on user operations and produces traces of high accuracy.

## 6.2 Open Research Directions

What concerns the human-subject studies about the impact of overhead on user experience, future work may concern considering different type of participants such as people of all ages or people with different levels of experience with interactive applications. Moreover, the addition of longer tasks to the experimentation with variable levels of overhead could be considered.

Regarding techniques extracting knowledge from field executions, we could consider studying new refinement procedures during the generation of the *Abstraction Functions*, in order to reach higher levels of accuracy in the traces produced by the Finite State Automaton.

For techniques delaying the saving of the events collected at runtime, future research may consider developing a procedure for monitoring completely accurate traces by implementing a mechanism that keeps track of the objects that are about to change during the execution.

Even though the monitoring approaches presented in this thesis work are designed with the aim of capture and observe events from instances running in the field with low overhead, Runtime Verification approaches could greatly benefit of *Controlled Burst Recording* and *Delayed Saving* techniques, since they provide a base architecture for *broad spectrum* monitoring that could be used also for verifying properties at runtime.

# Bibliography


[1] Apache. Apache log4j. `https://logging.apache.org/log4j/2.x/`, visited on 2018.

[2] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 35–42. ACM, 2002.

[3] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Transactions on Software Engineering and Methodologies*, 21(1):2:1–2:35, Dec. 2011.

[4] R. Babaee, A. Gurfinkel, and S. Fischmeister. *P*revent : A predictive run-time verification framework using statistical learning. In *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 205–220, 2018.

[5] L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER, 2010.

[6] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer, 2018.

[7] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, and J. N. Seyster. Adaptive runtime verification. In *Proceedings of the International Conference on Runtime Verification (RV)*, pages 168–182. LNCS/Springer, 2012.

[8] A. Basiri, A. Blohowiak, L. Hochstein, and C. Rosenthal. A platform for automating chaos experiments. In *ISSRE*. IEEE, 2016.

[9] F. Bellifemine, A. Poggi, and G. Rimassa. Jade: a fipa2000 compliant agent development environment. In *Proceedings of the fifth international conference on Autonomous agents*, pages 216–217. ACM, 2001.

[10] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, 2002.

[11] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.

[12] D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of jade multiagent systems. In *Intelligent Distributed Computing VIII*, pages 81–91. Springer, 2015.

[13] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human - Computer Interaction*, 17(3):333–356, 2004.

[14] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, 2009.

[15] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007.

[16] J. Cohen. Statistical power analysis for the behavioral sciences. erlbaum. *Hillsdale, NJ*, 1988.

[17] O. Cornejo. Flexible in-the-field monitoring. In *Proceedings of the International Conference on Software Engineering (ICSE) - Companion*, 2017.

[18] O. Cornejo, D. Briola, D. Micucci, and L. Mariani. In the field monitoring of software applications. In *NIER track ICSE*. IEEE, 2017.

[19] I. Corp. Ibm spss statistics for macintosh version 24.0. https://www.ibm.com/products/spss-statistics, 2016.

[20] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. volume 30, pages 859–872, 2004.

[21] M. Diep, S. Elbaum, and M. Dwyer. Trace normalization. In *Proceedings of the Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[22] J. Dolby, S. J. Fink, and M. Sridharan. Tj watson libraries for analysis (wala). http://wala.sourceforge.net/, visited on 2018.

[23] J. Duncan, L. Phillips, and P. McLeod. *Measuring the Mind: Speed, Control, and Age*. Oxford Scholarship, 2005.

[24] Eclipse Community. Eclipse. http://www.eclipse.org, visited in 2018.

[25] Eclipse Community. Aspectj. https://www.eclipse.org/aspectj/, visited on 2018.

[26] Eclipse Community. Eclipse equinox weaving. https://www.eclipse.org/equinox/weaving/, visited on 2018.

[27] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering (TSE)*, 31(4):312–327, 2005.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 27(2):99–123, 2001.

[29] A. Ferrando, D. Ancona, and V. Mascardi. Decentralizing mas monitoring with decamon. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 239–248. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[30] A. Field. *Discovering statistics using IBM SPSS statistics*. Sage, 2013.

[31] L. Gazzola. Field testing of software applications. In *Proceedings of the International Conference on Software Engineering (ICSE) - Companion*, 2017.

[32] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzè. An exploratory study of field failures. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 67–77, 2017.

[33] G. V. Glass, P. D. Peckham, and J. R. Sanders. Consequences of failure to meet assumptions underlying the fixed effects analyses of variance and covariance. *Review of educational research*, 42(3):237–288, 1972.

[34] K. Havelund, G. Reger, and G. Rosu. Runtime verification past experiences and future projections. *LNCS 10000*, 2018.

[35] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001.

[36] R. Hocke. Sikulix capture and replay tool. `http://sikulix.com`, visited on 2018.

[37] P. Hosek and C. Cadar. VARAN the unbelievable: An efficient n-version execution framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[38] J. A. Hoxmeier and C. D. Cesare. System response time and user satisfaction: An experimental study of browser-based applications. In *Proceedings of the Association of Information Systems Americas Conference*, pages 10–13, 2000.

[39] Intel. Pin: Building customized program analysis tools with dynamic instrumentation. `http://web.stanford.edu/class/cs343/resources/cs343-annot-pin.pdf`, visited on 2018.

[40] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[41] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2012.

[42] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 234–243. IEEE, 2007.

[43] Y. Joshi, G. M. Tchamgoue, and S. Fischmeister. Runtime verification of ltl on lossy traces. In *Proceedings of the Symposium on Applied Computing*, pages 1379–1386. ACM, 2017.

[44] N. Juristo and A. Moreno. *Basics of software engineering experimentation*. Springer Science & Business Media, 2013.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[46] P. R. Killeen and N. A. Weiss. Optimal timing and the Weber function. *Psychological Review*, 94(4):455–468, 1987.

[47] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 179–182. ACM, 2010.

[48] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *ACM Sigplan Notices*, 38(5):141–154, 2003.

[49] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *Acm Sigplan Notices*, 40(6):15–26, 2005.

[50] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[51] Microsoft. Windows 10. `http://www.microsoft.com`, visited in 2018.

[52] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[53] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Software En-*

*gineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 156–165. IEEE, 2005.

[54] F. F. Nah. A study on tolerable waiting time: How long are web users willing to wait? *Behavior and Information Technology*, 23(3):153–163, 2004.

[55] Netflix. Netflix. `http://www.netflix.com`, visited in 2018.

[56] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11:1–11:29, 2011.

[57] L. Nielsen. User interface directions for the web. *Commun. ACM*, 42(1):65–72, Jan. 1999.

[58] G. Norman. Likert scales, levels of measurement and the "laws" of statistics. *Advances in Health Sciences Education*, 15(5):625–632, Dec 2010.

[59] O.Cornejo, D. Briola, D. Micucci, and L. Mariani. Fragmented monitoring. In A. Francalanza and G. J. Pace, editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017.*, volume 254 of *EPTCS*, pages 57–68, 2017.

[60] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing customized program coverage. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2016.

[61] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering*, 24(4):865–904, 2017.

[62] Oracle. Java logging technology. `https://docs.oracle.com/javase/6/docs/technotes/guides/logging/`, visited on 2018.

[63] A. Orso. Monitoring, analysis, and testing of deployed software. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, 2010.

[64] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[65] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, volume 27. ACM, 2002.

[66] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[67] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 277–284. IEEE, 1999.

[68] J. Samieson. Likert scales: how to (ab)use them. *Medical Education*, 38(12):1217–1218, 2004.

[69] T. Schlossnagle. Monitoring in a devops world. *Communications of the ACM*, 61(3):58–61, Mar. 2018.

[70] S. C. Seow. *Designing and engineering time: the psychology of time perception in software*. Addison-Wesley Professional, 2008.

[71] B. Shneiderman. Designing the user interface strategies for effective human - computer interaction. *SIGBIO Newsl.*, 9(1), Mar. 1987.

[72] B. Shneiderman, C. Plaisant, M. Cohen, and S. Jacobs. *Designing the User Interface: Strategies for Effective Human Computer Interaction, fifth ed.* Addison-Wesley, 2009.

[73] Tigris. Argouml. `http://argouml.tigris.org`, visited in 2018.

[74] W. M. Van der Aalst. *Process mining: data science in action*. Springer, 2016.

[75] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

[76] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.