A publication of ISCA*:
International Society for Computers
and Their Applications

INTERNATIONAL JOURNAL OF COMPUTERS AND THEIR APPLICATIONS

TABLE OF CONTENTS

Page

Editor's Note 1

Guest Editorial 2

Defining the Paradigm of a Highly Automated System that Protects Against Human Failures and Terrorist Acts and Application to Aircraft Systems 4
Maximilian M. Etschmaier and Gordon Lee

Multiple Regression Analysis and Learning System for Estimation of Blood Pressure Variation Using Photo-Plethysmograph Signals 12
Michio Yokoyama, Takumi Negishi, Mitsuru Mizunuma, Kazuya Otani; Hidenobu Hanaki, and Kozo Nishimura

Measuring the Impact of Network Performance on Cloud-Based Speech Recognition 19
An Empirical Study of Apple Siri and Google Speech Recognition
Mehdi Assefi, Guangchi Liu, Mike P. Wittie, and Clemente Izurieta

Test Suite Selection in Junit Testing Environment Based on Software Metrics 29
Shadi Banitaan, Kevin Daimi, Mohammed Akour, and Yujun Wang

Assessing Software Defects Using Nano-Patterns Detection 35
Ajay K. Deo, Zadia Codaabux, Kazi Zakia Sultana, and Byron J. Williams

Transforming C Applications with Meta-Programing 57
Songqing Yue and Jeff Gray

Investigating the Relationships between Use Cases Attributes and Source Code Size 69
William Flageol, Mourad Badri, and Linda Badri

*** "International Journal of Computers and Their Applications is abstracted and indexed in INSPEC and Scopus."**

Volume 23, No. 1, March 2016 **ISSN 1076-5204**

International Journal of Computers and Their Applications

A publication of the International Society for Computers and Their Applications

ISCA Headquarters···•···64 White Oak Court, Winona, MN 55987···•···Phone: (507) 458-4517
E-mail: isca@ipass.net • URL: http://www.isca-hq.org

Editor's Note: March 2016

It is my distinct honor, pleasure and privilege to serve as the Editor-in-Chief of the International Journal of Computers and Their Applications (IJCA). I have a special passion for the International Society for Computers and their Applications.

I would like to begin this volume by giving a review of this past year. In 2015 we had 68 articles submitted to the International Journal of Computers and Their Applications. We currently have 16 that are still under review. The ISCA Board just voted to change the publication fees from a per page to a flat fee. For ISCA members it is $400 per article and it is $500 for non-members, and the journal will not be accepting articles that are less than 6 pages. The authors of these papers will be encouraged to submit their papers to ISCA conferences.

Of great interest is the fact that ISCA conference proceedings and this journal will begin to appear in the Search Digital Library at searchdl.org. Hopefully, over the next year we will have this volume and also begin working backwards through previous volumes to get IJCA online.

I look forward to working with everyone in the coming years to maintain and further improve the quality of the journal. I would like to invite you to submit your quality work to the journal for consideration of publication. I also welcome proposals for special issues of the journal. If you have any suggestions to improve the journal, please feel free to contact me.

Frederick C. Harris, Jr.
Computer Science and Engineering
University of Nevada, Reno
Reno, NV 89557, USA
Phone: 775-784-6571
Email: Fred.Harris@cse.unr.edu

This year we have 4 issues planned (March, June, September, and December). We begin with a special issue from the best papers at the ISCA Fall Conference cluster (CAINE, and SEDE). We have a proposal for the best papers from the ISCA Spring Conference cluster (CATA/BICOB) which will appear in the September issue. The other two issues (June and December) are being filled with submitted papers.

I would also like to announce that I begun a search for a few Associate Editors to add to our team. There are a few areas that we would like to strengthen our board with, such as Image Processing. If you would like to be considered, please contact me via email with a cover letter and a copy of your CV.

Frederick C Harris, Jr.
Editor-in-Chief
Email: Fred.Harris@cse.unr.edu

Guest Editorial: Special Issue from ISCA Fall-2015 Conferences

This Special Issue of IJCA is a collection of five refereed papers selected from the following ISCA conferences (co-located at the Hilton San Diego, Harbor Island, San Diego, California, USA, October 12-14, 2015):

- CAINE 2015: 28th International Conference on Computer Applications in Industry and Engineering
- SEDE 2015: 24th International Conference on Software Engineering and Data Engineering

Each paper submitted to the conferences was reviewed by at least two members of the International Program Committee, as well as by additional reviewers, judging the originality, technical contribution, significance and quality of presentation. After the conferences, nine best papers were recommended by the Program Committee members to be considered for publication in this Special Issue of IJCA. The authors were invited to submit a revised version of their papers. After extensive revisions and a second round of review, seven papers were accepted for publication in this issue of the journal.

The papers in this special issue cover a wide range of research interests in the community of computers and applications. The topics and main contributions of the papers are briefly summarized below.

Maximilian M. Etschmaier and Gordon Lee of San Diego State University have a paper entitled “Defining the Paradigm of a Highly Automated System That Protects Against Human Failures and Terrorist Acts, And Application to Aircraft Systems.” In it they present a paradigm and use a case study to show its effectiveness, as well as discuss several recent crashes.

Michio Yokoyama, Takumi Negishi, Mitsuru Mizunuma, all of Yamagata University in Japan and Kazuya Otani, Hidenobu Hanaki, and Kozo Nishimura, of TOKAI RIKA Co., Ltd, Japan have a paper entitled “Multiple Regression Analysis and Learning System for Estimation of Blood Pressure Variation Using Photo-Plethysmograph Signals.” In this paper they propose a blood pressure estimation system using photo-plethysmograph signals. Their experiments show the estimates are within 10 mmHg when compared with measures from a traditional cuff.

Mehdi Assefi, Guangchi Liu, Mike P. Wittie, and Clemente Izurieta from Montana State University have a paper entitled “Measuring the Impact of Network Performance on Cloud-Based Speech Recognition: An Empirical Study of Apple Siri and Google Speech Recognition.” They measured transcription delay and accuracy with varying packet loss and presented their results which are statistically significant.

Shadi Banitaan, Kevin Daimi, Mohammed Akour, and Yujun Wang have a paper entitled “Test Suite Selection in JUnit Testing Environment based on Software Metrics.” The authors are from the University of Detroit, USA and Yarmouk University, Jordan. In this paper they focus on programs written in Java and tested with Junit. Their results show that you can significantly reduce the number of test cases needed to detect most of the errors.

Ajay K. Deo, Zadia Codabux, Kazizakia Sultana, Byron J. Williams from Mississippi State University, USA have a paper entitled “Assessing Software Defects Using Nano-Patterns Detection.” They presented a study that that evaluated software defects using nano-patterns. They studied three open source systems from Apache and found that certain nano-patterns are more defect prone than others.

Songqing Yue of the University of Central Missouri, and Jeff Gray of the University of Alabama, USA have a paper entitled “Transforming C Applications with Meta-programming.” In this paper they describe how to bring computational reflection to the C programming language through a Meta Object Protocol. They present a Domain-Specific Language called SPOT to allow developers to specify direct manipulation of C Programs.

William Flageol, Mourad Badri and Linda Badri of the University of Quebec, Trois-Rivières, Quebec, Canada have a paper entitled “Investigating the Relationships between Use Cases Attributes and Source Code Size.” In this paper they present a study that investigates the relationship between use case attributes and source code size. An empirical study of several open source Java projects is presented and the results provide evidence that support their hypothesis.

As guest editors we would like to express our genuine appreciation for the encouragement and support from the ISCA. We also owe many thanks to the authors and program committees of the conferences from which these papers were selected.

We hope you enjoy this special issue of the IJCA and we look forward to seeing you at a future ISCA conference. More information about the ISCA society can be found at http://www.isca-hq.org.

Guest Editors:

Gongzhu Hu, Central Michigan University, USA, CAINE 2015 Conference Chair
Takaaki Goto, Ryutsu Keizai University, Japan, CAINE 2015 Program Chair
Frederick C Harris, Jr., University of Nevada, Reno, USA, SEDE 2015 Conference Chair
Yan Shi, University of Wisconsin-Platteville, USA, SEDE 2015 Program Co-Chair
Dr. Wenying Feng, Trent University, CANADA, SEDE 2015 Program Co-Chair

March 2016

Defining the Paradigm of a Highly Automated System that Protects Against Human Failures and Terrorist Acts and Application to Aircraft Systems

Maximilian M. Etschmaier[1] and Gordon Lee[2]
San Diego State University, San Diego, CA 92182 USA

Abstract

There are many systems in which the human plays a role in the overall performance of the system. In highly automated systems, the role of the human is to support automatic processes and procedures to the extent that human input may only be required to provide high level goals, command unforeseen changes in the course of the system, or respond to exceptional system behavior. In general, it is difficult for an automated system to determine the state of the human's intent, except through the impact the control input has on the state of the system. An effective and efficient protection focuses on the potential impact and mitigates potential critical failures before they become a reality. This is the approach taken in this paper; it is a direct application of a total systems approach. Defining the paradigm of a "highly automated system," we will show how an effective system architecture can be used to develop protection against failures of the human element. The architecture obviates many of the complex, expensive and intrusive measures that have resulted from piecemeal approaches that are currently in use or under consideration. A case study of how an airliner can be protected against human failure in the cockpit, including deliberate, destructive acts by cockpit personnel or terrorist humans in the cockpit is provided to illustrate the approach. The effectiveness of the approach is discussed in the context of several recent crashes.

Key Words: Purposeful systems, human-machine symbiosis, system security, aircraft safety and security, human factors, human failures.

1 Introduction

The crash of an airliner on a scheduled flight into the French Alps on March 24, 2015 has refocused attention of the aviation community as well as the media on the vulnerability of aircraft to failures of the human element. The preliminary understanding of the crash is that the first officer, due to a psychiatric condition, committed suicide and purposely steered the aircraft into rising terrain. Even if this scenario ultimately proves to be incorrect, the question of how to protect aircraft safety vis-à-vis a member of the cockpit crew intent on taking his life by destroying the aircraft is a valid one that currently does not have a satisfactory solution. We will examine this issue in the context of the whole spectrum of failures of the human element in the cockpit of a modern airliner. These failures may be voluntary or involuntary. They may include incapacitation of the cockpit crew by illness, by environmental conditions created by a failure of the aircraft system, or by external force, such as a terrorist act. We will also examine situations where an aircraft malfunction puts the human in a position of control that he/she is essentially not equipped to assume.

The level of automation of modern airliners is such that the term "highly automated system" might appear quite appropriate. The same is true for complex systems in a wide variety of other domains, such as surface and marine vehicles, nuclear and thermal power plants, furnaces, reactors, and other equipment in the processing industry, discrete manufacturing systems, as well as information processing networks. The purpose of automation in these systems goes beyond a machine substituting for human labor, both physical and mental, and reaches a point where it is the interaction of the human element and the machine that assures that sustained system operation will realize the system purpose. In such a system, then, the human element constitutes an integral part of a purposeful system [10].

We present a definition of a highly automated system and identify the attributes required of such a system. We propose an architecture for control of highly automated systems that includes the human element as an integral part and thus makes it possible to deal with the failures of the human element through a holistic systems analysis. The architecture recognizes that it is all but impossible to directly monitor the condition of the human element. Instead, since in a highly automated system the human input is in the form of high-level control inputs, mostly as strategy, it is possible to monitor and diagnose its impact. For a civilian airliner, we will show that diagnosis can be limited to evaluating the system state relative to a small number of "envelopes" of feasible states. The architecture can be implemented largely by relatively minor repurposing of existing automation systems.

A prerequisite for the architecture to produce the desired level of protection is that the existing automation systems are or can be made to be internally consistent and actually perform as expected. Complications may arise when failures of the

[1] College of Sciences. Email: metschmaier@mail.sdsu.edu.
[2] Dept. of Elec & Comp Engr. Email: glee@mail.sdsu.edu

human element coincide with failures of the "physical" system since this would leave the system in a state without any trustworthy authority to fall back on. In this case it may be necessary to seek recourse with an outside authority or some form of "deus ex machina," to mitigate the consequences of an otherwise catastrophic outcome. An example of how an incomplete realization of the presented concept of a highly automated system can lead to a catastrophic outcome is provided in the discussion of the AF447 accident in Section 5.

2 Highly Automated Systems

In a highly automated system, the role of the human element is largely restricted to providing higher level (strategic) control inputs and supervising the operation of the machine. In its role of controller, the human element is heavily supported by functions provided by the "physical" system. The role may be important such as defining the overall goals and performance objectives or supervising the system when the state of the system veers towards a critical failure. The research literature is rich in the analysis and design of automated systems. Much of the earlier work focused on manufacturing systems (in [5], for example, Choi and Xirouchakis give an overview of flexible manufacturing and the issue of production planning) while current research focuses on intelligent vehicles (automobiles or aircraft). In [21, 23], for example, Jamson, Merat, and their colleagues look at the various tasks that a driver must execute and how highly automated vehicles could remove some of the "mundane" tasks of keeping the vehicle on the road and allow the driver to focus on in-vehicle tasks, such as entertainment or checking one's smart phone. While somewhat controversial, these studies do provide an opportunity for researchers of automated systems to investigate the boundary between the human and the system, i.e., how should the tasks associated with the control of a system be divided between human and machine?

In [30], Vanholme et. al., develop a system architecture and associated hardware implementation for a platform that allows the human to select the cooperation between the human and the vehicle. Some high level tasks such as environmental mapping, planning a route to a final destination, and traffic conditions can be generated through multiple sensors and databases. While interesting, the human must still decide on the cooperation level between the driver and the vehicle.

This division of cooperation has been recognized by many. In [13], for example, Frau develops a user-interface that may aid researchers in capturing data on human-machine interactions which could be used to define the boundaries of this cooperation. Gold et. al. [16] investigate at which point the driver must take back control of a highly automated vehicle. Issues such as reaction time versus risk level (such as potential collision) were studied and the authors found that shorter take-over requests (TOR) led to poorer vehicle performance. Inagaki and others [19, 20] develop a probability theoretic model for trading authority when the situational awareness differs between the human and the machine. Grote and others [18] provide an excellent overview on the state of human-machine cooperation and the need for more research in understanding the boundary between this cooperation. Geyer and others in [15] take a different approach in addressing the cooperation boundary by looking at maneuvering commands rather than stabilization commands for vehicle control. That is, considering a higher level set of goals that the vehicle should target may provide a better solution to the human-machine cooperation. They use a conduct-by-wire principle to formulate the vehicle guidance. This architecture is the basis for this paper; however, the principles used here are based on a systems approach to prevent critical failures rather than a guidance strategy.

Other approaches address the cooperation between the human and the machine from the perspective of Ergonomics. The focus there is on the role of the human in relationship to the machine and the requirements for the design of the machine to enable this role. The design issues mostly involve the mode in which the machine communicates with the human, i.e., the Gestalt of the user interface. Goodrich and others [17] employ the synergy between a horse and its rider as a metaphor for human-machine symbiosis (the "H-metaphor") in controlling an aircraft. The centerpiece of the architecture is haptic feedback through which a human can interact with the aircraft directly and in an intuitive and often subconscious way. This enables the human to be in control even if the attention is focused elsewhere. Similarly, the aircraft will recognize when the human has stopped providing expected control inputs and suggest, and eventually initiate, an emergency landing. The result is a harmonious blending of the human controller and the aircraft into a mutually supportive arrangement. The human remains in a position to serve as the ultimate arbiter of control ambiguity. Abbink and others [1] describe experiments with limited symmetry in control authority between the human and the machine based on a haptic interface. They find that much work remains to be done to eliminate situations where the human and the machine wind up fighting each other. Pizziol and others [27-28] discuss more general situations where the human and the machine are in conflict, either by choice of the human, which would include failures of the human element, or by failures of the machine. In [27] they focus on knowledge conflicts, which are situations where the goals of the human and the machine are in agreement, but the information on which to base decisions are in conflict. Both use a model based on Petri-nets to analyze and resolve the arising conflicts.

Stensson and Jansson [29] point to the remarkable work of Bainbridge [4] who identified, some thirty years ago, the "ironies of automation" that higher levels of automation required increased cognitive and manual skills of the human tending to such systems. They use Kant's theory of the categorical imperative to identify limits of automation. They point out the difference between automation and autonomy and show that, consistent with Bainbridge, the notion of autonomy would require properties that machines will not possess in the foreseeable future.

In this paper, the human-machine system is viewed as one integral whole that is designed from the very beginning to fully integrate the human element into the system. We follow the definition of a purposeful system of [10] to base the system design on the functions the system is expected to deliver. Part of the system design are provisions that assure the prevention of losses of functions that would have critical, that is unacceptable, consequences. Specifically, the paper deals with critical failures of the human element in the system and treats them very much like failures of physical elements as losses of functions.

3 An Architecture for Highly Automated Systems

Even though the notion of a “highly automated system” has been widely used to describe systems with a high degree of automation, there is little clarity in what exactly differentiates highly automated from plain automated systems. Clearly, there cannot be some arbitrary point on a continuum of automation beyond which a system would become “highly automated.” Instead, the distinction should be based on identifiable structural properties.

Let these properties be defined as shown in Figures 1 and 2. The system consists of a human element, a control computer, and a machine, all integral parts of the system. The human element originates the strategic direction of the system and can also provide direct control. It interacts with the machine exclusively through the control computer. Direct input of control instructions to the machine is excluded. Both, the human element and the control computer are monitoring the state of the system, including each other’s state and control input, and the environment.

Figure 1 shows how the control authority over the operational system element is shared between the human element and the sum of system elements that can be summarized as Control Computer. The human element may provide input at two levels: at the operational level the controls can be received and executed by the operational system without interpretation; strategic input requires operationalization by the control computer in order to be executable. Balance of the control authority requires that both forms are monitored and possibly censored (overridden) by the control computer. To enable the control computer to perform its censor function, it must not be possible for the human

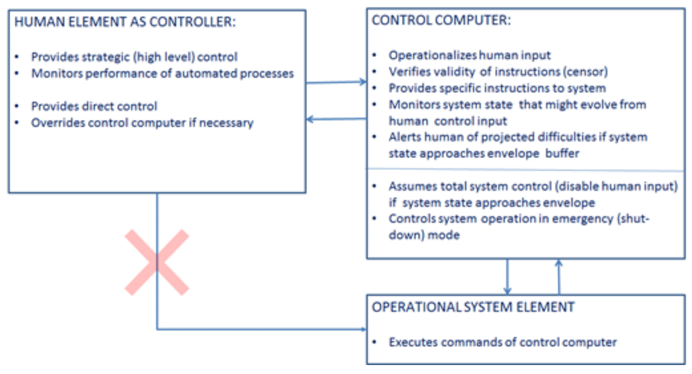

HUMAN ELEMENT AS CONTROLLER:
• Provides strategic (high level) control
• Monitors performance of automated processes
• Provides direct control
• Overrides control computer if necessary
CONTROL COMPUTER:
• Operationalizes human input
• Verifies validity of instructions (censor)
• Provides specific instructions to system
• Monitors system state that might evolve from human control input
• Alerts human of projected difficulties if system state approaches envelope buffer
• Assumes total system control (disable human input) if system state approaches envelope
• Controls system operation in emergency (shut-down) mode
OPERATIONAL SYSTEM ELEMENT
• Executes commands of control computer

Figure 1: Shared authority in a highly automated system

element to provide input to the operational system element without going through the computer. In a symmetrical manner, the performance of the control computer is monitored, and possibly overridden, by the human element. Figure 2 provides the architecture and operational perspective on the arrangement.

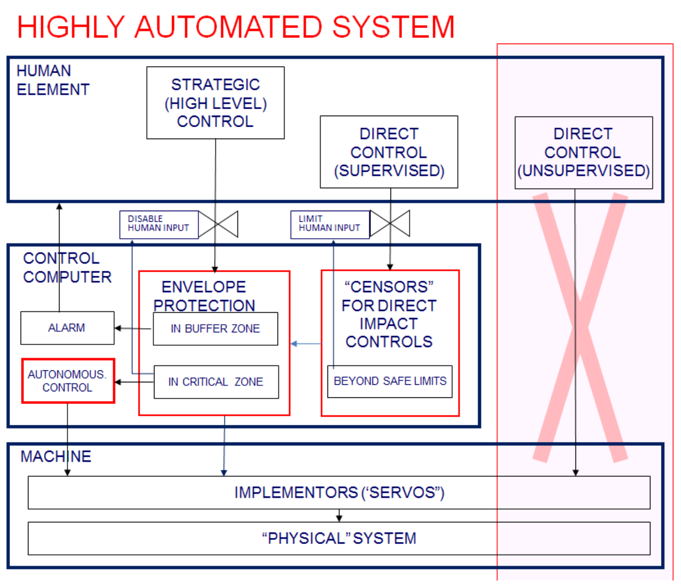


Figure 2: Operational definition of a highly automated system

Our architecture of a highly automated system is similar to [15]. It differs from [15], however, because it includes a certain degree of autonomy for the control computer by providing for the possibility of the control computer to take supreme system authority away from the human element. However, this possibility is reserved to situations where the human control input would inescapably lead to a catastrophic consequence. And the authority of the control computer is limited to what amounts to a safe shut down - and in extreme situations mitigating the consequences of a catastrophic outcome (i.e., saving human lives). Thus the architecture in essence preserves the hierarchical order of the system. In order to exercise any level of control authority, the control computer needs to possess real time data about its state and a model through which it can understand the data, identify and diagnose states, and project a feasible trajectory that avoids entering unacceptable states.

Situations where supreme control authority is taken away from the human element are determined through a functional failure analysis that is part of the functional system design process defined in [10]. Opposite to the traditional failure modes and effects analysis, this process starts with the identification of system functions and determines the potential consequences of a loss of the function. For seizing authority from the human element only functions, the loss of which could lead to catastrophic consequences, are of relevance. Normally, the analysis then moves upstream to determine cause and mode (mechanism) of each failure to determine how to keep the failure from happening. However, since in the present situation, the failure is ultimately caused by the human, such analysis would involve monitoring and controlling the human element. As stated by Bainbridge [27], "in human-machine systems the human agent is hardly controllable and no "model" of the human's decision processes is available." The only way to develop measures to prevent critical human failures is to analyze the way in which failures unfold. This requires a closer examination of the role of the human element in the system.

As shown in Figure 2, the control authority of the human element is exercised in two forms: (1) Strategic (high level) control, which provides a plan for the future operation of the system and impacts the system state with a longer horizon; and (2) Direct but supervised control, with essentially immediate impact. If the state of the system at any time is defined in some state space, future system operation appears as a trajectory in that state space. This is the projection of the consequences of the strategic control input provided by the human. A failure can be defined as occurring when the system state moves beyond the envelope that defines the set of feasible states.

A strategy to prevent critical failures caused by the human through the input of strategic control can be formulated as follows: partition the state space as shown in Figure 3 by defining three envelopes F, C, and B, and designate the space outside F as the failure state, the state between F and C as critical state, and the state between B and C as buffer state. The state inside B is the normal state within which the human element enjoys unchallenged control authority. If the system state is within the buffer state, the control computer will notify the human element that danger is imminent and corrective action is necessary. If the system is in the critical state immediate corrective action is required to keep it from entering the failure state. The human element has failed to heed the warning and cannot be trusted to take proper action. Therefore, passing envelope F requires the control computer to assume control authority. This transition of authority needs to be unequivocal and without possibility of human intervention. The system being in the failure state does not mean that it has actually failed. Rather, there is no possibility to prevent a critical failure. Any corrective action in that state is futile.

In a highly automated system, the role of the system operator

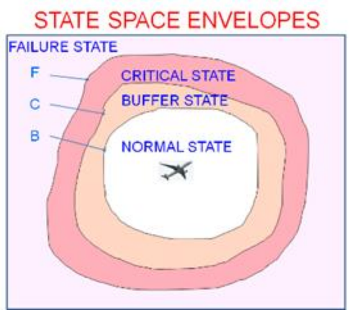


Figure 3: State space envelopes

(human controller) is supported by automatic processes and procedures to the extent that human input is required only to command unforeseen changes in the course of the system and to respond to exceptional system behavior. A failure of the human element in such a system may not be noticed immediately. In fact, a failure may go on and progress only to be noticed when a threat to safety or security is imminent. At that point, the existing system automation may well be capable of safely and securely controlling the system. However, this requires that it is properly configured and prepared for such a situation. Since the system is designed to be under the authority of the human element, controlling the system when the human element has failed, by necessity, needs to occur without involvement of the human element (i.e., overriding any human input) at least until the human authority can be re-established. In general it is difficult for an automated system to determine the state of the human element, except through the impact the control input has on the state of the system. Since it is the impact that is of concern to regaining and assuring control over the system, any impact may be the result of a variety of control inputs, and protection will not involve the human element anyway, it is not useful to spend any effort to determine the nature of the human failure. Instead, an effective and efficient protection focuses on the potential impact and mitigating it before it can take its course.

Strategies for preventing critical failures through human input of direct control can essentially be formulated along the same lines. However, the time of transition from the normal state to the failure state, i.e., the reaction of the system to the control input, is virtually immediate. Any direct control input is one of many parameters that, together and in combination with the state of the current system and the environment, determine the future state of the system. Whereas a strategic control input is in the form of one unique plan, direct control input is generally in the form of setting a control parameter with real or integer value. The control computer can use a comprehensive (simulation) model of the system in the environment to create envelopes in the control parameter space analogous to what is shown in Figure 3. Depending on the position in the control parameter space, the control computer will restrict the range of the control input of the human element. Any restriction may be preceded or accompanied by a warning to the human element. Since direct control inputs create emerging long-term trajectories, compliance of those with the envelopes for strategic controls also needs to be monitored.

The control architecture described for human input of direct control differs from what is widely implemented, for example in transportation vehicles as "fly – or drive by wire," in two ways: (1) it covers the entire system through one architecture; and (2) it does not allow for the human element to override the control computer after it has taken control.

It is that difference that makes it possible to prevent critical consequence from the whole range of failures of the human element.

Current approaches focus on protection against failures caused by involuntary human actions, like errors, omissions, misperceptions, and misunderstandings. They do not protect against voluntary or willful actions, or failures caused by outside force, like sabotage or terrorists gaining access to controls. Such actions are commonly not considered in the design of control systems. In the following section, we will show how the concept of a highly automated system can be realized in an aircraft. Examination of a number of aircraft accidents will demonstrate that failures of the human element in the control of an airliner can be all but eliminated.

4 An Architecture for Protecting an Airliner Against Critical Human Failures, including Terrorist Acts

In a highly automated airliner, strategic control input can affect the following functions: the horizontal flight path, the vertical flight profile (terrain clearance), and movement relative to other traffic (collision avoidance) or prohibited sections of airspace. Direct control can affect the following functions: aircraft attitude, configuration, power settings, and the landing process. Protection against failures of the human element in exercising direct control are widely implemented in modern fly-by-wire aircraft, although only selected aircraft manufacturers have opted to grant the flight computer the absolute authority required by highly automated systems. An example of how this protection can be implemented in controlling the operation of a landing gear is included in [9]. The paper defines "censors" which limit the human input to a safe range determined by the state of the aircraft and the environment.

Failures of the human element that can occur in strategic control are summarized in Table 1.

Table 1: Critical failure analysis

PROTECTION AGAINST CRITICAL FAILURES OF STRATEGIC HUMAN INPUT

FUNCTION	FAILURE	PROTECTION MEASURE
FLIGHT PATH MANAGEMENT	APPROACHING SPACE FROM WHICH NO SUITABLE LANDING SITE CAN BE REACHED WITH AVAILBLE FUEL	DISABLE HUMAN INPUT AFTER WARNING FIND SUITABLE AIRPORT DIRECT FLIGHT TO IT, AUTOLAND
TERRAIN CLEARANCE	APPROACHING SPACE FROM WHICH IMPACT WITH TERRAIN CANNOT BE AVOIDED	DISABLE HUMAN INPUT AFTER WARNING DIRECT AIRCRAFT TO SAFE POSITION TEST HUMAN REPONSE. IF NONE, FIND SUITABLE AIRPORT, DIRECT FLIGHT TO IT, AUTOLAND
TRAFFIC AVOIDANCE	APPROACHING SPACE OCCUPIED BY, OR RESERVED FOR OTHER AIRCRAFT, OR PROHIBITED	DISABLE HUMAN INPUT AFTER WARNING DIRECT AIRCRAFT TO SAFE POSITION TEST HUMAN REPONSE. IF NONE, FIND SUITABLE AIRPORT DIRECT FLIGHT TO IT, AUTOLAND

An examination of the failures shows that every one of them could involve incapacitation of the human caused by preceding failures of the aircraft (e.g., failure of the aircraft skin leads to catastrophic decompression, or smoke or noxious gases entering the cabin), suicide attempts by the pilot, or acts of terrorism. As mentioned before, it may not be possible to identify the ultimate cause. Nor may it be possible to use knowledge of it to protect against a catastrophic failure. Instead, the control computer can monitor the aircraft state relative to the envelopes defined in Figure 3 and initiate the chain of protective measures identified in Table 1. As shown in Figure 2, this monitoring occurs also for supervised direct control. Unsupervised direct control must be excluded because it would make it impossible to protect against suicidal acts by the pilot or against terrorists seizing control.

The control computer contains an on-line model to monitor and control aircraft sustainment (including fuel level) such as outlined in [8, 10], a model of the aircraft flight state that includes access to up-to-date navigation charts and identifies usable airports and excluded airspace. Also, it must be able to direct a completely automatic landing. All these features are readily available in most modern airliners or could be developed within the state of the art. They would obviate many of the measures currently in place or being discussed to prevent terrorist and suicidal acts.

In the following section, we will examine how this architecture would have performed in some of the recent aircraft disasters and would have provided protection against terrorist acts.

5 Examination of Failures of the Human Element in Airliners

The Germanwings Flight 9525 mentioned at the beginning of this paper was on a controlled flight into terrain [11]. It would have been possible for the flight computer to determine that the flight trajectory was leading to a state from which collision with terrain was unavoidable, warn the pilot, and, when he failed to respond, seize control from him. A system to warn of vertical proximity is on all modern airliners. The envelope protection required of a highly automated aircraft would include full awareness of surrounding terrain that can be obtained from navigational charts. There would be no need for the control computer to know the mental state of the first officer, or that he was alone in the cockpit, and that the captain, who might have been able to avert the crash, was unable to overcome the locked, armored cockpit door.

Accident investigations focus on identifying the cause of the accident, and on possible ways to prevent similar accidents from happening in the future. A combination of two factors was identified as the cause: the mental state of the first officer, and the inability of the captain to overcome the locked cockpit door. It is tempting to argue that if either one of those causes could be ruled out, a repeat of the same accident scenario could be prevented. Unfortunately, screening persons whose mental state predisposes them to violent suicidal acts is well beyond the state of the art. A Swedish study has found that the percentage of people with depression who commit violent crimes is extremely low — just 3.7 percent of men and 0.5 percent of women with depression commit such crimes, compared with 1.2 percent of men and 0.2 percent of women in the general population. Besides screening would violate rights to privacy [11, 22].

Ironically, armored doors that are impenetrable without consent from within the cockpit were mandated as a measure to protect against a repeat of terrorist acts of September 11, 2001 where terrorists overpowered the pilots. Besides, there is no assurance that a second person in the cockpit could have prevented a crash.

The terrorist attacks of September 11, 2001 involved controlled flights into terrain [7], i.e., the towers of the World Trade Center and the Pentagon. It could not have happened had the aircraft been configured in the architecture of a highly automated system, no matter how the terrorists gained control of the cockpit. As the aircraft state entered the buffer state, a warning would have been broadcast; and no matter what the terrorists would have done, upon entering the critical state, the computer would have seized authority and brought the aircraft to a safe emergency landing. In fact, it is unlikely that a group of sophisticated terrorists would even attempt an action like the World Trade Center attacks if they knew that aircraft are configured as highly automated systems.

Flight MH370 stopped communicating with air traffic control, was observed to make a sharp change in course and altitude to fly over the open ocean and outside reach of air traffic control [2, 26]. It is presumed to have crashed after the fuel was exhausted. The flight path has only been partially reconstructed, and despite intensive search, a wreckage was never found. There is speculation that the cause was a deliberate act by the pilot. Alternately, the pilots might have been incapacitated by some mechanical failure which also would have disabled communication, but not affected the control computer or the operability of the control surfaces. The aircraft continued flying on autopilot until the fuel was exhausted.

Efforts are now underway to assure that any aircraft can be tracked anywhere [6]. However, following the path of the aircraft would not have prevented an eventual crash. In fact, several aircraft with disabled pilots, in the past, were followed by aircraft that could only watch them crash. One widely observed incident occurred on October 25, 1999 with a Learjet carrying a highly ranked golf pro. When radio contact with the flight was lost north of Gainesville, Florida, the airplane was intercepted by several military aircraft. The military pilots in a close position found the windshields of the Learjet to be frosted or covered with condensation and could not see into the cabin or establish contact with the crew. When the Learjet reached Aberdeen South Dakota, they observed the airplane entering a spiral to the ground. All occupants on board the airplane were killed, and the airplane was destroyed. The National Transportation Safety Board determined the probable cause of this accident as incapacitation of the flight crewmembers as a result of their failure to receive supplemental oxygen following a loss of cabin pressurization, for undetermined reasons [25]. In this flight, like in flight MH370, a highly automated aircraft would have recognized that it is leaving the normal state of the flight path, and upon entering the critical state of imminent fuel exhaustion, would have seized control and brought the aircraft to a safe emergency landing. This would have been done independent of the state of the pilots.

The crash of Air France Flight 447 was initiated by the failure of all three airspeed sensors [31]. The airspeed sensors are Pitot tubes that provide information that is vital for both the automated system (flight computer) and the cockpit crew to control the aircraft. When the flight computer recognized that the airspeed information was unreliable, it correctly determined that it could not control the flight through the autopilot. Since it had no procedures available to determine at least an approximation of the airspeed through other means, it was left without a critical piece of information to provide support and censor functions on direct pilot input. It was programmed to stop providing the censor function in such cases. This left the pilot, who had much less situational awareness, to deal with the situation. The aircraft entered a stall from which it did not recover.

Ironically, the situation was made more difficult because, when the airspeed signal did become available again, the computer intermittently tried to regain control, fighting and confusing the pilot. A highly automated aircraft in such a situation would search for alternative sources of information for the airspeed instead of passing responsibility to the pilot. Interestingly, an alternate method for determining airspeed with sufficient accuracy is now included in some aircraft [14].

The examples show that hastily configured measures to protect against failures of the human element and terrorist attacks may impose exorbitant cost and inconvenience on an industry without achieving the intended goal, or possibly even opening other vulnerabilities.

According to FAA projections, the installation of the fortified cockpit doors cost the industry up to $112.7 million plus 27.5 million over ten years for increased fuel consumption due to the higher weight of the doors [12]. Reference [24] tells about considerable operational problems caused by the fortified doors and the operational requirements imposed on them, as well as serious impact on aircraft safety, and significantly higher cost figures.

The cost of the installation of global tracking capability on all current long-haul aircraft in the US transoceanic fleet is estimated at $35 million, in addition to considerable ongoing charges for the use of communication facilities [3]. While the global tracking capability may constitute an important step in efforts of security agencies, it will not be effective in preventing the type of incident that motivated its installation.

6 Conclusions

Successful automation requires a system structure that is internally consistent and properly implemented. Increasing the level of automation of a system will only yield desired system performance improvement if it is designed following a holistic systems approach. We have defined the concept of a highly automated system through a control architecture that provides a limited symmetry of control authority between the human element and a control computer. While the architecture strengthens the role of the human element as the dominant authority, it also empowers the computer to seize all control authority from the human element when it determines that human control instructions inescapably will lead to a critical failure. However, the role of the computer in such situations is strictly limited to charting a course towards a safe shutdown. This will avoid the much-feared situation of a runaway robot that is superior in power and resilience to the human element and cannot be stopped from harming humans and damaging property.

Automation following the architecture of a highly automated system will yield a step change in performance. In particular, it enables any system to protect itself against any critical failures of the human element, irrespective of whether these failures are the result of negligence, errors in judgement, accidental impairment, or willful acts of destruction. This provides direct protection against terrorist acts which depend on terrorists gaining control over the system. Such protection would be more effective and less disruptive and expensive than indirect methods currently deployed or considered. The case study included shows how the architecture applies to aircraft safety. The architecture is entirely compatible with state of the art technology. Analysis of selected examples of human failures and terrorist acts shows that it would have provided full protection in every case, obviating the need for indirect protection measures called for by the public that are extremely expensive and disruptive without being able to guarantee protection.

References

[1] David A. Abbink, Mark Mulder, and Erwin R. Boer, "Haptic Shared Control: Smoothly Shifting Control Authority," Cognitive Technology Workshop, 14:19-28, DOI 10.1007/s10111-011-0192-5, 2012.

[2] Australian Transport Safety Bureau, MH370 – Flight Path Analysis Update, http://www.atsb.gov.au/media/5163181/AE-2014-054_MH370%20-FlightPath AnalysisUpdate.pdf, October 8, 2014.

[3] Aviation Safety, Proposals to Enhance Aircraft Tracking and Flight Data Recovery May Aid Accident Investigation, but Challenges Remain, United States Government Accountability Office, GAO-15-443, April 2015.

[4] L. Bainbridge, "Ironies of Automation," *Automatica*, 19(6):775-779, 1983.

[5] Yong-Chan Choi, and Paul Xirouchakis, "A Production Planning in Highly Automated Manufacturing System Considering Multiple Process Plans with Different Energy Requirements," *International Journal of Advanced Manufacturing Technology*, 70:853-867, 2014.

[6] John Croft, "Old and New Issues Inundate IATA Safety Agenda," Aviation Week, http://aviationweek.com/commercial-aviation/old-and-new-issues-inundate-iata-safety-agenda, May 26, 2015.

[7] Barbara Elias, Ed., Government Releases Detailed Information on 9/11 Crashes, National Security Archive, http://nsarchive.gwu.edu/NSAEBB/NSAEBB196/index.htm, August 11, 2006.

[8] Maximilian M. Etschmaier, Stuart Rubin, and Gordon Lee, "On the Use of SOMPA Core Modeling for Systems Design: A Case Study," IEEE World Automation Congress, Kona, Hawaii, August 2014.

[9] Maximilian M. Etschmaier, Stuart Rubin, and Gordon Lee, "A System of Systems Approach to the Design of a Landing Gear System: A Case Study," 27th International Conference on Computer Applications in Industry and Engineering, New Orleans, October 13-15, 2014.

[10] Maximilian M. Etschmaier, "Purposeful Systems: A Conceptual Framework for System Design, Analysis, and Operations," *International Journal for Computers and Applications*, 22(2):1-13, June 2015.

[11] Seena Wolf Fazel, Zheng Achim Chang, Henrik Larrson, Guy M. Goodwin, and Paul Lichtenstein, "Depression and Violence: A Swedish Population Study," *Lancet Psychiatry*, 2:224-32, 2015.

[12] Federal Register, http://www.gpo.gov/fdsys/pkg/FR-2002-01-15/html/02-965.htm), 67(10), Tuesday, January 15, 2002.

[13] Giuseppe Frau, "User-Interface Design for Highly Automated Systems a Structured Approach," *Proceedings of the ACM 3rd International Conference on Application and Theory of Automation in Command and Control Systems*, pp. 148-151, 2013.

[14] Fred George, Aviation Week Evaluates Boeing 787, http://aviationweek.com/commercial-aviation/aviation-week-evaluates-boeing-787, December 10, 2012.

[15] Sebastian Geyer, Stephan Hakuli, Hermann Winner, Benjamin Franz, and Michaela Kauer, "Development of a Cooperative System Behavior for a Highly Automated Vehicle Guidance Concept based on the Conduct-by-Wire Principle," *Proceedings of the Fourth IEEE Intelligent Vehicles Symposium*, pp. 411-416, 2011.

[16] Christian Gold, Daniel Damböck, Lutz Lorenz, and Klaus Bengler, "Take Over! How Long Does It Take to Get the Driver Back into the Loop?" *Proceedings of the Human Factors and Ergonomics Society 57th Annual Meeting*, pp. 1938-1942, 2013.

[17] Kenneth H. Goodrich, Paul C. Schutte, Frank O. Flemisch, and Ralph A. W. Williams, "Application of the H-Mode: A Design and Interaction Concept for Highly Automated Vehicles, to Aircraft." *Proceedings of the AIAA/IEEE Digital Avionics Systems Conference: Network-Center Environment - The Impact on Avionics and Systems*, pp. 4A3-1-13, 2006.

[18] Gudela Grote, Johannes Weyer, and Neville A. Stanton, "Beyond Human-Centered Automation – Concepts for Human–Machine Interaction in Multi-Layered Networks," *Ergonomics*, 57(3):289-294, DOI: 10.1080/00140139.2014.890748, 2014.

[19] T. Inagaki, Special Issue on Human-Automation Cogency, Cognitive Technology Workshop, 14:1-2 DOI 10.1007/s10111-011-0197-0, 2012.

[20] Toshiyuki Inagaki and Thomas B. Sheridan, "Authority and Responsibility in Human–Machine Systems: Probability Theoretic Validation of Machine-Initiated Trading of Authority," Cognitive Technology Workshop, 14:29–37, DOI 10.1007/s10111-011-0193-4, 2012.

[21] A. Hamish Jamson, Natasha Merat, Oliver M. J. Carsten, and Frank C. H. Lai, "Behavioral Changes in Drivers Experiencing Highly-Automated Vehicle Control in Varying Traffic Conditions," *Transportation Research:*

Part C: Emerging Technologies, Elsevier Publishers, 30:116-125, 2013.

[22] Tanya Lewis, "Germanwings Crash: Mental Illness Alone Does Not Explain Co-Pilots Behavior," Life Science, http://www.livescience.com/50291-germanwings-copilot-mental-illness.html, March 2015.

[23] Natasha Merta, A. Hamish Jamson, Frank C. H. Lai, Michael Daly, and Oliver M. J. Carsten, "Transition to Manual: Driver Behavior when Resuming Control from a Highly Automated Vehicle," *Journal on Transportation Research: Part F: Traffic Psychology and Behavior,* Elsevier Publishers, 27:274-282, 2014.

[24] New Doors Causing Cockpit Problems, LA Times, http://articles.latimes.com/print/2003/dec/14/nation/na-doors14, December 14, 2003.

[25] NTSB Aircraft Accident Brief DCA00MA005, November 28, 2000.

[26] Office of the Chief Inspector of Air Accidents, Ministry of Transport, Malaysia, MH370 Preliminary Report, http://www.dca.gov.my/MH370/Preliminary%20Report.pdf. April 9, 2014.

[27] Sergio Pizziol, Catherine Tessier, and Frédéric Dehais, "What the Heck Is It Doing? Better Understanding Human-Machine Conflicts Through Models, CEUR," *Proceedings of the 1st Workshop on Rights and Duties of Autonomous Agents (RDA2)*, Montpellier, 885:44-49, August 28, 2012.

[28] Sergio Pizziol, Catherine Tessier, and Frédéric Dehais, "Petri Net-Based Modelling of Human–Automation Conflicts in Aviation," Ergonomics, DOI: 10.1080/00140139.2013.877597, 2014.

[29] Patrik Stensson and Anders Jansson, "Autonomous Technology – Sources of Confusion: A Model for Explanation and Prediction of Conceptual Shifts," *Ergonomics*, 57(3):455-470, DOI: 10.1080/00140139.2013.858777 http://dx.doi.org/10.1080/00140139.2013.858777, 2014.

[30] Benoit Vanholme, Benoit Lusetti, Dominique Gruyer, Sébastien Glaser, and Saïd Mammar, "Highly Automated Driving on Highways: System Implementation on PC and Automotive ECUs," *Proceedings of the 14th International IEEE Conference on Intelligent Transportation Systems*, pp. 1465-1470, 2011.

[31] Wikipedia, Air France 447, http://en.wikipedia.org/wiki/Air_France_Flight_447, May 27, 2015.

Maximilian M. Etschmaier's professional work is focused on the analysis, design, and operation of complex systems in a wide variety of domains. He is currently an Adjunct Professor in the College of Sciences at San Diego State University. Previous positions include, Guest Researcher at the National Institute of Science and Technology (NIST), Senior Scientist at the United Technologies Research Center, Chairman of the Management Board of Joanneum Research in Graz, Austria, Vice President of Systems and Control at Bricmont Associates (now Andritz Bricmont), Professor of Engineering at the Universities of Pittsburgh and Massachusetts, Head of Operations Research of Deutsche Lufthansa AG, and Visiting Professor at the University of Graz and University of Innsbruck. He has advised business and public sector clients on policy and strategy development, lead process improvement ventures, and supported international technology transfer.

Dr. Etschmaier is a native of Austria. He holds a PhD in Engineering from the Technical University in Graz, Austria, and an MS in Operations Research from Case Western Reserve University, where he was a Fulbright Scholar. He has participated in national and international scientific and professional organizations, serving in leadership positions, organizing and hosting meetings and sessions, and presenting and publishing numerous papers.

Gordon K. Lee was born and raised in Hawaii. He received his B.S. degree in Electrical Engineering from the University of Hawaii in 1972, his M.S.E.E. degree from the University of Connecticut in 1974 and his Ph.D. degree from the University of Connecticut in 1978. From 1978 through 1989, Dr. Lee was at Colorado State University in the Department of Electrical Engineering where he rose to the level of Full Professor. He was also the Director of the Institute for Robotic Studies.

In 1989, Dr. Lee became a faculty member in the Department of Mechanical and Aerospace Engineering at North Carolina State University and also served as Director of Graduate Programs in the Department of Mechanical and Aerospace Engineering and later as Assistant Dean for Research Programs in the College of Engineering. Dr. Lee joined San Diego State University in December 2000 where he served as the Associate Dean and Director of the Joint Doctoral Program for the College of Engineering. He was also a full Professor in the Department of Electrical and Computer Engineering and is currently Professor Emeritus in that department.

His research interests are in the areas of robotics and intelligent control systems, particularly evolutionary control algorithms, fuzzy systems and neural networks, as well as in the applications of these methods to mobile robotic colonies. His research projects have been funded by government agencies as well as industry. He has published over 275 technical documents; Dr. Lee is a senior member of IEEE, a member of AIAA and a senior member of ISCA. He is also currently an Associate Editor for the International Journal on Intelligent Automation and Soft Computing.

Multiple Regression Analysis and Learning System for Estimation of Blood Pressure Variation Using Photo-Plethysmograph Signals

Michio Yokoyama[†,‡], Takumi Negishi[†], Mitsuru Mizunuma[†]
Yamagata University, Yamagata 992-8510, JAPAN

Kazuya Otani[§], Hidenobu Hanaki[§] and Kozo Nishimura[§]
TOKAI RIKA Co., Ltd, Aichi 480-0195, JAPAN

Abstract

In this paper, a blood pressure estimation system is proposed. Blood pressure variation is estimated by multiple regression analysis using photo-plethysmograph signals. Multiple regression analysis has been performed considering the multicollinearity between explanatory variations. Furthermore, by changing kinds of parameters of the pulse wave used for estimation, improvement of accuracy of blood pressure estimation has been aimed. Experimental results have shown that the estimated blood pressure values have been within about ±10mmHg as compared with measured blood pressure values using a cuff.

Key Words: Blood pressure estimation, multiple regression analysis, photo-plethysmography, infrared LED sensor, systolic/diastolic blood pressure, learning system, correlation, multicollinearity.

1 Introduction

In recent years, the numbers of lifestyle patients and their preliminary group tend to increase, and this leads to social problems. The lifestyle patients are predicted to increase more and more in the future. Since early detection of lifestyle-related diseases is difficult, it is important to manage day-to-day individual health conditions. For the prevention of lifestyle-related diseases, ubiquitous monitoring systems of physical condition have been noticed nowdays.

As an indicator of daily health management, blood pressure values are available. Blood pressure measuring devices on the current market almost always require a cuff. They are too large to handle and it takes several seconds for each measurement. Therefore, to try managing daily health with blood pressure monitoring imposes a burden in daily life. Recently, there have been a few studies on handy blood pressure measurement without a cuff [2, 3] such as phase shift method, pulse wave velocity method, etc.

The purpose of our study is to estimate the variation of blood pressure value with a plurality of parameters obtained from the photo-plethysmograph [4]. In this paper, multicollinearity among the parameters is noticed for the cuff-less blood pressure estimation method with photo-plethysmography. Then, a novel blood pressure estimation method is proposed with consideration of multicollinearity. Moreover, estimated blood pressure values on the vehicle steering have been evaluated as compared with the cuff-measured values.

2 Photo-Plethysmography and Pulse Wave Parameters

2.1 The Principle of Photo-Plethysmography

The photo-plethysmography is adopted here for blood pressure estimation method. Infrared LED/PD with a wavelength of 950nm is utilized. The photo- plethysmography is a method to observe the pulsations of blood vessels by measuring the change in the photo absorption of hemoglobin at the blood vessel flow. Since hemoglobin with/without oxygen has remarkable absorption change mainly in the near-infrared band, pulsation is influenced by the transmitted or reflected light. The near-infrared light emitted from LED is irradiated to the finger; the photodiode receives the transmitted or reflected light that is not absorbed by hemoglobin. Figure 1 shows the pulse wave measurement circuits using photo-plethysmography. The photo-diode (PD) detects the reflection or transmission light, which includes scattering information of blood vessel corresponding to pulsation, from infrared LED through a finger. The detected signal is then filtered and amplified at the subsequent signal processing circuits for waveform shaping.

[†] Jonan 4-3-16, Yonezawa.
[‡] E-mail: yoko@yz.yamagata-u.ac.jp.
[§] 3-260 Toyota, Oguchi-Cho, Niwa-gun.

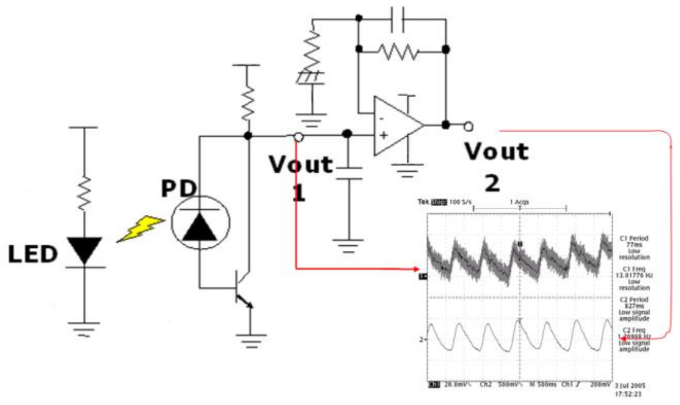


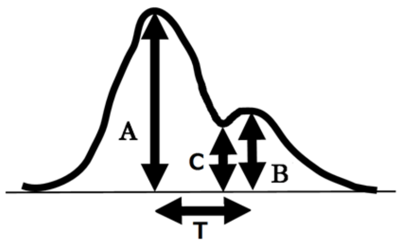
Figure 1: Conceptual diagram of the pulse wave measurement circuits using photo-plethysmography

2.2 Pulse Wave Parameters

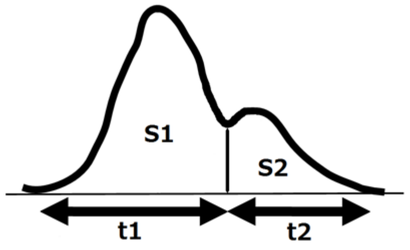
Since pulse wave is closely related to blood pressure, characteristic parameters obtained from the pulse wave are effective for estimating the blood pressure. Multiple regression analysis with the characteristic parameters is utilized to estimate the variations of blood pressure.

The number of parameters for multiple regression analysis is selected among 18 types of pulse wave characteristic parameters, as follows:

- Pulse rate: The number of times of rippling pulse wave per minute.
- Top of ejected wave: Maximum amplitude of wave caused by pumping blood from the heart (A in Figure 2(a)).
- Top of reflected wave: Maximum amplitude of wave caused by reflection at the peripheral vascular (B in Figure 2(a)).
- Incisura: Height of chasm which exists between the ejected and reflected waves (C in Figure 2(a)).
- Reflection Index (RI): Ratio of ejected and reflected waves; RI(％)＝(B/A)×100
- T: Interval between tops of ejected and reflected waves.
- t1: Interval from generation point of pulse wave to incisura.
- t2: Interval from incisura to end of pulse wave.
- T: Period of one pulse wave (t1＋t2).
- S1: Partial area of one pulse wave until incisura.
- S2: Partial area of one pulse wave after incisura.
- S: Area of one pulse wave (S1＋S2).
- A/C: Ratio of the maximum amplitude of ejected wave to the incisura.
- B/C：Ratio of the maximum amplitude of reflected wave to the incisura.
- t1/t2: Ratio of t1 to t2.
- S1/S2: Ratio of S1 to S2.



(a)



(b)

Figure 2: Pulse wave parameters

The changes between the above parameters are used in multiple regression analysis. In this paper, the symbol of Δ represents the changes. The changes are calculated by subtraction of the two pulse wave parameters. Furthermore, the ratio of two wave parameters is also used as the parameters.

- Ejected wave ratio: Increase or decrease ratio of the ejected wave(A’/A).
- Reflected wave ratio: Increase or decrease ratio of the reflected wave(B’/B).

Above these 18 types of pulse wave parameters are candidates for use in blood pressure estimation. Then, 18 types of parameters are narrowed down at the view point of correlation. More information is described below.

3 Overview of Blood Pressure Variation Estimation

3.1 Blood Pressure Variation Estimation System

In order to estimate the variation of blood pressure with the photo-plethysmography, the changes and/or ratios in the pulse wave parameters are used in the estimation equation. The estimation equation is derived from multiple regression analysis [1]. In addition, with use of the blood pressure values measured at the cuff as a reference blood pressure, the current blood pressure value is estimated by simply measuring pulse wave. Figure 3 shows an overview of the practical system. The cuff measured values are also used in the learning system to improve accuracy of the estimation equation.

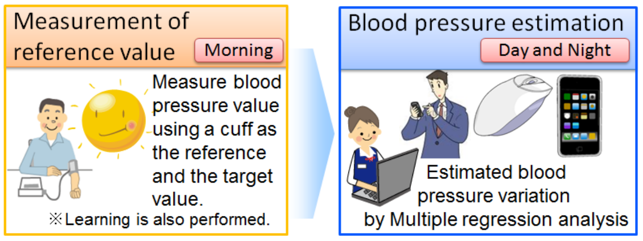

Measurement of reference value
Morning
Measure blood pressure value using a cuff as the reference and the target value.
※Learning is also performed.
Blood pressure estimation
Day and Night
Estimated blood pressure variation by Multiple regression analysis

Figure 3: Overview of practical use of the system

Estimated blood pressure value is calculated by the following formula.

[Estimated blood pressure value] =
[Reference of blood pressure value]
+ [Estimated blood pressure variation (ΔBP)]

3.2 Flowchart of Blood Pressure Estimation

Figure 4 indicates the flowchart of blood pressure estimation.

Variation data of the pulse wave and the blood pressure are prepared. By multiple regression analysis, the estimation equation is derived from these data. In addition, blood pressure and pulse wave for the reference data are also prepared.

1) A pulse wave is measured for estimation.
2) If there is no blood pressure measurement by the cuff, estimation of blood pressure is performed. The changes of parameters from reference data are calculated after measuring the pulse. The changes in the pulse wave parameters are substituted for the estimation equation in order to calculate the estimated variation of blood pressure. The estimated variation of blood pressure is added to the reference blood pressure. Then, this is the estimated blood pressure value.
3) If there is blood pressure measurement by the cuff, learning system is performed to improve accuracy of estimation equation. Once in several times, estimation equation is corrected with cuff value and pulse wave parameters. At the learning, both cuff value and pulse wave are measured at the same time. Adding variation data of pulse wave parameters and the blood pressure to database, multiple regression analysis is re-performed for renewal of the estimation equation.

This learning system is described later in detail.

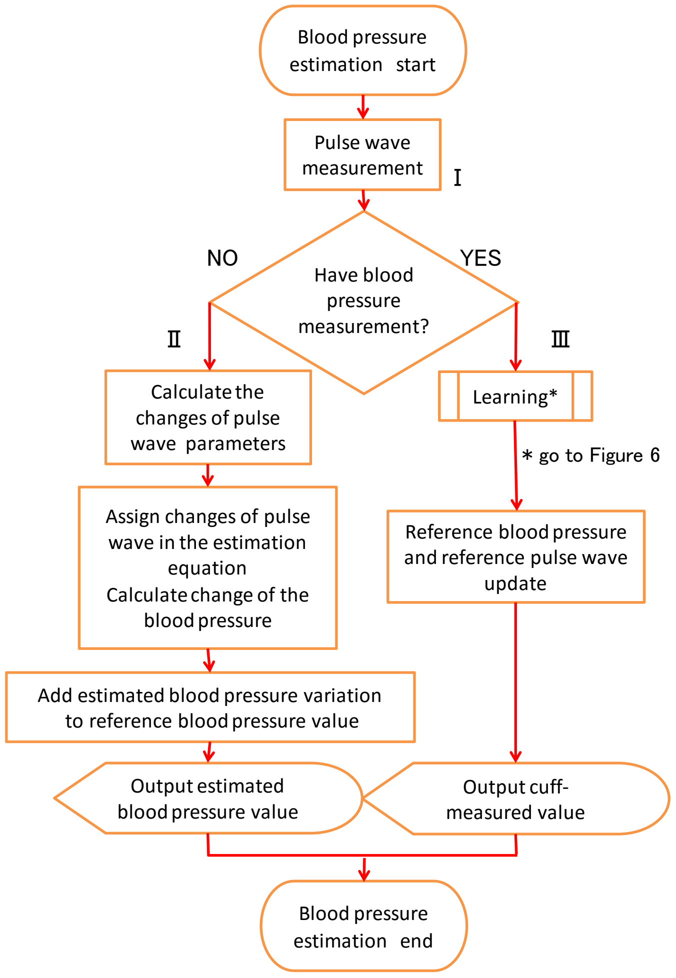

Blood pressure estimation start
Pulse wave measurement
I
Have blood pressure measurement?
NO
YES
II
III
Calculate the changes of pulse wave parameters
Learning*
* go to Figure 6
Assign changes of pulse wave in the estimation equation Calculate change of the blood pressure
Reference blood pressure and reference pulse wave update
Add estimated blood pressure variation to reference blood pressure value
Output estimated blood pressure value
Output cuff-measured value
Blood pressure estimation end

Figure 4: Flowchart of blood pressure estimation

4 The Principle of Blood Pressure Variation Estimation

4.1 Multiple Regression Analysis

The equation used for blood pressure variation estimation is derived by the multiple regression analysis with variation data of blood pressure and pulse wave parameters. The multiple regression analysis is one of the multivariate analysis by using a plurality of data. It is a calculation to derive the estimation equation so that the error may become the minimum. In this work, the changes in pulse wave parameters and blood pressure correspond to explanatory and explained variations, respectively.

Figure 5 indicates a conceptual diagram of multiple regression for blood pressure estimation. Several parameters such as changes in pulse rate, interval T and so on are utilized for calculation of multiple regression analysis in order to estimate the changes in blood pressure (ΔBP). The detailed procedure of estimation equation derivation is the same as the procedure of the learning described in Section 5.

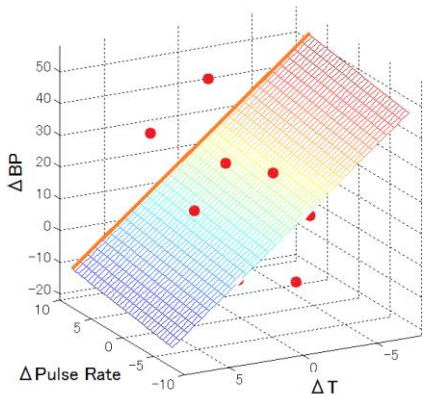


Figure 5: Conceptual diagram of multiple regression equation

4.2 Multiple Correlation Coefficients and Coefficient of Determination

Multiple correlation coefficient here means the correlation coefficient between "measured value" and "estimated value by estimation equation calculated from multiple regression analysis", as represented by R. R is ranged from 0 to 1. If the estimation equation is perfect, R is 1. The coefficient of determination is defined as the squared multiple correlation coefficient. These two coefficients are used at learning.

5 Construction of Estimation Equation and Learning

5.1 Construction of Estimation Equation

There are individual differences in the correlation between pulse wave parameters and blood pressure values. Therefore, the type of pulse wave parameters used for blood pressure variation estimation should not be fixed. So, a correlation test of the blood pressure and pulse wave parameters is performed. The pulse wave parameters which have correlation with the blood pressure are candidates for use in estimating.

The presence of multicollinearity is responsible for lowering the estimation accuracy. To investigate the presence of multicollinearity, the coefficient of determination of parameters should be examined. When the coefficient of determination is more than a threshold, the combination of parameters is excluded.

The combination of high estimation accuracy is examined. Among the remaining combinations, correlation between the measured and the estimated blood pressure values is tested. The combination whose determination coefficient is the highest is selected for estimation.

Construction of estimation equation is performed at the time of the learning and at the beginning of using the system. In this work, learning is defined as "to update the estimation equation at each time of cuff blood pressure measurement."

5.2 Procedure of Learning

Figure 6 indicates the flowchart of learning.

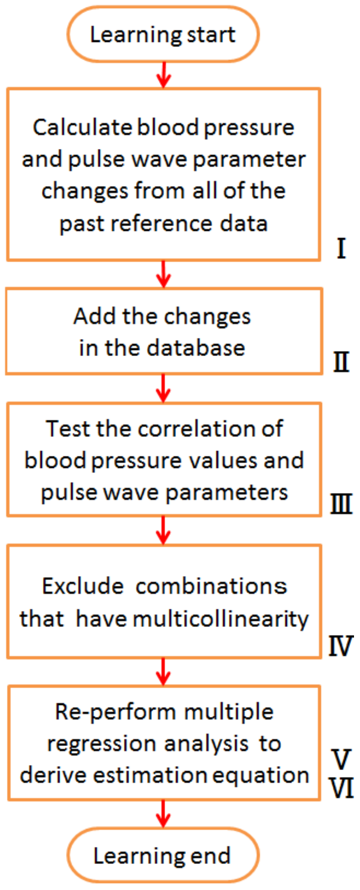

Learning start
Calculate blood pressure and pulse wave parameter changes from all of the past reference data
I
Add the changes in the database
II
Test the correlation of blood pressure values and pulse wave parameters
III
Exclude combinations that have multicollinearity
IV
Re-perform multiple regression analysis to derive estimation equation
V
VI
Learning end

Figure 6: Flowchart of learning system of estimation equation

1) In learning, blood pressure and pulse wave are measured at the same time. These measured values are replaced as the new reference values. Then, the changes between the new and old reference values are calculated.
2) The changes are added in the database for multiple regression analysis.
3) The correlation tests of the blood pressure value and pulse wave parameter in the database are performed. Pulse wave parameters with high correlation are candidates to be used for estimation.
4) In order to examine the presence of multicollinearity in all possible combinations of candidate parameters, the coefficients of determination are calculated. One of the parameters is estimated by the other parameters in combination. Correlation between the estimated and the actual parameters is calculated. If there is even one coefficient of determination of the parameter which is greater than 0.5, the combinations are excluded.
5) Coefficients of determination of the blood pressure value are calculated from the measured and the estimated blood pressure values by the remaining combinations in 4). Among the combinations, when the combination whose coefficient of determination of blood pressure values is the highest, this combination is adopted to estimation.
6) Multiple regression analysis is re-performed with the combination of parameters adopted in 5) and blood pressure. Then, a new estimation equation is derived.

5.3 The Test of Correlation

In order to narrow down candidates of the explanatory variables, the correlation between each pulse wave parameter and blood pressure value is tested with variation data in database. The data is assumed to be in accordance with the normal distribution. Pearson's correlation coefficient to be used in the present study is represented by -1 to 1. T-test is used for the test of the correlation; hazard ratio $\alpha = 0.05$. Parameters which have no correlation are excluded from the candidates in this procedure.

5.4 Multicollinearity

Multicollinearity exists when there is a high correlation among the explanatory variables with each other. The higher dependence the explanatory variables have, the more accuracy of estimation equation degrades. The influence of multicollinearity is predicted by calculating with relationship between one of the pulse wave parameters and others.

If there is even one coefficient of determination of the explanatory variable which is greater than 0.5, the combinations of parameter are excluded for estimation.

6 Experiments

6.1 Experimental Methodology

For the measurement of the photo-plethysmography, PowerLab4/26(AD instruments Pty Ltd) is used. First, a sensor mounted on the steering is examined to determine whether a pulse wave can be measured stably. Then, the pulse wave of the left hand thumb is measured for 30 seconds. The finger pressing force on a steering wheel is measured by means of the force gauge; AD-4932A-50N(A&D Co., Ltd.) for reference.

For the measurement of systolic and diastolic blood pressures with cuff, HEM-1020 (Omron Healthcare Co., Ltd.) is used. Blood pressure values are measured four times in total. They are measured twice each before and after pulse wave measurements. The average of four blood pressure values is calculated. Although correction by cuff values is once every 10 estimations in this system, cuff values are measured every time. They are saved for evaluation of the estimation accuracy.

These measurements are one set, and total of 105 sets are measured. The first five sets are used to derive the initial estimation equation. Ten variation data obtained by calculating the changes and/or the ratios of the first five blood pressures and pulse wave parameters are used to derive the initial estimation equation. The estimated blood pressure of 100 sets of data is utilized to evaluate the estimation accuracy. The learning is performed every 10 estimations.

6.2 Experimental Result

Pulse wave is measured with the sensor on the steering. Figure 7 is a photograph of the pulse wave sensor. Figure 8 represents measured pulse waves with the mounted sensor.

Figure 8 shows that pulse wave has been successfully measured with the mounted sensor. The measured finger pressing force on the steering has ranged from about 1N to 3N. Since a pulse waveform shape of photo-plethysmography is affected by finger pressure, the research subject has tried to keep a moderate grasping force constantly for a period of the experiment.

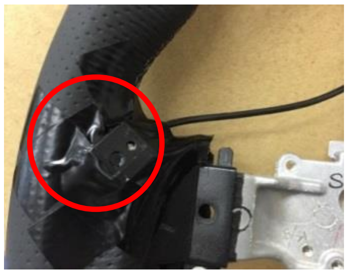


Figure 7: Sensor mounted on the steering

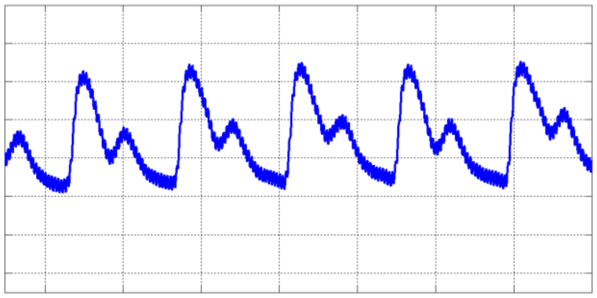


Figure 8: Pulse wave measured by the sensor

The result of the systolic blood pressure variation estimation is shown in Figure 9. Blood pressure values of the target cuff values are red-solid line, estimated blood pressure values are blue-solid line, and timing of the learning is yellow-square.

Estimation error of the systolic blood pressure is within ± 10mmHg as shown in Figures 9 and 10. Therefore, it has been found that this method is effective for systolic blood pressure estimation.

On the other hand, the result of the diastolic blood pressure variation estimation is shown in Figure 11. Diastolic blood pressure values of the target cuff values are red-solid line, estimated blood pressure values are blue-solid line, and timing of the learning is yellow-square. The correlation of pulse wave parameters with diastolic blood pressure value was tested in addition to that with systolic blood pressure. The selected pulse wave parameters with a high correlation for diastolic blood pressure estimation analysis were different from those in the systolic blood pressure estimation case.

Estimation error of the diastolic blood pressure is shown in Figure 12. The estimation error within ± about 10mmHg has shown in case of the diastolic blood pressure estimation. Accordingly, this estimation method is found to be also effective for diastolic blood pressure estimation. For improvement of estimation accuracy, research subjects and measurement experiments should be increased more intensely from the viewpoints of individual customizing with learning in estimation equation and selection of correlative pulse-wave parameters with blood pressures for estimation.

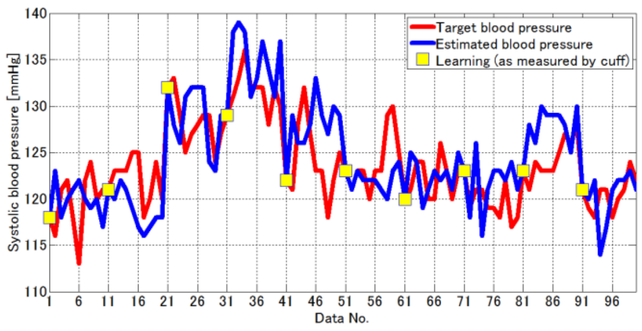


Figure 9: Estimation results of the systolic blood pressure

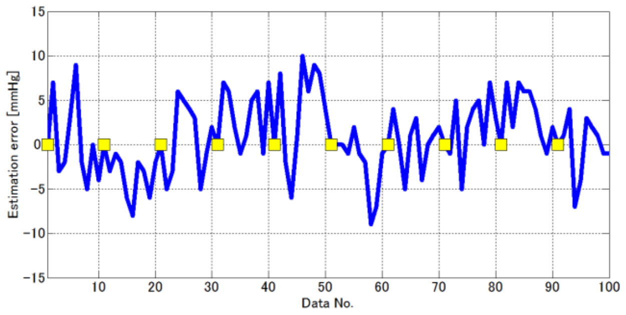


Figure 10: Estimation error of the systolic blood pressure

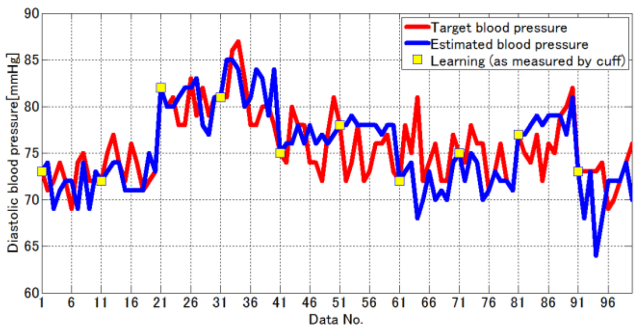


Figure 11: Estimation results of the diastolic blood pressure

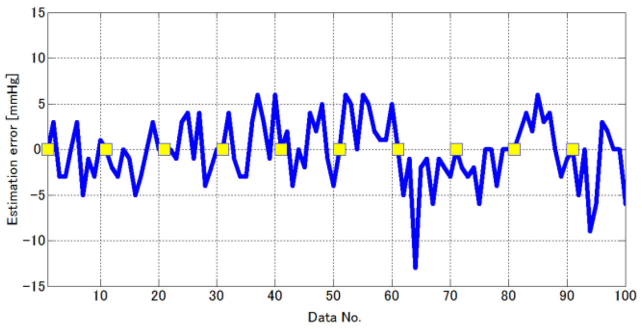


Figure 12: Estimation error of the diastolic blood pressure

For influence of finger pressure in grasping a steering on estimation, the proposed estimation method with the multiple regression analysis seems to be robust for the fluctuation of finger pressure to a certain extent. In spite of fluctuation, this is considered to be caused by the nature of multiple regression analysis itself with selection of plural correlative parameters and learning. Moreover, improvement of this estimation method should be investigated with regard to soft- and/or hard-ware solution for the pressure fluctuation.

7 Conclusions

It has been found that variations of blood pressure are estimated with multiple regression analysis of photo-plethysmograph signals. As a result of consideration of multicollinearity between explanatory variations and learning system after using selected pulse wave parameters for analysis, the estimated systolic and diastolic blood pressure values have been both within about ±10mmHg as compared with measured blood pressure values using a cuff.

Furthermore, in order to improve the estimation accuracy, there still exist several problems to be solved such as selection of highly-correlative parameters, stable measurement of pulse wave under finger pressures fluctuation, and so on.

References

[1] R. E. Ogunsakin, R. B. Ogunrinde, O. Omotoso, and O. B. Adewale, "On Regression Analysis of The Relationship Between Age and Blood Cholesterol on Blood Pressure," *International Journal of Scientific & Technology Research*, 1(9):92-94, October 2012.

[2] S. Omata and M. Haruta, "Development of Cuff-Less Blood Pressure Measuring Instrument and Its Institute of Application to Telemedicine," Jpn. Society for Medical and Biological Engineering, Japan, (in Japanese), 2014.

[3] S. Suzuki and K. Oguri, "Cuffless Blood Pressure Estimation by Error-Correcting Output Coding Method Based on an Aggregation of AdaBoost with a Photoplethysmograph Sensor," Engineering in Medicine and Biology Society, 2009.

[4] T. Tamura, "Blood Pressure Variation Estimation Using Photo-Plethysmography", Master's Thesis, Yamagata University, Japan, 2014 (in Japanese)

Michio Yokoyama was born in Yamagata, Japan, on March 1, 1967. He received his B.E. degree in Electrical Engineering from Yamagata University, in 1989, and his M.E. degree in Electrical Engineering and his Ph.D. degree in Electronic Engineering from Tohoku University in 1991 and 1994. In 1994, he joined Research Institute of Electrical Communication, Tohoku University, Sendai, Japan, where he was engaged in research design and development of RF-CMOS circuits for digital cellular phone system. In 2001, he joined Yamagata University, where he has engaged in research on ubiquitous healthcare system. He is a member of Japan Electronics Packaging, Japan Society of Applied Physics, Institute of Electrical Engineers of Japan and Institute of Electronics, Information and Communication Engineers.

Takumi Negishi received his B.E. degree in Bio-Systems Engineering from Yamagata University in 2014. He is currently a student for M.E. degree in Bio-Systems Engineering at Graduate School of Science and Engineering, Yamagata University. His research interests include ubiquitous healthcare system, vital signal sensing & analysis.

Mitsuru Mizunuma was born in Yamagata, Japan, on July 15, 1950. He received the Associate degree in engineering from Technical College, Yamagata University, Japan, in 1973. He became a technical official of the Department of Electronic Engineering, Faculty of Engineering, Yamagata University in 1970, where he engaged in research on electronic circuits, distributed constant circuits, image processing, fuzzy theory, and digital/analog integrated circuits design. He is currently a technical staff member in the Department of Bio-System Engineering, Faculty of Engineering, Yamagata University. His research interests include the design of the ubiquitous physical condition grasp support system, and it includes the design of motion sickness simple measurement, evaluation and warning system.

Kazuya Otani (photo and bio not available).

Hidenobu Hanaki (photo and bio not available).

Kozo Nishimura (photo and bio not available).

Measuring the Impact of Network Performance on Cloud-Based Speech Recognition

An Empirical Study of Apple Siri and Google Speech Recognition

Mehdi Assefi,[*] Guangchi Liu,[†] Mike P. Wittie,[‡] Clemente Izurieta[§]
Montana State University, Bozeman, MT 59715, USA

Abstract

Cloud-based speech recognition systems enhance Web surfing, transportation, health care, etc. For example, using voice commands helps drivers search the Internet without affecting traffic safety risks. User frustration with network traffic problems can affect the usability of these applications. The performance of these type of applications should be robust in difficult network conditions. We evaluate the performance of several client-server speech recognition applications, under various network conditions. We measure transcription delay and accuracy of each application under different packet loss and jitter values. Results of our study show that performance of client-server speech recognition systems is affected by jitter and packet loss, which commonly occur in WiFi and cellular networks.

An experimental study on client-server speech recognition applications is reported in *Impact of the network performance on cloud-based speech recognition systems*, in which a solution that uses network coding to improve the performance of cloud-based speech recognition applications has been proposed. The aforementioned paper is published in ICCCN 2015 [8]. Designing and implementing of experimental testbeds by using TCP and UDP connections and also designing and implementing another testbed that uses fountain codes on UDP connection has been introduced in the paper. In this paper, we design and implement an extensive experimental evaluation of five client-server speech recognition applications to compare the performance of these applications under different network conditions.

Key Words: Cloud Speech Recognition, Quality of Experience, Software Measurement, Streaming Media, Real-time Systems.

1 Introduction

Performance evaluation of cloud-based speech recognition systems under different network conditions has received much less attention than other streaming systems. Although Apple Siri and Google Speech Recognition (GSR) are popular applications that help users to interact with search engines using voice commands, an experimental evaluation of these applications is noticeably missing.

Delay and accuracy of the voice recognition process is an important parameter that affects the quality of a user's experience with cloud-based speech recognition applications. Streaming voice from the client to the server and converting it to text are two phases of this process and should have the low delay and high accuracy in order to satisfy the quality of a user's experience. Delays of this process should also be consistent under all different network conditions.

In this paper, we describe the design and implementation of an experimental evaluation of Siri and GSR. We also evaluate three client-server speech recocgintion systems using TCP, UDP, and network coding over UDP. We evaluate these applications under different packet loss and jitter values and measure the delay and accuracy of each under different network conditions. Specifically, we employ four statistical models to evaluate the effects of packet loss and jitter, respectively. Each model is designed to evaluate two factors (jitter and packet loss) with two blocking variables on the response variable - delay and accuracy. The blocking variable is the application for all experiments. The ANOVA test is used to evaluate effects of packet loss and jitter for each experiment respectively. Results of our study show that delays in all applications are affected by packet loss and jitter. Results also show that the accuracy of three applications is affected by packet loss and jitter.

The remainder of this paper is organized as follows: In Section II we explore related work. In Section III we describe our experimental methods. In Section IV we describe overall results. In Section V we describe our experimental design and the mathematical model used to analyze experimental data. Section VI discusses results. Finally, in Section VII we discusses threats to validity of our experiment and conclude in Section VIII.

2 Related Work

Yang Xu *et al.* performed a measurement study on Google+, iChat, and Skype [31]. They explored the architectural features

[*]Department of Computer Science, Email: mehdi.assefi@msu.montana.edu.
[†]Department of Computer Science, Email: guangchi.liu@msu.montana.edu.
[‡]Department of Computer Science, Email: mwittie@montana.edu.
[§]Department of Computer Science, Email: clemente.izurieta@montana.edu.

of these applications. Using passive and active experiments, the authors unveiled some performance details of these applications such as video generation and adaption techniques, packet loss recovery solutions, and end-to-end delays. Based on their experiments the server location had a significant impact on user performance and also loss recovery in server-based applications. They also argued that using batched re-transmissions was a good alternative for real time applications instead of using Forward Error Correction (FEC) –an error control technique in streaming over unreliable network connections.

Te-Yuan Huang *et al.* did a measurement study on the performance of Skype's FEC mechanism [21]. They studied the amount of the redundancy added by the FEC mechanism and the trade-offs between the quality of the users' experience and also the resulting redundancy due to FEC. They tried to find an optimal level of redundancy to achieve the maximum quality of the users' experience.

Te-Yuan Huang *et al.* also performed a study on voice rate adaption of Skype under different network conditions [20]. Results of this study showed that using public domain codecs was not the ideal choice for users' satisfaction. In that study, they considered different levels of packet loss in their experiments and created a model to control the redundancy under different packet loss conditions.

Kuan-Ta Chen *et al.* proposed a framework for user QoE measurement [11]. Their proposed framework, OneClick, provided a dedicated key that could be pressed by users whenever they felt unsatisfied by the network conditions with streaming media. OneClick was implemented on two applications – instant messaging applications and shooter games.

Another framework that quantified the quality of a user's experience was proposed by Kuan-Ta Chen *et al* [12]. The proposed system was able to verify participants' inputs, so it supported crowd-sourcing. Participation is made easy in this framework. The framework generates interval-scale scores. They argue that researchers can use this framework for measuring the quality of a users' experience without affecting quality of the results and achieve a higher level of diversity in users' participation while also keeping cost low.

Budzisz *et al.* proposed and developed a delayed-based congestion control system [10]. The proposed system offers low standing queues and delay in homogeneous networks, and balanced delay-based and loss-based flows in heterogeneous networks. They argue that this system can achieve these properties under different loss values, and outperform TCP flows. Using experiments and analysis, they demonstrate that this system guarantees aforementioned properties.

Hayes *et al.* proposed an algorithm which tolerates non-congestion related packet loss [18]. They proved experimentally that the proposed algorithm improves the throughput by 150% under packet loss of 1% and improves the ability to share the capacity by more than 50%.

Akhshabi *et al.* proposed an experimental evaluation of rate adaption algorithms for streaming over HTTP [4, 5]. They experimentally evaluated three common video streaming applications under a range of bandwidth values. Results of this study showed that congestion control of TCP and its reliability requirement does not necessarily affect the performance of such streaming applications. Interaction of rate-adaption logic and TCP congestion control is left as an open research problem.

Chen *et al.* experimentally studied performance of multipath TCP over wireless networks [13]. They measured the latency resulting from different cellular data providers. Results of this study show that Multipath TCP offers a robust data transport under various network traffic conditions. Studying the energy costs and performance trade-offs should be considered as a possible extension of this study.

Google is currently working on a new transport protocol for the Internet called QUIC (Quick UDP Internet Connections) [25]. QUIC uses UDP and solves problems of packet delay under different packet loss values in TCP connections. QUIC solves this problem by multiplexing and FEC.

An experimental investigation on the Google Congestion Control (GCC) in the RTCWeb IETF WG was performed by Cicco *et al.* [14]. They implemented a controlled testbed for their experiment. Results of this experimental study show that the proposed algorithm works well but it does not utilize the bandwidth fairly when it is shared by two GCC flows or a GCC and a TCP flow.

Cicco *et al.* have also experimentally investigated the High Definition (HD) video distribution of Akamai [15]. They explained details of Akamai's client-server protocol which implements the quality adaption algorithm. Their study shows that the proposed technique encodes any video at five different bit rates and stores all of them at the server. The server selects the bit rate that matches the bandwidth that is measured based on the signal received from the cilent. The bitrate level adaptively changes based on the available bandwidth. Authors of the paper also evaluated the dynamics of the algorithm in three scenarios.

Winkler *et al.* ran a set of experiments to asses quality of experience on television and mobile applications [28, 29]. Their proposed subjective experiment considers different bitrates, contents, codec, and network traffic conditions. Authors of the paper used Single Stimulus Continous Quality Evaluation (SSCQE) and Double Stimulus Impairment Scale (DSIS) on the same set of materials and compared these methods and analyzed results of experiments in view of codec performance.

A mesh-pull-based P2P video streaming using Fountain codes is proposed by Oh *et al.* [24]. The proposed system offers fast and smooth streaming with low complexity. Experimental evaluations show that the proposed system has better performance than existing buffer-map-based video streaming systems under packet loss values. Considering jitter as another important factor and evaluation of behavior of the proposed system considering jitter values can be a potential extension of this study.

Application of Fountain Multiple Description Coding (MDC) in video streaming over a heterogeneous peer to peer network is considered by Smith *et al.* [26]. They conclude that Fountain MDC codes are favorable in such cases, but there are some restrictions in real-world P2P streaming systems.

Finally, Vukobratovic *et al.* proposed a novel multicast streaming system that is based on Expanding Window Fountain (EWF) codes for real-time multicast [27]. Using Raptor-like precoding has been addressed as a potential improvement in this area.

3 Experimental Testbeds

We design and implement our experimental testbed to study the performance of cloud-based speech recognition systems under loss and jitter. Clients of such systems transmit voice data through a network traffic shaper, where we change jitter and packet loss values in the communication network. We set a bandwidth to 2Mbps which is typical on 3G connections [19]. The server receives voice data, translates the voice into text, and sends the text and search results based on the converted text to the client. The client calculates the delay of the server response. To calculate the accuracy of transcription we use Levenshtein distance [32]. Accuracy is measured as the match percentage of the original string used to generate the voice and the resulting transcription. The client uses Wireshark Version 1.12.4 to timestamp the traffic of voice transmission to and from the server [6]. We developed a Windows application using Visual C# to timestamp the voice playback. All experiments are performed on a Windows 7 platform for GSR, and on iOS 7.0 for Siri. The traffic shaper is a `netem` box which runs the Fedora Linux operating system. We ran our experiment 30 times for each value of loss and jitter and for each cloud speech recognizer.

3.1 Experimental Testbed for GSR

We use the GSR service available in Google Chrome. There is also another alternative for using Google voice recognition. Google offers a voice recognition Web service that can be used in Windows applications. Figure 1 shows the architecture of our experimental setup.

Clients transmit voice packets to the Google server through the `netem` box that changes network traffic performance. We used a recorded voice with a length of 26.4 seconds for all experiments in order to have a consistent measurement. Google starts to recognize voice as soon as it receives the first voice packet, and sends converted text back to the client. The client records the time of each packet and also voice transmission time to calculate the transcription time of the experiment. The client also compares the resulting text to the original string which was used to generate the voice command and calculates transmission accuracy using the Levenshtein distance [32].

3.2 Experimental Testbed for Siri

The experimental setup for Siri is similar to GSR. We use an iPhone as the client. A client is connected to the Internet through

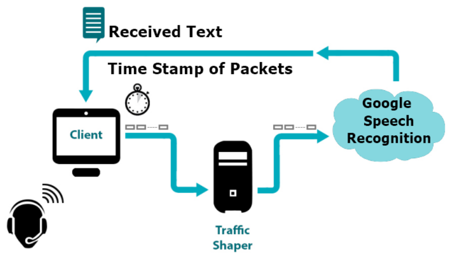


Figure 1: Experimental testbed for GSR.

a WiFi router then to a `netem` box. Here we also used Wireshark to timestamp the transmission of voice packets and reception of results from the Siri server. Figure 2 depicts this setup.

3.3 Experimental Testbed for Nuance Dragon

We consider key characteristics of Siri and GSR to design an in-lab testbed that shows the same behavior. Siri and GSR both use TCP transport protocol [7, 17]. To replicate speech recognition algorithms we used Nuance technology which uses the same algorithms to convert the voice to the text as Siri [1, 3]. Nuance technology is available as Dragon Naturally Speaking software [2].

Our testbed consists of a client that is connected to a speech recognition server through `netem`. Client streams the voice over a TCP connection that goes through the `netem` box. The server starts to convert voice to the text as soon as it receives the first voice packet. The server sends the resulting text back to the client. We record the accuracy of the returning text and the round trip time of the process to evaluate the performance of the system. We repeat the experience 30 times for each traffic setup. Figure 3 shows the architecture of the TCP streaming testbed.

The program timestamps when the voice playback starts and finishes. We call these timestamps fct and lct (first client transfer and last client transfer), respectively. Network traffic conditions are controlled by `netem`. A logger on the server is responsible for keeping the timestamp of packets and storing the first and last timestamps in a file. We call these timestamps fsr and lsr (first server packet received and last server packet received), respectively. In order to timestamp the transcription delay, we developed a text editor to collect the Dragon's output and timestamp the time when the first and last character was created by Dragon. We call these timestamps ffr and lfr (first text file character received and last text file character received), respectively. Every time a new character is created by dragon, our text editor sends that character to the sender and a program on the sender collects the received characters and stamps the time of the first and the last received character. We call these timestamps fcr, and lcr (first client received and last client received), respectively. Figure 4 shows the data flow from the client to the server and also the data flow from Dragon's output to the client. This figure also shows the relative order of the timestamp variables used for our evaluation.

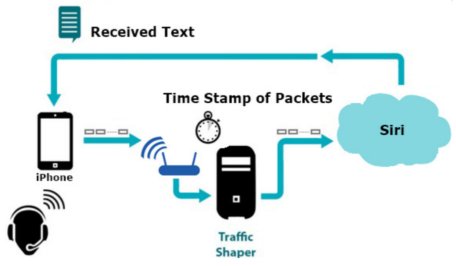


Figure 2: Experimental testbed for Siri.

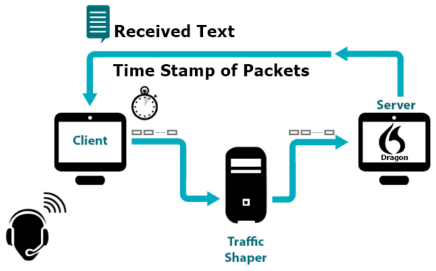


Figure 3: Experimental testbed for TCP.

The recorded timestamps for each round of the experiment monitor the behavior of different parts of the testbed. (ffr - fsr) represents response time of the Dragon, (lfr - fsr) represents the total time of the speech recognition on the server, (lcr - fct) represents the total time of each round of experiment. We used (lcr - lct) as the delay of the remote speech recognition system.

3.4 Experimental Testbed for UDP

TCP waits for each packet to be received and retransmits lost packets. Reliable transmission is not necessarily a good choice for real-time communications, in which transmission delay reduces the feeling of interactivity. UDP is a good alternative when the application tolerates moderate packet loss. We changed our TCP testbed to send UDP packets to observe the effect of packet loss and jitter on delay and accuracy of the speech recognition software. The UDP testbed has the same architecture as TCP, but the streaming part of the testbed has been changed to use UDP packets. We ran the UDP testbed with the same conditions as the TCP.

3.5 Experimental Testbed for UDP with Network Coding

We implemented a P2P streaming system using a linear fountain and replaced it with the standard UDP stream. Other parts of the testbed are the same.

3.5.1 Fountain Codes

Fountain codes are used in erasure channels such as the Internet. Channels with erasure transmit files in multiple small packets and each packet is either received without error or is lost [23]. Coded packets sent to the receiver are combinations

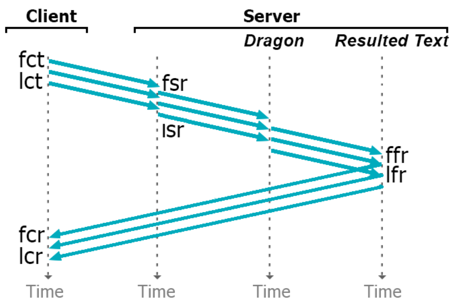


Figure 4: Timestamp Variables.

of original packets. Once the receiver receives enough coded packets it is able to decode and extract the original packets. Figure 5 illustrates the mechanism behind the fountain codec that is used in our solution [26]. Sender takes a group of packets, creates a number of coded packets, and sends them to the receiver along with information needed for their decoding. The receiver extracts the original packets after receiving enough coded packets by solving a linear equation created by the received information.

3.5.2 Fountain Encoder

The Fountain encoder generates an unlimited number of encoded packets using original ones. In order to decode packets of a stream, we group every X consecutive original packets together. Fountain encoder generates enough number of coded packets using original packets of the group, and we will find this number later in this section. Each encoded packet is a bit-wise sum of packets of group:

$$EP_n = \sum_{x=1}^{X} P_x G_{xn}, \tag{1}$$

where G_{xn} is a random binary number consisting of X bits and P's are original packets. The sum operation is done by XOR-ing packets. The resulting packet is sent to the receiver and G_{xn} is also put in the header for the decoder to be able to extract original packets after receiving enough number of coded packets. Figure 6 demonstrates the process of coding and sending packets over a lossy network. Grey shaded packets are not received. The sender creates and sends n coded packets from each group. In order to have enough information to extract the original packets, n should be greater than X. The number of coded packets required to be received by the receiver to have probability 1-δ of decoding success is $\approx$ X+log_2 1/δ [23].

3.5.3 Fountain Decoder

With enough number of received packets, the receiver is able to extract original packets. Lets say there are X original packets and the receiver has received K packets. The binary numbers that we used in our encoder make a K-by-X matrix. If K<X, the decoder does not have enough information to extract the original

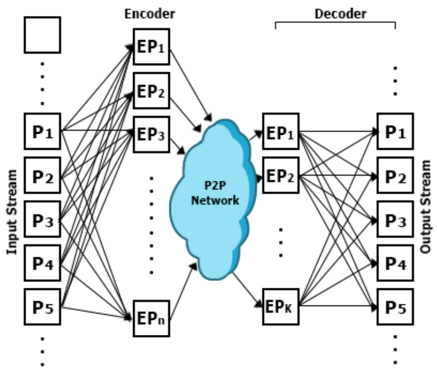


Figure 5: Coding and sending packets over a lossy network [8].

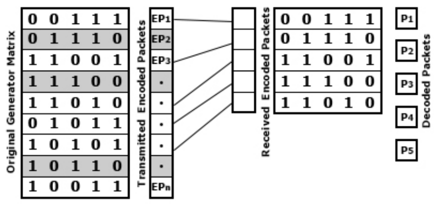


Figure 6: The generator matrix of the linear code [8].

packets. If k=X, it is possible to recover packets. If the resulting K-by-K matrix is invertible, the decoder is able to calculate the inverse of G^{-1} by Gaussian elimination and recover

$$t_x = \sum_{k=1}^{K} t_k G_{kx}^{-1}. \quad (2)$$

The probability that a random K-by-K matrix is invertible is 0.289 for any K greater than 10 [23]. The decoder should receive extra packets to increase the probability of having an inversible matrix. The time complexity of encoding and decoding of linear Fountain codes are quadratic and cubic in number of encoded packets but this is not important when working with packets less than a thousand [23]. Using faster versions of fountain codes, like the LT code or Raptor codes offers less complexity [16].

4 Overall Results

To investigate the effect of packet loss and jitter on delay and accuracy, we generate packet loss from 1% to 5% and jitter from 20 ms to 200 ms respectively on our testbeds and observe the resulting accuracy and delay. Siri and GSR both keep 100% accuracy under high values of packet loss and jitter, so we just consider accuracy values for the other three testbeds. The effect of packet loss and jitter on roundtrip delay of applications is shown in Figures 7 to 16, where the y axis displays delay(s), and the x axis displays packet loss (percentile). and jitter (ms), respectively. There are increasing trends as packet loss and jitter increases, for all applications. For GSR, an increase of 1 packet loss unit (percentile), leads to delay increases in the range of 0-100 ms. An increase of 1 unit (20 ms) in jitter leads to increases in delay from 0-100 ms. In addition, the variance of delay also increases as packet loss and jitter increase, indicating a trend of instability. For Siri, the increase in 1 unit (percentile) packet loss leads to increases in delay of 200 ms; which is worse than GSR. On the other hand, jitter has less impact on delay. In addition, the variance of delay is unchanged, compared to GSR. For Dragon under TCP, packet loss and jitter both affect the roundtrip delay. Variance of delay increases as jitter increases, indicating a trend of instability in the case of high values of jitter. For Dragon under UDP, packet loss does not affect the roundtrip delay, but variance of delay increases as we increase the packet loss. Jitter, on the other hand, affects the roundtrip delay of Dragon testbed under UDP. Variance of delay also increases with increasing jitter. Figures 19 and 22 show that using the network coding with UDP improves the accuracy of UDP when packet loss increases. Comparing the results from Figure 18, we can say that the accuracy of Dragon under UDP with using Fountain codes has been improved by about 30% in the existence of high values of packet loss. Comparing the results from Figures 22 and 20 shows that the accuracy of Dragon under UDP improves under different values of jitter, if we apply Fountain coding. Results show that the accuracy of Dragon has been improved to 85% with 200ms jitter, and this value is 30% when we do not use Fountain coding. From Figures 17 and 20, we can see the effect of increasing packet loss and jitter on the accuracy of Dragon under TCP. Accuracy of Dragon under TCP decreases by 15% when jitter is 200ms. Figure 20 also shows that the accuracy does not change when jitter is between 0 to 100ms and after this point, system starts to lose the accuracy. Variances of accuracy also start to increase from this point.

5 Experiment Design

5.1 Model

Since our data is collected by varying jitter and packet loss respectively, we designed four statistical models to assess the effect of jitter and packet loss on delay and accuracy, respectively. Also, since our data is collected from five applications (i.e. Siri, GSR, fontain, TCP and UDP), we take the application as a blocking variable. Each model contains one factory and one blocking variable. For the first model, the response variable is delay, the independent variable is jitter and the blocking variable is application. Also, to guarantee the assumptions still hold for the following ANOVA tests, we apply log transformation on the response variable. Hence, the first model can be expressed as:

$$log(y_{dij}) = \mu + \alpha_i + \beta_j + e_{ij} \quad (3)$$

where α is jitter, β is application.

For the second model, the response variable is delay, the independent variable is packet loss and the blocking variable is application. The model can be expressed as:

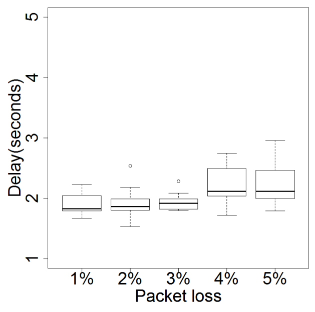


Figure 7: Impact of packet loss on delay of GSR

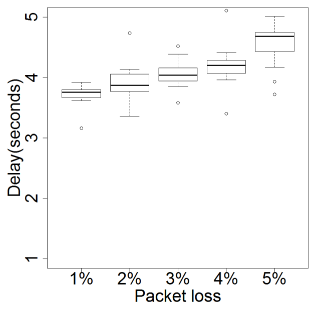


Figure 8: Impact of packet loss on delay of Siri

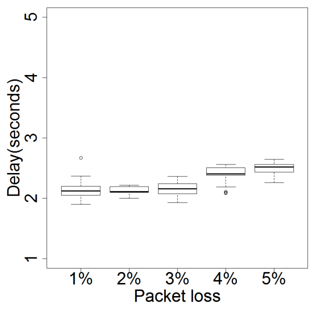


Figure 9: Impact of packet loss on delay of Dragon with TCP

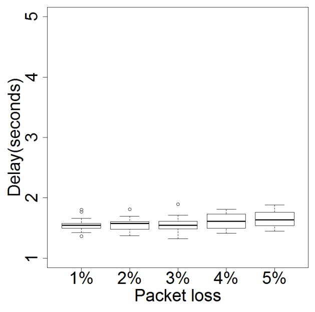


Figure 10: Impact of packet loss on delay of Dragon with UDP

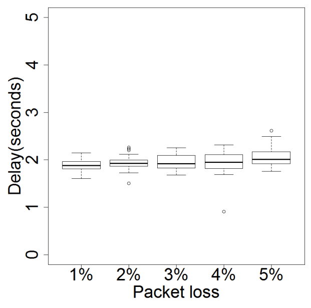


Figure 11: Impact of packet loss on delay of Dragon with UDP by using network coding

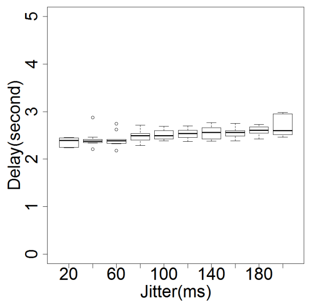


Figure 12: Impact of jitter on delay of GSR

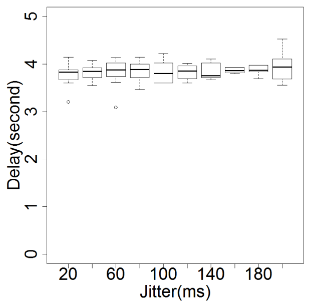


Figure 13: Impact of jitter on delay of Siri

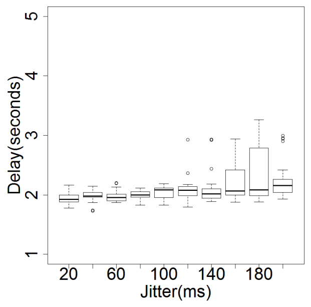


Figure 14: Impact of jitter on delay of Dragon with TCP

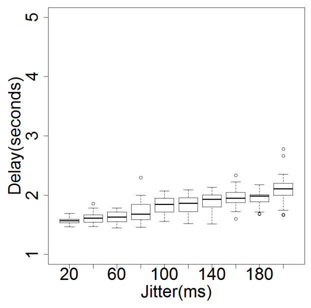


Figure 15: Impact of jitter on delay of Dragon with UDP

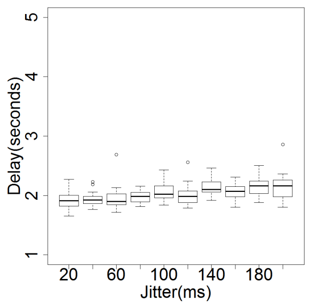


Figure 16: Impact of jitter on delay of Dragon with UDP by using network coding

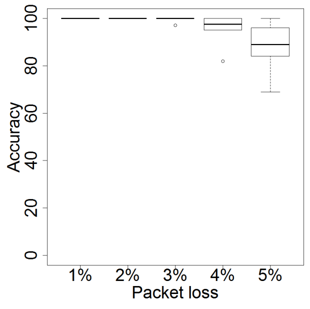


Figure 17: Impact of packet loss on accuracy of Dragon with TCP

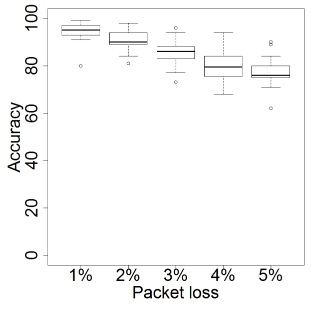


Figure 18: Impact of packet loss on accuracy of Dragon with UDP

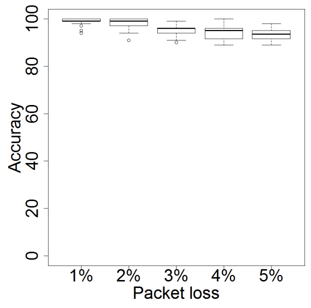


Figure 19: Impact of packet loss on accuracy of Dragon with UDP by using network coding

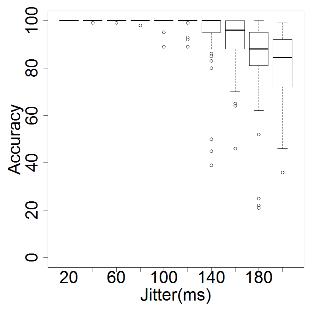


Figure 20: Impact of jitter on accuracy of Dragon with TCP

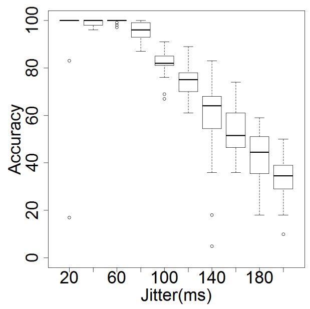


Figure 21: Impact of jitter on accuracy of Dragon with UDP

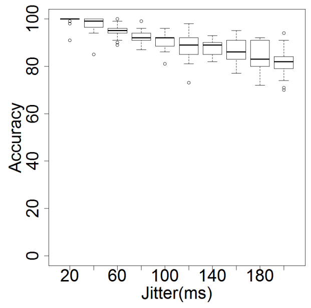


Figure 22: Impact of jitter on accuracy of Dragon with UDP by using network coding

$$log(y_{dij}) = \mu + \gamma_i + \beta_j + e_{ij} \quad (4)$$

where γ is packet loss, β is application.

For the third model, the response variable is accuracy, the independent variable is jitter and the blocking variable is application. The model can be expressed as:

$$log(y_{aij}) = \mu + \alpha_i + \beta_j + e_{ij} \quad (5)$$

where α is jitter, β is application.

For the fourth model, the response variable is accuracy, the independent variable is packet loss and the blocking variable is application. The model can be expressed as:

$$log(y_{aij}) = \mu + \gamma_i + \beta_j + e_{ij} \quad (6)$$

where γ is packet loss, β is application.

For model 3, the factor (jitter) has 10 alternatives, which are the jitter duration ranging from 20 to 200 ms. For model 4, the factor (packet loss) has 5 alternatives, which are the proportion of lost packet ranging from 1% to 5%. The blocking variable for both models 4 and 3 have 5 alternatives, which are SiRi, GSR, fontain, TCP and UDP, respectively. For model 5, the factor (jitter) has 10 alternatives, which are the jitter duration ranging from 20 to 200 ms. For model 6, the factor (packet loss) has 5 alternatives, which are the proportion of lost packet ranging from 1% to 5%. The blocking variable for both models 6 and 5, however, have only three alternatives, which are fontain, TCP and UDP, respectively. The reason is that for SiRi and GSR, the accuracy is always 100%, no matter how the factor changes. Therefore, we ignore them for accuracy.

5.2 Assumptions

To guarantee the effectiveness of ANOVA test, some assumptions should be checked before conducting the ANOVA tests. In this paper, the interaction of dependent variables, the normality of errors and the constant variance of errors are tested. All of these assumptions hold for an effective ANOVA test on the collected data and models (Eq. 3, 4, 5, and 6). Hence, we can conduct ANOVA tests, whose result will be shown in the next section.

Table 1: Statistical Findings of Effect on Delay: Jitter and Packet Loss

Jitter	**Df**	**Sum Sq**	**Mean Sq**	**F value**	**Pr(<F)**
Jitter	9	2.30	0.255	29.97	<2e-16
App	4	49.02	12.255	1437.63	<2e-16
Residuals	905	7.71	0.009	–	–
Packet Loss	**Df**	**Sum Sq**	**Mean Sq**	**F value**	**Pr(<F)**
Packet Loss	4	1.87	0.467	60.17	<2e-16
App	4	40.04	10.009	1290.49	<2e-16
Residuals	535	4.15	0.008	–	–

Table 2: Statistical Findings of Effect on Accuracy: Jitter and Packet Loss

Jitter	**Df**	**Sum Sq**	**Mean Sq**	**F value**	**Pr(<F)**
Jitter	9	29.16	3.24	62.84	<2e-16
App	2	23.94	11.972	232.21	<2e-16
Residuals	920	47.42	0.052	–	–
Packet Loss	**Df**	**Sum Sq**	**Mean Sq**	**F value**	**Pr(<F)**
Packet Loss	4	0.7102	0.1775	53.52	<2e-16
App	2	1.0846	0.5423	163.49	<2e-16
Residuals	299	0.9918	0.0033	–	–

6 ANOVA: Results and Conclusions

As can be seen in Table 1 there is conclusive evidence that delay is affected by both jitter and packet loss. More specifically, the f-values of jitter and application are 29.97 (df. = 9, p-value = 2e-16) and 1437.63 (df. = 4, p-value = 2e-16) while the f-values of packet loss and application are 60.17 (df. = 4, p-value = 2e-16) and 1290.47 (df. = 4, p-value = 2e-16), respectively. The underlying reasons are as follows. As we mentioned before, jitter causes packets to arrive out of order and TCP needs to reorder packets before delivering them to the application layer. TCP also re-transmits lost packets. Both packet loss and jitter reduce the voice stream quality and this affects the performance of the speech recognition. On the other hand, the application affects the delay more seriously. The f-values of application for jitter and packet loss are 1437.63 and 1290.47, respectively. In other words, Siri causes much more delay than GSR. This is because Siri generates accurate transcription by starting the speech recognition process just after receiving the whole voice. That means Siri needs to receive the whole stream before starting to generate the text. That increases the delay in processing the whole text and accounts for the majority of total delay. GSR, on the other hand, keeps the result accurate by interaction between the transport and application layers and so it offers less delay even under high values of packet loss and jitter, compared to Siri.

As can be seen in Table 2, there is conclusive evidence that accuracy is affected by both jitter (p-value = 2e-16, f-value =62.84 on 9 df.) and packet loss (p-value = 2e-16 , f-value =53.52 on 4 df.). More specifically, the f-values of jitter and application are 62.84 (df. = 9, p-value = 2e-16) and 232.21 (df. = 4, p-value = 2e-16) while the f-values of packet loss and application are 53.52 (df. = 4, p-value = 2e-16) and 163.49 (df. = 4, p-value = 2e-16), respectively. Interestingly, for accuracy, the impact of jitter and packet loss begin is greater than that for delay. However, their impacts are still less than application.

7 Threats to Validity

7.1 Conclusion Validity

Conclusion validity makes sure that there is a statistical relationship between the experiment and results, with a given significance [30]. A perfect experiment would be conducted in randomly selected locations around the world and using randomly selected Internet providers. The experiment should repeat many times in each location. Our testbeds for Apple Siri and Google Speech Recognition were limited to a campus network, thus limiting the statistical strength of the results. Selecting the location and Internet provider randomly, as well as increasing the number of sites can increase the conclusion validity.

7.2 Internal Validity

Internal Validity refers to the causal effects between independent and dependent variables, and for any relationship to exist, we should make sure that it is not as a result of a factor that there is no control over or that it has not been measured [30]. One of the possible threats to internal validity is the hardware limitations of the devices running GSR and Siri. More specifically, the processing speed of memory and CPU will affect the processing of data streams in a PC. Another possible threat is the status of the PC. For example, when the OS is busy, it does not have enough time to respond to the interruptions generated from GSR or Siri, hence generating and thus affecting delay.

7.3 Construct Validity

Construct validity refers to the relationship between theory and study. Experiments need to be set up such that to the highest degree possible, they are representative of the theory under test. The experiments reflect the construct of cause and results reflect the construct of effects well [30]. Since the delay generated by the Internet (e.g., router, DNS, etc.) is complicated and unpredictable, it is hard to say the extent to which packet loss and jitter impact delay. Also, the transportation and routing layers employ self-adaptive mechanisms to adjust the performance of specific applications, e.g., GSR and Siri. In the end, both the jitter and the packet loss are generated by a specific program (i.e., simulated), rather than real network conditions. It is hard to know whether the simulated impact has the same effects of real jitter or packet loss.

7.4 External Validity

The external validity is all about generalization. Can we generalize the result of the treatment outside the scope of our study in case of a causal relationship between cause and the construct [30]? All of the experiments were conducted in our lab and through our campus network. It is likely that the configuration of our campus network is different from other networks, such as firewalls and TCP/UDP controls. Hence, the conclusion obtained from the experiment cannot be generalized to common network environments. In addition, the available bandwidth of different regions in United States is different. It is possible that this diversity affects the conclusion that it cannot be applied to the other regions in United States. Finally, the sample is small (the evaluation is run on one desktop in a laboratory setting). A larger scale experiment running on more desktops, as well as laptops and smart phones, will lessen external threats.

8 Conclusions and Future Work

We designed and implemented experimental evaluations of Siri and GSR, and Dragon. Using experiment data, we designed four models to evaluate the effects of jitter and packet loss separately. After conducting ANOVA tests for each experiment, we found that the effects of packet loss and jitter on delay are statistically significant but the impact is not important compared to the one that comes from the application, because from the tables we can see that the application generated most of the impact. In addition, we found that GSR performs better than Siri in respect to delay. Results from the Dragon testbeds shows that the effects of packet loss and jitter on delay and also accuracy are statistically significant but the impact is not important compared to the one that comes from the application.

Statistical findings of effect of jitter and loss on the accuracy and delay show that the application generated most of the impact.

Delay of all applications is affected by packet loss and jitter. In order to design and implement real-time cloud speech recognition applications for more critical tasks, there should be mechanisms to measure loss/jitter tolerant systems. Network coding is a possible solution to reduce the effect of packet loss and jitter [8, 24, 26, 27]. Using TCP keeps these applications accurate under packet loss and jitter values, but as we saw in our results, it affects the roundtrip delay. By using UDP and network coding, we can keep the system accurate under different values of jitter and packet loss while we reduce the resulting delay. Future cloud based speech recognition applications that use cellular networks are still required to overcome this problem; which is due to the presence of jitter from packet transmission over different paths.

This experiment can also be extended by running Siri and GSR over different cellular networks, and adding the celluar data provider as another blocking variable.

Running the experimental setup over a wide geographical range of clients and also using different cellular data providers can result in more accurate results. Considering clients with a diversity of hardware and software configurations can be another extension for this research.

Acknowledgements

We thank the National Science Foundation for supporting this work through grant NSF CSR-1527097.

References

[1] *Nuance Communications*. `http://www.nuance.com/news/pressreleases/2009/20091005_ecopy.asp`.

[2] *Nuance Technologies*. `http://research.nuance.com/category/speech-recognition/`.

[3] *Siri*. `https://support.apple.com/en-us/ht4992`.

[4] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 157–168, New York, NY, USA, 2011. ACM.

[5] Saamer Akhshabi, Sethumadhavan Narayanaswamy, Ali C Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptive video players over http. *Signal Processing: Image Communication*, 27(4):271–287, 2012.

[6] Jay Beale Angela Orebaugh, Gilbert Ramirez and Joshua Wright. Wireshark and ethereal network protocol analyzer toolkit. *Syngress Media Inc*, 2007.

[7] Apple. *iOS: Multipath TCP Support in iOS 7*, 2014. `http://engineering.purdue.edu/~mark/puthesis`.

[8] Mehdi Assefi, Mike P. Wittie, and Allan Knight. Impact of network performance on cloud speech recognition. *ICCCN*, IEEE. Aug. 2015.

[9] Mehdi Assefi, Mike P. Wittie, Guangchi Liu, and Clemente Izurieta. An experimental evaluation of apple siri and google speech recognition. *SEDE*, ISCA. Oct. 2015.

[10] Ł Budzisz, Rade Stanojević, Arieh Schlote, Fred Baker, and Robert Shorten. On the fair coexistence of loss-and delay-based tcp. *IEEE/ACM Transactions on Networking (TON)*, 19(6):1811–1824, 2011.

[11] Kuan-Ta Chen, Cheng-Chun Tu, and Wei-Cheng Xiao. Oneclick: A framework for measuring network quality of experience. In *INFOCOM 2009, IEEE*, pages 702–710. IEEE, 2009.

[12] Kuan-Ta Chen, Chen-Chi Wu, Yu-Chun Chang, and Chin-Laung Lei. A Crowdsourceable QoE Evaluation Framework for Multimedia Content. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 491–500. ACM, 2009.

[13] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *ACM IMC*, Oct. 2013.

[14] Luca De Cicco, Gaetano Carlucci, and Saverio Mascolo. Experimental investigation of the google congestion control for real-time flows. In *SIGCOMM workshop on Future human-centric multimedia networking*, Aug. 2013.

[15] Luca De Cicco and Saverio Mascolo. *An experimental investigation of the Akamai adaptive video streaming*. Springer, 2010.

[16] M Eittenberger, Todor Mladenov, and Udo R Krieger. Raptor codes for p2p streaming. In *Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2012.

[17] Google. *Performing speech recognition over a network and using speech recognition results*. `http://www.google.com/patents/US8335687`.

[18] David A Hayes and Grenville Armitage. Improved coexistence and loss tolerance for delay based tcp congestion control. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 24–31. IEEE, 2010.

[19] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *ACM Mobile systems, applications, and services*, Jun. 2010.

[20] Te-Yuan Huang, Kuan-Ta Chen, and Polly Huang. Tuning skype's redundancy control algorithm for user satisfaction. In *INFOCOM, Apr. 2009*.

[21] Te-Yuan Huang, Polly Huang, Kuan-Ta Chen, and Po-Jung Wang. Could skype be more satisfying? a qoe-centric study of the fec mechanism in an internet-scale voip system. *Network, IEEE*, 24(2):42–48, 2010.

[22] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, pages 662–665, 2013.

[23] David JC MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, Dec. 2005.

[24] Hyung Rai Oh and Hwangjun Song. Mesh-pull-based p2p video streaming system using fountain codes. In *Computer Communications and Networks (ICCCN)*, Jul. 2011.

[25] J. Roskind. *QUIC: Design Document and Specification*, Dec. 2013. `https://docs.google.com/a/chromium.org/document/d/1RNHkxVvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34`.

[26] Guillaume Smith, P Tournoux, Roksana Boreli, Jérôme Lacan, and Emmanuel Lochin. On the limit of fountain mdc codes for video peer-to-peer networks. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Jun. 2012.

[27] Dejan Vukobratovic, Vladimir Stankovic, Dino Sejdinovic, Lina Stankovic, and Zixiang Xiong. Scalable video multicast using expanding window fountain codes. *IEEE Transactions on Multimedia*, 11(6):1094–1104, Oct. 2009.

[28] Stefan Winkler and Ruth Campos. Video quality evaluation for internet streaming applications. In *Electronic Imaging 2003*, pages 104–115. International Society for Optics and Photonics.

[29] Stefan Winkler and Frédéric Dufaux. Video quality evaluation for mobile streaming applications. In *Visual Communications and Image Processing 2003*, pages 593–603. International Society for Optics and Photonics, 2003.

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in Software Engineering. pages 102–104. Springer Science & Business Media, 2012.

[31] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. Video telephony for end-consumers: measurement study of google+, ichat, and skype. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 371–384. ACM, 2012.

[32] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, Jun. 2007.

Mehdi Assefi is a graduate student in the Computer Science department at Montana State University. Born in Mashhad, Iran, his research interests include cloud computing, network performance evaluation, and real-time streaming. Mehdi Assefi has approximately 11 years teaching and research experience.

Guangchi Liu is currently a Ph.D candidate in the department of computer science, Montana State University, Bozeman, MT, USA. He received his B.E. in biomedical engineering, and M.E. in electrical and computer engineering from Southeast University, China in 2009 and 2012, respectively. His research interests include Internet of things, trust assessment, social network, and wireless sensor network.

Dr. Mike P. Wittie is a RightNow Technologies Assistant Professor and a co-director of the Networks+Algorithms Lab at the Montana State University Computer Science Department since Fall 2011. His research interests focus on latency reduction, network measurement, and content delivery in wide-area networks. He received his PhD in Computer Science from the Computer Science Department at the University of California, Santa Barbara, and MSE in Computer Science and BA in Cognitive Science from the University of Pennsylvania. He worked professionally on military datalink integration for Anzus Inc. (now Rockwell Collins).

Dr. Clemente Izurieta is an Assistant Professor in the Computer Science department at Montana State University and holds a PhD from Colorado State University. Born in Santiago, Chile, his research interests include empirical software engineering, design and architecture of software systems, the measurement of software quality, and technical debt. Dr. Izurieta has approximately 16 years experience working for various RD labs at Hewlett Packard and Intel Corporation and currently directs the Software Engineering Laboratories (SEL) at Montana State. SEL has funding from NSF, DoD, and the State of Montana and supports 3 PhD and 6 MS students.

Test Suite Selection in JUnit Testing Environment Based on Software Metrics

Shadi Banitaan[*], Kevin Daimi[*]
University of Detroit Mercy, Detroit, Michigan USA

Mohammed Akour[‡]
Yarmouk University, JORDAN

Yujun Wang[*]
University of Detroit Mercy, Detroit, Michigan USA

Abstract

As software systems evolve, test suites become very large. The size of test suites has a direct impact on both the cost and the effort of software testing. To reduce the cost of software testing, researchers have proposed different test case selection techniques. Test case selection techniques aim to identify test cases that are not needed to run based on some criteria. In this work, we propose an approach for test case selection using software metrics. We examine the ability of several complexity and size metrics to find the most complex and error-prone classes. Testers can then run test cases that are associated with the complex classes only. We focus our experiments on systems written in Java and tested with the JUnit testing framework. The results reveal that the proposed approach significantly reduces the number of test cases needed while detecting most of seeded errors.

Key Words: Test case selection, software metrics, unit testing, software testing.

1 Introduction

Software is an essential part in several of the devices and the systems that we use in our daily life. It is very crucial for software to be very reliable, especially the one that is attached with mission critical devices such as pacemakers. Software failure (i.e., incorrect software behavior) may lead to risky harms such as aircraft crashes. Moreover, software failure may have a direct impact on economy. Therefore, software under development should be evaluated first prior to delivery to customers. Software testing is the main method that is usually used to test and evaluate software under development [13]. The primary goal of testing is to find defects before customers find them out [13].

Unit testing is the first phase of testing. The goal of unit testing is to ensure that all software units are working correctly in isolation. In Object oriented (OO) software, the class is the smallest testable unit and class testing is determined by the methods and the behavior of the class [19]. To apply unit testing, both white box and black testing techniques can be used. White box testing techniques involve access to software code while black box testing techniques do not involve access to software code. In white box unit testing, software developers write test cases to make sure that classes behave as intended. Unfortunately, it is not possible to comprehensively test large software products. In addition, Software developers do not find enough time to conduct unit testing. Also, unit testing is very costly. One of the solutions to the aforementioned problems is to focus the unit testing on the most complex and error-prone classes.

Complex classes can be identified using software metrics. Software metrics are quantitative measures of some properties of software. They are extensively used to control the software development and measure the quality of software products (e.g., integrity and maintainability). Chidamber and Kemerer [11] proposed six metrics of OO design such as lack of cohesion in methods (LCOM) and number of children (NOC). Whitmire [22] explained measurable characteristics of an OO design [19]. Some of these characteristics are size, complexity, coupling, and cohesion. Binder [8] proposed several design metrics that contribute to a systems testability. In this work, we identify complex classes using both size and complexity metrics. Identifying complex classes helps the testing team to focus their testing effort on just part of the classes to save time and resources. The premise is that errors are most likely to occur on the more complex classes. To summarize, this paper makes the following contributions:

- Propose an approach for test case selection using complexity and size metrics.
- Conduct an experimental study on two Java applications.
- Evaluate the proposed approach using the error seeding technique.
- Compare the proposed work with one of the state-of-the-art approaches.

[*] Computer Science and Software Engineering. Email: (banitash, daimikj, wangyu6)@udmercy.edu.
[‡] Computer Information Systems. Email: mohammed.akour @yu.edu.jo

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 describes the proposed approach. The experimental evaluation and discussion are presented in Section 4. Section 5 concludes the paper.

2 Related Work

Software testing is very costly. Therefore, reducing the cost of testing is a big challenge. Three main branches have been proposed in literature to reduce the cost of testing that include test prioritization [14, 21, 24], test selection [6, 10, 17], and test minimization [9, 16, 20]. Test prioritization aims to rank test cases so that test cases that are more effective according to such criteria will be executed first to maximize fault detection. Test selection aims to identify test cases that are not needed to run on the new version of the software. Test minimization aims to remove redundant test cases based on some criteria in order to reduce the number of tests to run. Our approach belongs to the test selection category.

Several researchers used the optimization search techniques to solve the test selection problem [7, 12, 18, 23]. Mansour and El-Fakih [18] used simulated annealing and genetic algorithms for regression test case selection. Their results revealed that simulated annealing and genetic algorithms can find the optimal or near-optimal number of retests within a reasonable time. Yoo and Harman [23] presented a study that investigated the effectiveness of three algorithms for Pareto efficient multi objective test case selection. Their results showed that greedy algorithms are not always Pareto efficient in the multi-objective paradigm.

Recently, some approaches were proposed to reduce the cost of testing by reducing the number of developed test cases using software metrics. Bouchaib [15] used a set of complexity metrics at class, method, and statement level to target test cases. The experimental results showed that the developed test cases using the proposed approach detected 100% of seeded errors and at least 60% of mutants. Banitaan et al. [5] proposed an approach to select the test focus in integration testing. Their approach used a combination of dependency and complexity metrics to give a weight for each method-pair connection. After that, the approach predicted the number of test cases needed to test each connection. They evaluated their approach using error seeding technique. Their experimental results on several Java applications showed that the small number of developed test cases (half the number of method-pair connections) detected at least 80% of integration errors. In this work, we investigate the use of several complexity and volume metrics to solve the test case selection problem.

3 The Proposed Approach

In this paper, we propose a simple approach to reduce the cost of unit testing using class-level software metrics. Figure 1 illustrates the activity diagram for the proposed approach. The first step is to calculate the metric from the source code. When calculate metric function completes, it produces the values of the calculated metrics. After that, Rank all Classes function ranks classes based on the metric values for all classes. Then, test cases associated with the top ranked 25% of classes will be selected in the first iteration. The next step is to run the selected test cases against the faulty versions of the application and the error detection rate is calculated. The process of selecting test cases is an iterative process. We stop if the selected test cases achieve 70% error detection rate. Otherwise, we select more test cases until the 70% error detection rate is achieved. For example, if the error detection rate using the 25% of the ranked classes is less than 70% then we compute the error detection rate using the test cases associated with the top 30% of the ranked classes. The iterative process continues by adding additional 5% of the test classes until the 70% error detection rate is achieved. All steps are applied and repeated using each of the selected metrics. It is noteworthy to mention that the aim is to detect a large percentage of errors using a small number of test cases. We use 70% as a threshold for the percentage of detected errors to stop the test selection process.

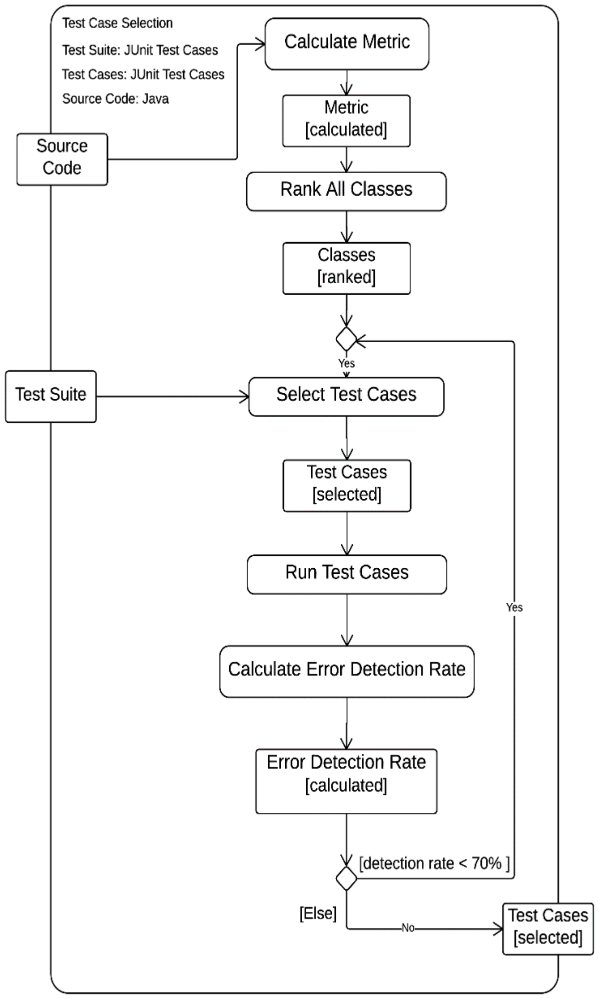


Figure 1: The proposed test selection approach

3.1 Metrics

In this section, we define the selected complexity and size metrics. We selected four complexity metrics; SumCyclomatic, AvgCyclomatic, MaxCyclomatic, and SumEssential. The cyclomatic complexity measures the number of independent paths through the source code. The higher the complexity the more difficult the program becomes to test. The essential measures the amount of unstructured code. We selected four size metrics; AvgLineCode, CountLine, CountDeclMethodAll, and CountDeclInstanceVariable. The idea of this selection is to investigate if complexity and size metrics can be used to identify the more error-prone classes. We use the Understand tool by SciTools [4] to calculate the complexity and size metrics. Table 1 presents the description of the selected metrics.

Table 1: Description of metrics

Metric	Description
SumCyclomatic	The sum of cyclomatic complexity of all nested functions or methods
AvgCyclomatic	The average cyclomatic complexity for all nested functions or methods
MaxCyclomatic	The maximum cyclomatic complexity of all nested functions or methods
SumEssential	The sum of essential complexity of all nested functions or methods
AvgLineCode	The average number of lines containing source code for all nested functions or methods
CountDeclMethodAll	The number of methods, including inherited ones
CountDeclInstanceVariable	The number of instance variables
CountLine	The number of all lines

3.2 Subjects

In this work, we focus our experiments on systems written in Java and tested with the JUnit testing framework. Therefore, we selected two open-source applications implemented in Java that have JUnit test suites. The first case study subject is Commons Math [1]. Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang. The second subject is Commons IO [2]. Commons IO is a library of utilities to assist with developing IO functionality. It includes six main areas: utility classes, input, output, filters, comparators, and file monitor.

Table 2 shows a summary of the selected systems where Classes denotes the number of classes, Methods denotes the number of methods, and Test Cases denotes the number of test cases.

Table 2: Summary of subjects

Application	Classes	Methods	Test Cases
Commons Math	161	1728	1578
Commons IO	97	884	1213

4 Evaluation

In this section, the performance of the proposed approach is evaluated with respect to the error detection capabilities. The two Java applications do not have errors. Therefore, errors should be injected into the applications. Researchers usually use two approaches to inject errors: mutation and error seeding. The mutation approach produces a large number of errors but these errors may not be descriptive of actual errors [14]. Error seeding cannot create a large number of errors practically and efficiently, but it can generate actual errors [14]. Therefore, the error seeding technique is used in this work. Errors are placed in the source code by a graduate student. The error seeder injects different types of errors based on his experience with real programs. The error seeder injects 40 errors in each application. The error detection rate, the savings, and the score are calculated. The error detection rate is calculated by dividing the number of detected seeded errors by the total number of seeded errors. The savings metric measures the percentage of saving of test cases. It can be calculated using the following formula.

$$Savings = \frac{T - T_{select}}{T} \times 100$$

Where T is the total number of test cases available for the application while T_{select} is the number of selected test cases. The formula assumes that all test cases have uniform costs. The score metric takes both error detection rate and number of test cases in consideration. The score is computed as follows:

$$Score = \frac{Error\ Detection\ Rate}{T_{Select}}$$

Table 3 shows the results of applying the proposed approach on the two Java applications. It reports the number of selected test cases based on the eight class-level metrics, the savings, the score, the error detection rate, and the percentage of classes that are tested. The results show that the SumCyclomatic metric is the only metric that gives an error detection rate higher than or equal to 70% on the first iteration (i.e., by testing the highly ranked 25% of classes). In terms of savings, the SumCyclomatic metric gives the best results while the MaxCyclomatic metric gives the worst results for the Commons Math application. For Commons IO, the maximum percentage of savings is obtained by using the AvgLineCode metric while the minimum percentage of savings is obtained by using the CountDeclInstanceVariable metric. In terms of score, the highest score is obtained by using SumCyclomatic while the lowest score is obtained by using MaxCyclomatic for the Commons Math application. For the Commons IO application, the highest score is obtained by using AvgLineCode while the lowest score is obtained by using CountDeclInstanceVariable. The results of the Commons IO application also show that the best error detection rate is obtained by using AvgLineCode and CountLine. However, AvgLineCode has a higher score than CountLine because it produces much more savings. As a

Table 3: Results of the proposed approach

Application	Metric	# Of Test Cases	Savings	Score	Detection Rate	% of Tested Classes
Commons Math	SumCyclomatic	490	**68.95**	**0.143**	70%	25%
	AvgCyclomatic	586	62.86	0.119	70%	50%
	MaxCyclomatic	713	54.82	0.105	75%	40%
	SumEssential	589	62.67	0.127	75%	30%
	AvgLineCode	544	65.53	0.129	70%	30%
	CountDeclMethodAll	554	64.89	0.126	70%	30%
	CountDeclInstanceVariable	652	58.68	0.110	72%	50%
	CountLine	618	60.84	0.118	73%	30%
Commons IO	SumCyclomatic	562	53.67	0.114	75%	25%
	AvgCyclomatic	840	30.75	0.087	73%	40%
	MaxCyclomatic	653	46.17	0.112	73%	35%
	SumEssential	665	45.18	0.105	70%	30%
	AvgLineCode	465	**61.67**	**0.172**	80%	30%
	CountDeclMethodAll	490	59.61	0.143	70%	30%
	CountDeclInstanceVariable	923	23.91	0.076	70%	50%
	CountLine	669	44.85	0.120	80%	30%

CountLine. However, AvgLineCode has a higher score than CountLine because it produces much more savings. As a result, the proposed approach is effective in detecting at least 70% of seeded errors by testing the highly complex classes ranked by the selected complexity and size metrics. The results indicate that the highly ranked classes comprehend most of the errors. Moreover, the proposed approach reduces the number of test cases needed to detect most of the errors (i.e., high percentage of savings). We also compared our work with the method coverage approach where test suites are ranked based on the number of methods they cover. The EclEmma code coverage tool [3] is used to collect the method coverage of the test suites. Figure 2 shows the comparison in terms of savings and score. For Commons Math, the selected metrics outperform the coverage based approach in terms of savings. For Commons IO, all selected metrics except CountDeclInstanceVariable outperform the coverage based approach in terms of savings. For Commons Math, the score of the selected metrics outperforms the coverage based approach. For Commons IO, the score of all selected metrics except CountDeclInstanceVariable outperforms the coverage based approach. Based on the results of the comparisons, we notice that using the complexity and size metrics give better results as compared with the coverage based approach.

5 Conclusion

In this paper, we introduced an approach for test case selection. We examined the ability of several complexity and size metrics to detect the most complex and error-prone object-oriented classes. After identifying the most complex and error-prone classes, testers can then execute test suites that are associated with those classes only. We applied the proposed approach on two open-source Java applications, namely Commons Math and Commons IO. Error seeding technique is used to inject errors. The results revealed that the proposed approach is effective in detecting at least 70% of seeded errors by focusing the testing on the most complex classes. As a result, the proposed approach is feasible in reducing the number of test cases needed.

References

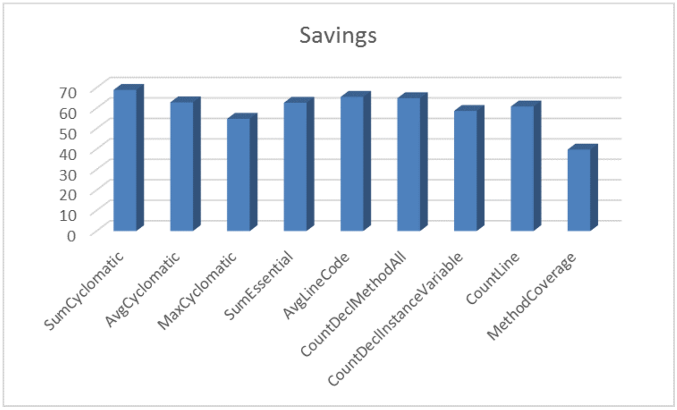
[1] *The Apache Commons Mathematics Library*. http://commons.apache.org/math/.

[2] *Commons io*. http://commons.apache.org/io/.

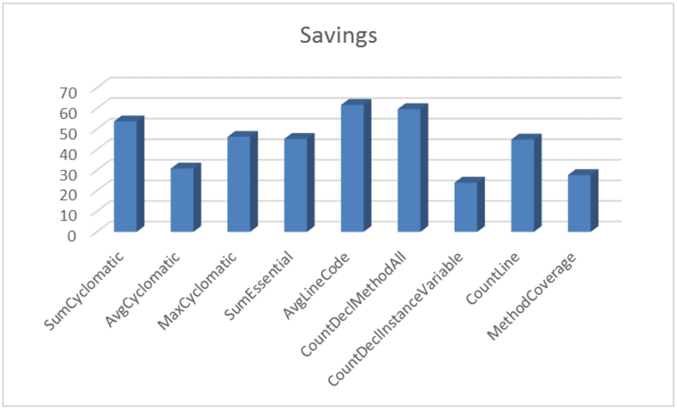
[3] *Eclemma: Java Code Coverage for Eclipse*. http://eclemma.org/.

[4] *Understand your code*. http://scitools.com/.

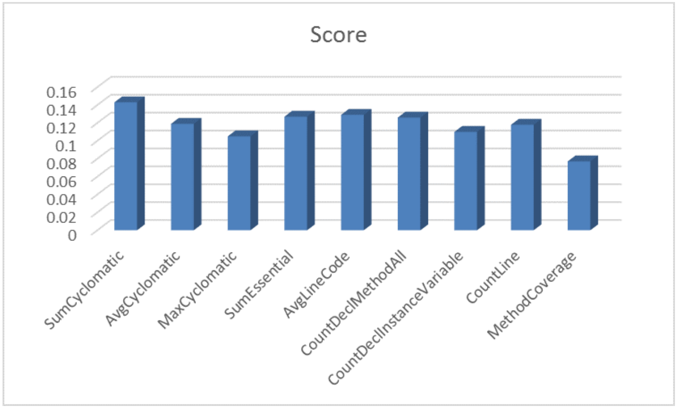
[5] Shadi Banitaan, Mamdouh Alenezi, Kendall Nygard, and Kenneth Magel, "Towards Test Focus Selection for



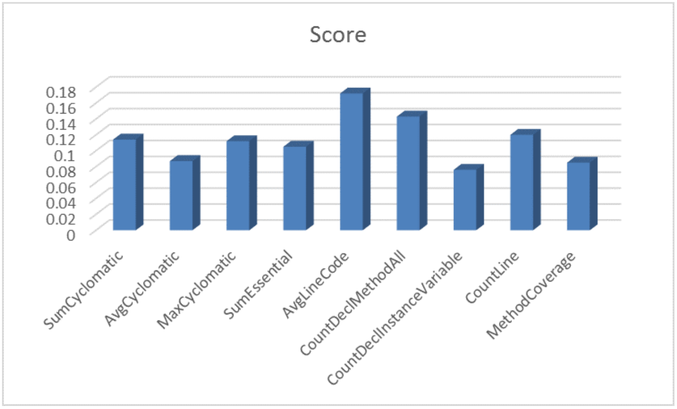
(a)Savings of Commons Math



(b) Savings of Commons IO



(c) Score of Commons Math



(d) Score of Commons IO

Figure 2: The results of comparing the proposed work with the coverage based approach

Integration Testing using Method Level Software Metrics," *Proceedings of the 10th International Conference on Information Technology: New Generations*, pp. 343-348, April 2013.

[6] Shadi Banitaan, Kevin Daimi, Yujun Wang, and Mohammed Akour, "Test Case Selection using Software Complexity and Volume Metrics," 24th International Conference on Software Engineering and Data Engineering (SEDE), ISCA, 2015.

[7] Kevin Barltrop, Brad Clement, Greg Horvath, and Cin-Young Lee, "Automated Test Case Selection for Flight Systems using Genetic Algorithms," *Proceedings of the AIAA Infotech@ Aerospace Conference*, pp. 1-8, 2010.

[8] Robert V Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM*, 37(9):87-101, 1994.

[9] Dale Blue, Itai Segall, Rachel Tzoref-Brill, and Aviad Zlotnick, "Interaction-Based Test-Suite Minimization," *Proceedings of the 2013 International Conference on Software Engineering*, pp. 182-191. IEEE Press, 2013.

[10] L. C. Briand, Y. Labiche, and S. He, "Automating Regression Test Selection Based on Uml Designs," *Inf. Softw. Technol.*, 51:16-30, Jan 2009.

[11] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, 20:476-493, 1994.

[12] Luciano S de Souza, Ricardo Bastos Cavalcante Prudencio, and Fl_avia de Almeida Barros, "A Constrained Particle Swarm Optimization Approach for Test Case Selection," *SEKE*, pp. 259-264, 2010.

[13] Srinivasan Desikan and Gopalaswamy Ramesh, *Software Testing: Principles and Practices*, Pearson Education India, 2012.

[14] Hyunsook Do, Gregg Rothermel, and Alex Kinneer, "Prioritizing Unit Test Cases: An Empirical Assessment and Cost-Benefits Analysis," *Empirical Software Engineering*, 11(1):33-70, 2006.

[15] Bouchaib Falah, *Test Case Selection Based on a Spectrum of Complexity Metrics*, PhD Dissertation, North Dakota State University, 2011.

[16] H. Y. Hsu and A. Orso, "Mints: A General Framework and Tool for Supporting Test-Suite Minimization,"

IEEE 31st International Conference on Software Engineering, 2009, ICSE 2009, *IEEE*, pp. 419-429, 2009.

[17] Y. Ledru, G. Vega, T. Triki, and L. Bousquet, "Test Suite Selection Based On Traceability AnnoTations," *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 342-345. ACM, 2012.

[18] Nashat Mansour and Khalid El-Fakih, "Simulated Annealing and Genetic Algorithms for Optimal Regression Testing," *Journal of Software Maintenance*, 11(1):19-34, 1999.

[19] Roger Pressman, *Software Engineering: A Practitioner's Approach*, 7th Edition, McGraw-Hill, Inc., New York, NY, USA, 2010.

[20] Sriraman Tallam and Neelam Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *Proceedings of the 6th "ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pp. 35-42, New York, NY, USA, 2005.

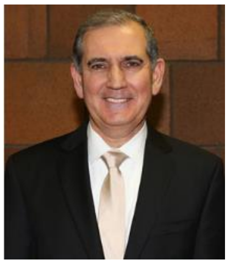
[21] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein, "Static Test Case Prioritization using Topic Models," *Empirical Software Engineering*, 19(1):182-212, 2014.

[22] Scott A Whitmire, *Object Oriented Design Measurement*, John Wiley & Sons, Inc., 1997.

[23] Shin Yoo and Mark Harman, "Pareto Efficient Multi-Objective Test Case Selection," Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, pp. 140-150, 2007.

[24] Z. Zhang, Y. Mu, and Y. Tian, Test Case Prioritization for Regression Testing Based on Function Call Path," 2012 Fourth International Conference on Computational and Information Sciences (ICCIS), IEEE, pp. 1372-1375, 2012.

Shadi Banitaan is currently an Assistant Professor at the Mathematics, Computer Science, and Software Engineering department at the University of Detroit Mercy. His research interests include data mining, mining software repositories, software testing, and software engineering. He is a member of the Association for Computing Machinery (ACM), a member of the Institute of Electrical and Electronic Engineers (IEEE), and a member of the IEEE Computer Society. He received a B.S. degree in Computer Science from Yarmouk University, an M.S. degree in Computer and Information Sciences from Yarmouk University, and a Ph.D. degree in Computer Science from North Dakota State University. He worked as a lecturer at the University of Nizwa from 2004 to 2009. He joined the University of Detroit Mercy in 2013.



Kevin Daimi received the B.S. degree from the University of Baghdad, Iraq, in 1971, the M.S. and Ph.D. degrees from the University of Cranfield, Bedford, England, in 1980 and 1983 respectively. He is currently a Full Professor of Computer Science and Software Engineering, and Director of Computer Science and Software Engineering Programs at the University of Detroit Mercy (UDM), Detroit, Michigan. His research interests include computer and network security, software engineering, computer science and software engineering education, and data mining. He is the recipient of the 2013 Faculty Excellence Award at UDM. He has worked as programmer, senior systems analyst, computer manager, computer consultant, director of computer center, and head of computer science department. He is a fellow of the British Computer Society (BCS), a senior member of the Association for Computing Machinery (ACM), and a senior member of the Institute of Electrical and Electronics Engineers (IEEE).

Mohammed Akour is an Assistant Professor in the Department of Computer Information System at the Yarmouk University (YU). He received his Bachelor (2006) and Master (2008) degree from Yarmouk University in Computer Information System with honor. He joined YU as a Lecturer in August 2008 after graduating with his Master in Computer Information System. In August 2009, he left YU to pursue his PhD in Software Engineering at the North Dakota State University (NDSU). He joined YU again in April 2012 after graduating with his PhD in Software Engineering from NDSU with honor. He serves as a reviewer for several conferences and Journals. He has participated as a co-chair for several conferences.

Yujun Wang is currently a graduate student in the Mathematics, Computer Science, and Software Engineering department at the University of Detroit Mercy. His research interests include component based software engineering and software development life cycle. He received a Bachelor's Degree ('08) in Computer Science from Neusoft University, China.

Assessing Software Defects Using Nano-Patterns Detection

Ajay K. Deo*, Zadia Codabux*, Kazi Zakia Sultana*, and Byron J. Williams*
Mississippi State University, Starkville, MS, 39762, USA

Abstract

Improving software quality and reliability is important for sustained software development efforts. Defect tracking has traditionally been used to measure quality throughout the software lifecycle. Various techniques have been used to identify code fragments containing defects. Locating code smells is one increasingly popular approach to identifying vulnerable code, but techniques relying on code smell identification have drawbacks. These techniques use ambiguous thresholds of traditional metrics to determine where the code smells exist in the code. There is a need for a more reliable, exact approaches that are consistent across software systems. Traceable patterns, such as method-level nano-patterns provide a direct binary assessment of code. This study evaluates software defects using nano-patterns by demonstrating that certain categories of nano-patterns are more defect-prone than others. We studied three open source systems from the Apache Software Foundation and found that ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalReader, and LocalWriter nano-patterns are more defect-prone than others. Apart from assessing software defects, we expect this new finding will contribute to further research on other applications of nano-patterns and improve coding practices.

Key Words: Software patterns, traceable patterns, nano-patterns, defect detection, software quality.

1 Introduction

Software systems are becoming more complex with an increasing project lifespan. Software quality has emerged as an important aspect of software development. However, software quality is an elusive concept which cannot be physically measured [12]. One simple measure of quality is the number of defects contained in a release. These defects directly impact a product's quality and overall customer satisfaction. Achieving zero software defects and zero software maintenance cost would be an unrealistic expectation for most software systems. Minimizing defects represents a tangible improvement in software development. Therefore, software developers strive to minimize the number of defects to improve software quality and lower development costs. Releasing software containing defects can damage the reputation of the software development company resulting in customers' loss of confidence in the product. This research focuses on understanding software quality using traceable software patterns to assess defects.

Software maintenance is a never-ending process. Changes made during a maintenance phase are often critical and can be expensive. Maintenance cost outweighs other development costs by a substantial margin [5]. Maintenance activities include defect fixes, enhancements, optimizations, refactoring, etc. Some of these activities (e.g., adding new functionality) are planned in advance. Maintenance due to environment changes is inevitable. However, other activities (e.g., defect fixing and optimization) are mostly the result of poor software quality. Cost incurred engaging in these maintenance activities can be minimized.

Changes are inevitable in software development. Rigorous testing of the entire code base for every single change is impractical as it can be tedious and expensive. To locate the defect-prone areas of software, developers use static analysis tools to identify code smells and other development irregularities. Software metrics are often used to determine the quality of the software. However, many metrics are vaguely defined or can have multiple definitions. Some metrics are not reliable in the sense they do not accurately represent the current state of the software system [6]. For example, the definition of number Lines of Code (LOC) can be with or without comments and with or without spaces. The same code written in two different styles may differ in LOC. LOC is difficult to use as it is not practical to restrict the developer within a certain number of LOC as a way to ensure software quality [12]. Therefore, using these metrics as quality indicators is often ambiguous.

Fowler et al. [15] came up with a concept of code smells for identifying bad code. "A code smell is a surface indication that usually corresponds to a deeper problem in the system" that may slow down software development or increase the risk of bugs [15]. Detecting code smells is an effective way of improving code quality by identifying bad components and identifying opportunities for refactoring. Code smells are detected by using a combination of threshold values of standard software metrics [14]. However, a lack of consensus on the threshold values of these metrics results in discrepancies

* Department of Computer Science and Engineering. Email akd175@msstate.edu, zc130@msstate.edu, ks2190@msstate.edu, williams@cse.msstate.edu.

in the detection of code smells. Changing these threshold values has a great impact on the number and accuracy of detected smells [14]. In consequence, the use of code smells as a pointer to assess the likelihood of software defects with high accuracy is difficult.

Gil et al. [16] introduced the concept of traceable patterns. Traceable patterns are similar to design patterns but at a lower level of abstraction. These patterns are automatically (mechanically) recognized as they are tied to a specific language and are related to a single software element. Traceable patterns are of various types depending upon their level of abstraction. Class-level traceable patterns are called micro-patterns (MP), whereas method-level traceable patterns are called nano-patterns. Gil et al. [16] defined 24 micro-patterns organized into 8 categories. Similarly, Singer et al. [29] listed 17 fundamental nano-patterns organized into 4 groups.

Micro-patterns can be used to capture good or bad program elements (i.e., whether the code represents acceptable coding practices or not). They can help developers identify those classes that belong to those categories of micro-pattern that are fault-prone [11]. Micro-patterns are an important tool for assessing software quality by spotting bad program components. Similarly, good program elements can be highlighted and encouraged as they make up good programming practices.

This study uses nano-patterns to evaluate software defects in OO systems. We demonstrate that certain categories of nano-patterns are more fault-prone than others and provide guidelines for programming practices on the use of nano-patterns. To the best of our knowledge, no study has examined nano-patterns to assess software system defects. This research builds upon existing work using traceable patterns while introducing new ways to view these method-level patterns and their ability to locate potentially defective areas of code.

The rest of the document is structured as follows. Section 2 describes micro-patterns, nano-patterns, and code smells. Section 3 gives an overview of related work. Section 4 focuses on the methodology of this study. We present our results in Section 5. Section 6 provides insights on the results. Section 7 lists the threats to validity, and Section 8 concludes and outlines the future work.

2 Background

This section gives an overview of traceable patterns and describes the two types of traceable patterns: micro-pattern which are class level patterns and nano-patterns which are methods level patterns.

2.1 Traceable Patterns

Gil et al. [16] came up with the notion of traceable patterns for Java code. Traceable patterns are similar to design patterns. However, they can be mechanically (automatically) recognized and stand at lower level of abstractions than design patterns. Traceable patterns are tied to the implementation language and "can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components". They impose a condition on a single software module; for example, a class in which all fields is static, a method that only returns void, etc. The following conditions qualify a pattern as traceable [16]:

- Patterns should not occur at random
- Patterns must capture a non-trivial idiom of the programming language
- Patterns should serve a concrete purpose

Traceable patterns can be of various types depending upon their level of abstraction (e.g., micro-patterns, nano-patterns, etc.). Class level patterns are called micro-patterns (MP), whereas method level traceable patterns are called nano-patterns.

2.1.1 Micro-Patterns. Gil et al. [16] defined a catalog of 27 micro-patterns organized into 8 categories relating to various programming practices in Java. Figure 1 shows a global map of the catalog where 27 micro-patterns are placed in 8 categories. The X-axis in the figure corresponds to class behavior while the Y-axis in the figure corresponds to class state. Categories on the left are those patterns that are more restrictive than those on the right, and categories of patterns on the top are more restrictive than those at the bottom. It can be seen that some patterns are placed in more than one category. Their empirical study shows micro-patterns to be found consistently in abundance (about 75% of code) in the software systems examined. Authors claimed that micro-patterns can enhance design, code learning and reuse, training, and automation [16].

The beauty of micro-patterns is that they are mechanically recognized and therefore can be a good way to assess the quality of software. Two studies, one by Destefanis et al. [11] and other by Giulio et al. [9] were conducted to find the relationship between fault-proneness and micro-patterns. Destefanis found certain micro-patterns to be more fault-prone than others, while Giulio found classes that did not match any micro-patterns were more likely to be fault-prone. Fontana et al. [14] investigated the correlations between code smells and micro-patterns and found Data Class to have some relation with the Data Manager, Extender, Immutable and Function Objection micro-patterns. They also found co-location of Duplicate Code with Data Manager, Extender, Outliner and Sink micro-patterns. Though micro-patterns look like a good candidate to assess software's quality, the number of studies conducted so far is minimal.

2.1.2 Nano-Patterns. A nano-pattern is a method level traceable pattern. They are traceable patterns because, "they can be expressed as a simple formal condition on the attributes, types, name and body" of methods written in the Java programming language [16, 29]. Nano-patterns have following properties [29]:

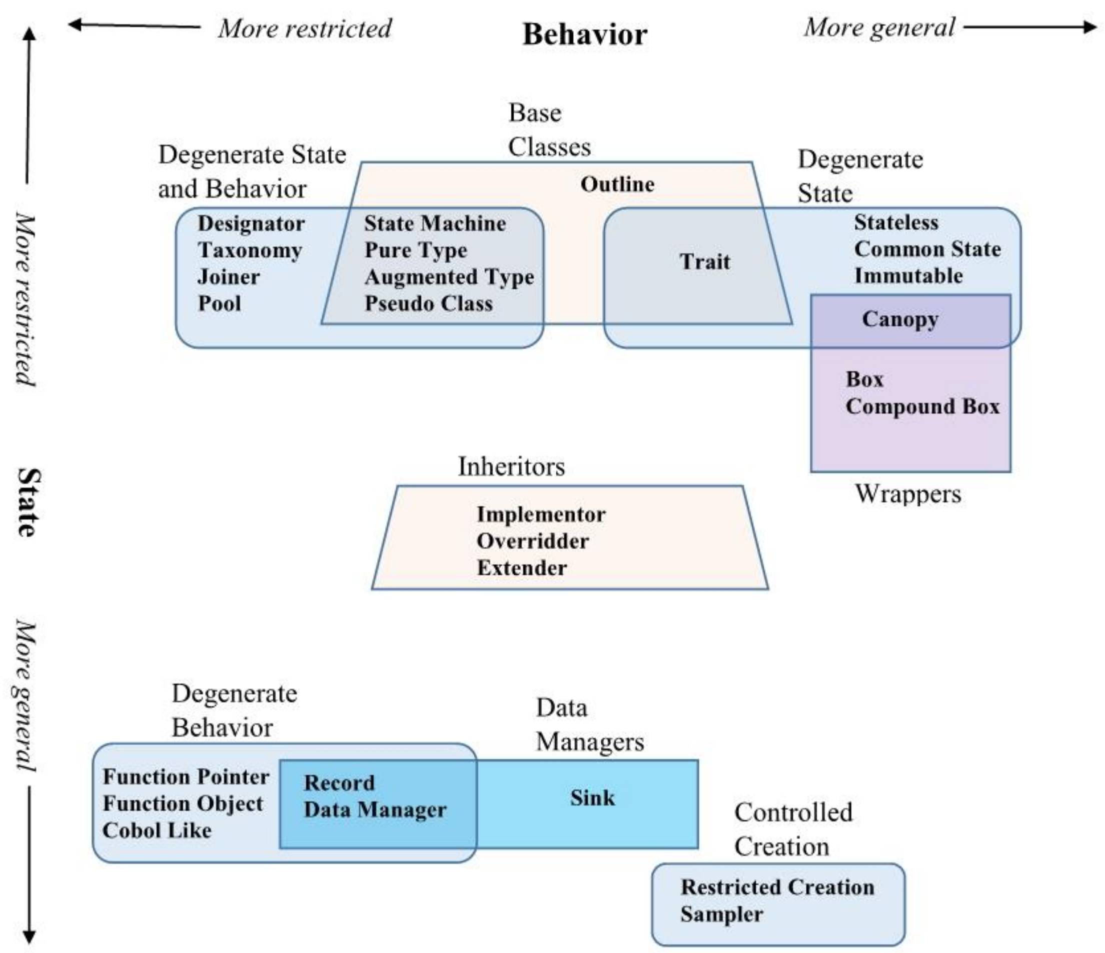


Figure 1: A map of the micro-patterns catalog [16]

Notes: "Rounded rectangles denote pattern categories in which state, behavior, or construction is degenerate, rectangles denote categories of patterns for containment, while trapezoids denote patterns used for inheritance" [16].

- Simple: Can be detected by manual inspection or the process can be easily automated
- Static: Should be detectable on bytecode analysis, without any program execution context
- Binary: Each property is either true or false

Høst et al. [18] initially came up with simple Java method attributes. Singer et al. [29] termed these attributes as fundamental nano-patterns. They also extended Høst et al.'s [18] attribute set to give a catalogue of 17 fundamental nano-patterns grouped into four intuitive categories as shown in Table 1. Boldface names in the table are for original patterns devised by Singer et al. [29]. All other patterns are derived from Høst et al.'s [18] catalogue. Fundamental nano-patterns can also be combined logically to derive composite nano-patterns as shown below where PureMethod composite nano-pattern is derived from FieldWriter, ArrayWriter, ObjectCreator, ArrayCreator, and Leaf fundamental nano-patterns [29].

PureMethod = ˥FieldWriter Λ˥ ArrayWriter Λ˥ ObjectCreator Λ˥ ArrayCreator Λ Leaf

Nano-patterns seem to have various potential applications. It can be used to find the similarity between methods, as similar methods should have similar patterns. They provide a good framework for quantitative analysis of large Java applications and are of great help for the techniques like data mining and clustering [29]. Combinations of nano-patterns with micro-patterns can be an aid to design, documentation and software comprehension [16]. Some combinations of low-level patterns can be potential indicator to high-level patterns. Therefore, nano-patterns can be utilized for detecting high level patterns like design patterns, code smells, etc., whose detection in general is acknowledged to be difficult. Like micro-patterns, it can be an important tool for assessing software quality. However, nano-patterns being a relatively new concept have not been explored in depth.

Table 1: Catalogue of fundamental nano-patterns [29]

Category	Name	Description
Calling	NoParams	takes no arguments
	NoReturn	returns void
	Recursive	calls itself recursively
	SameName	calls another method with the same name
	Leaf	does not issue any method calls
Object-Orientation	ObjectCreator	creates new objects
	FieldReader	reads (static or instance) field values from an object
	FieldWriter	writes values to (static or instance) field of an object
	TypeManipulator	uses type casts or instance of operations
Control Flow	**StraightLine**	no branches in method body
	Looping	one or more control flow loops in method body
	Exceptions	may throw an unhandled exception
Data Flow	**LocalReader**	reads values of local variables on stack frame
	LocalWriter	writes values of local variables on stack frame
	ArrayCreator	creates a new array
	ArrayReader	reads values from an array
	ArrayWriter	writes values to an array

3 Related Work

The idea of using software metrics and patterns to assess software defects is not new. In this section, we report on relevant research that addresses these concepts. After conducting a literature search, we were not able to find any studies evaluating the practical use of nano-patterns for software engineering applications. There was no prior work using nano-patterns to assess software defects. There are other techniques reported on in the literature and we highlight those studies in this section.

3.1 Metrics and Software Quality

Basili et al. [4] conducted an empirical study to investigate the suite of object-oriented design metrics. They were interested in learning if these metrics can be used as predictors of fault-prone classes and to determine if the metrics are suitable as early quality indicators. They reported that five out of the six Chidamber and Kemerer (CK) metrics (i.e., Weighted Methods per Class (WMC), Depth of Inheritance Tree of a class (DIT), Number of Children of a Class (NOC), Response for a Class (RFC), and Coupling between Objects classes (CBO)) were useful for predicting class fault-proneness during the high and low level design phase. They also noted that CK metrics were better predictors than the best set of traditional code metrics at the time.

Thwin et al. [32] showed the application of neural networks for estimating software quality using OO metrics (i.e., DIT, NOC, CBO, RFC, Inheritance Coupling (IC), Coupling Between Methods (CBM), Weighted Methods per Class (WMC), Number of Objects/Memory Allocations (NOMA)). Quality was derived from the number of defects in a class and number of lines changed per class (i.e., the higher the defect number, the lower the quality). It was concluded that the OO metrics used in the study were useful in predicting software quality.

Tahvildari et al. [31] proposed a framework that used a catalogue of object-oriented metrics for improving the quality of an OO legacy systems by suggesting potentially useful transformations (i.e., meta-patterns) for correcting potential design flaws. Initial experiments with this approach have demonstrated its feasibility and usefulness.

Abreu et al. [7] evaluated the impact of OO design (inheritance, polymorphism, coupling, etc.) on software quality attributes (reliability and maintainability) using a suite of metrics for OO design (MOOD). It was an empirical evaluation, and quality was measured on the basis of number of software defects and the amount of rework. They concluded that design alternatives may have a strong influence on resulting quality.

These studies used standard CK metrics and software design to assess defects and quality mostly at the class level. Though the results were positive, there is still a need for mechanisms that can assess software quality more accurately at lower levels of granularity (e.g., the method level).

3.2 Code Smells and Software Quality

Yamashita et al. [34] conducted a study to find out which code smells reflect the factors that affect code maintainability. Two sources for the analysis were used. In one source, observation and interviews with professional developers were carried out and, in the other, expert-based maintainability assessment was considered. The study provided insights on various maintainability factors. During the course of the study, some new factors like design consistency, duplicated code, etc. were also reported.

Foutse et al. [21] empirically investigated if the classes suffering from code smells are more change-prone than classes without code smells. The study found that in most of the cases, classes with code smells are more likely to change than classes without code smells. They also found that certain code smells are strongly correlated with change-proneness compared to others.

Olbrich et al. [24] studied god class code smells and brain class code smells in relation to change likelihood and defect likelihood. The results indicated a higher rate of changes and software defects for both god and brain class code smells compared to other classes. However, when the data was normalized both god and brain class code smells suffered a lower rate of changes and defects.

In 2010, Marco et al. [10] conducted an empirical study on six open source systems to evaluate the relationship between design flaws of software systems and software defects. Frequency of design flaws and correlation of flaws with post-release defects were analyzed.

Yamashita et al. [30] investigated how code smells were related to maintenance effort. With the help of an eclipse IDE plugin, the exact amount of time developers spent maintaining files was collected from six developers who performed three maintenance tasks. The study concluded that all 12 investigated code smells had limited effect on the maintenance effort. The authors suggest that reducing file size and improving work processes to reduce the number of revisions would reduce maintenance effort. However, the authors claim these suggestions are not trivial to implement. For example, reducing the amount of functionality from one class either increases the total number of files in a system or adds burden for other files.

Code smells are design flaws that are thought to lower the quality of software. It was expected that code smells would be associated with high defect likelihood and high change likelihood. However, results from various empirical studies are unable to definitively support such conjecture. For example, Olbrich et al. [24], initially found God and Brain classes to contain more defects and changes more frequently than other areas of the code. But when they normalized the measured effects by size, they found God and Brain classes to have fewer defects and changes less often. They further went on to state that these classes may serve as an efficient way of organizing code. In another study, Yamashita et al. [30] found that no code smells (among 12 investigated) were associated with increased effort when adjusting by file size and number of changes. Similarly, Zhang et al. [35] showed Duplicated code smells had more faults while Data Clumps, Switch Statements, Speculative Generality, and Middle Man smells were not likely to be fault-prone. Considering these studies, the use of code smells alone as an indicator of high defect likelihood does not seem appropriate.

4 Methodology

This section elaborates on the research goals and associated research questions, the tools used in this study, the experimental design, the selected open source software for collecting the historical data and finally, the data collection process.

4.1 Research Goals

The goals of this research are formatted using the Goal Question Metric (GQM) approach. In GQM [33], measurements are defined in a top-down fashion and research goals are defined first. Based on the goal, questions are generated that must be answered for the goals to be achieved. The questions require identifying metrics with the data that supports answers to the questions. The research goals for this study are presented below. The metrics are elaborated in the remaining subsections of Section 4.

G1: Analyze the distribution and likelihood of various nano-patterns occurring in defective methods.

The aim is to examine how nano-patterns are distributed in defective code and examine the relationship between nano-patterns.

RQ1: How frequently do nano-patterns occur in software defects?

RQ2: How are nano-pattern distributed in software defects?

RQ3: What is the co-occurrence between various nano-patterns in software defects?

G2: Determine which nano-patterns are more defect-prone.

Some of the nano-patterns are easier to understand and implement than others. Therefore, it is likely that some nano-patterns are more vulnerable to defects than others.

RQ4: Are certain nano-patterns disproportionately involved with defective methods?

RQ5: How are nano-patterns associated with a defect's priority?

RQ6: Does the frequency of nano-patterns in defect methods affect the priority rating?

4.2 Experimental Tools

The Nano-Pattern Detector and JIRA Extractor are the two tools mainly used for data extraction. Figure 2 shows the high level diagram of these tools. The Nano-Pattern Detector is a command line tool developed by Singer et al. [29] whereas the JIRA Extractor was developed internally for this research.

4.2.1 Nano-Pattern Detector. The Nano-Pattern Detector detects nano-patterns in Java bytecode class files [29]. The tool is based on ASM bytecode analysis toolkit [8]. The tool reads in Java files and outputs a bitstring of nano-patterns for each method in a class. The tool uses ASM API extensively with 600 lines of code written in Java. The tool operates in the following ways [29]:

- Simple iteration over a method bytecode array for searching specific bytecode instruction matched with particular nano-patterns.
- Simple regular expressions match on method signatures.

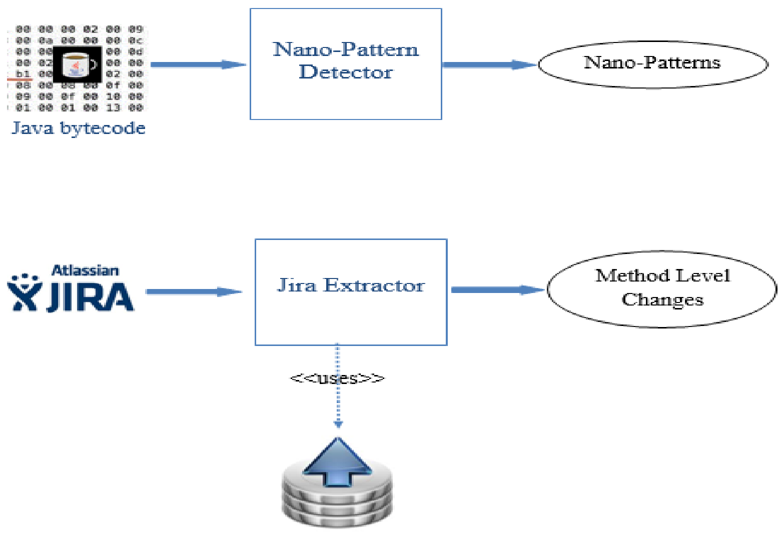


Figure 2: Nano-pattern detector and JIRA extractor

We modified the original source code to take input of the Java classes through 'properties' files instead of command line arguments. We additionally added new functionality to store results into a MySQL database (instead of simply displaying results to the console).

4.2.2 Jira Extractor. In its current state, the tool is able to extract issues from JIRA and get a list of all the classes and methods from SVN that were changed to address the issue. The tool makes extensive use of external libraries like JIRA REST Java Client (JRJC), SVNKit, Java parser, etc.

Figure 3 shows the high level workflow of the tool. The tool gets a list of issues from JIRA. For each issue, a revision number is extracted from SVN Log or by making a query to FishEye based on the issue number. FishEye is an Atlassian product that can be used to integrate source code with JIRA and also provides a read only access to Subversion [13].

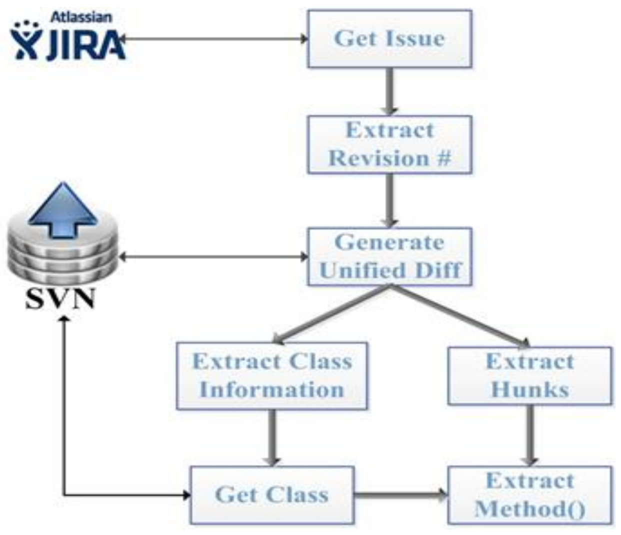


Figure 3: High level flowchart of JIRA extractor

From the revision number, unified diff content is generated using SVNKit. The unified diff content is then parsed to extract information about the set of classes that were changed. For each class, a set of hunk (hunks are set of changes in a unified diff file where change was effective) is also extracted. Finally, the content of a class corresponding to the extracted revision number is retrieved which is then parsed using Java Parser [20] to get the list of methods (along with their contents) whose content were changed.

In this study, the Nano-Pattern Detector was used to detect nano-patterns for a method and the JIRA Extractor was used to extract list of classes and methods responsible for a JIRA issue.

4.3 Experimental Design

The study is conducted in two steps:

- Identify nano-patterns of the methods in the source code.
- Search issue tracking systems for bugs and extract methods from source code repositories that were changed to fix the bugs.

Figure 4 illustrates our experimental design. First, we use the Nano-Pattern Detector tool to scan the entire source code of a software project and store the nano-pattern information for each method in the database. Then, we use the JIRA Extractor tool to search for issues in the issue tracking system (JIRA) and extract a list of methods that were changed to fix these issues. JIRA Extractor uses information like revision number, version number, issue type, etc. from issues to pull methods from the SVN repository. These methods are then matched with methods stored in the database to get their nano-patterns.

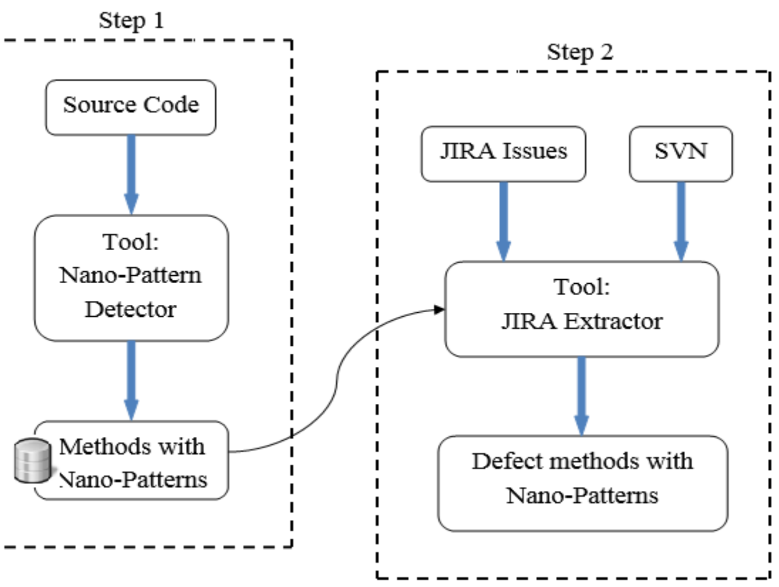

Step 1
Source Code
Tool: Nano-Pattern Detector
Methods with Nano-Patterns
Step 2
JIRA Issues
SVN
Tool: JIRA Extractor
Defect methods with Nano-Patterns

Figure 4: Experimental design

4.4 Historical Data

We chose open source software systems as our initial targets as their source code and issues list are openly available and the systems represent real-world, multi-developer, software intensive systems. Source code is managed through a version control system. A version control system keeps track of every single commit made on the source code repository. Project issues are managed by the JIRA issue tracking system. We selected three Java based open-source software projects from the Apache Software Foundation using the following criteria:

- Projects are written in Java
- Source code is publicly accessible
- Project issues are publicly accessible
- Source code is managed using SVN
- Project issues are managed using JIRA
- Project should have multiple release versions (at least 5 versions)
- Should have at least 1500 issues

4.4.1 Apache Hive. Apache Hive [2] is an open source volunteer software project under the Apache Software Foundation that facilitates querying and managing large datasets residing in distributed storage. There are over 8000 Hive issues logged in JIRA. We ran the static analysis tool Understand on the '/src' folder of different versions of Hive downloaded from "https://archive.apache.org/dist/hive/" to get the details of the source code as shown in Table 2.

4.4.2 Apache Lucene Core. Apache Lucene [1] is a "full-featured text search engine library" written in Java. It is an open source project from Apache Foundation and available for free download. It is released under the Apache Software License. Though originally written in Java, it has been ported to other programming languages like C++, python, Ruby, PHP, etc. There are over 5000 Lucene issues logged in JIRA. We ran the static analysis tool Understand on the '/src' folder of different versions of Lucene downloaded from "https://archive.apache.org/dist/lucene/java/" to get the details of the source code as shown in Table 3.

Table 2: Hive details

	Hive 0.14.0	Hive 1.0.0	Hive 0.8.0	Hive 0.10.0	Hive 0.12.0
Classes	8,305	8,289	2,379	3,559	5,872
Code Lines	672,725	668,363	220,218	283,829	465,026
Executable Statements	276,896	275,082	91,468	119,175	188,878
Files	3,408	3,395	1,277	1,461	2,527
Functions	59,457	58,599	20,127	25,142	42,130

Table 3: Lucene details

	Lucene 2.9.4	Lucene 3.1.0	Lucene 3.3.0	Lucene 3.5.0	Lucene 3.6.2
Classes	1,285	1,407	1,543	1,678	4,690
Code Lines	92,725	105,153	115,353	129,984	382,400
Executable Statements	44,230	46,578	51,221	56,189	148,750
Files	655	693	751	817	2,527
Functions	8,200	8,482	9,185	10,028	24,877

4.4.3 Apache Hadoop MapReduce Project. Apache Hadoop [17] is an open source software framework for storing and processing of large amounts of data. Hadoop Common, Distributed File System, Yarn, MapReduce, etc. are some of important modules of Apache Hadoop framework. Hadoop MapReduce was created at Google for parallel processing of large data sets. There are over 5000 MapReduce issues logged in JIRA. We ran the static analysis tool Understand on the mapreduce project inside '/src' folder for version '1.0.0' and '1.2.0', and on '/src' folder inside Mapreduce project module for version '2.0.0 – alpha' and '2.0.2 – alpha' to get the details of source code as shown in Table 4. All the versions were downloaded from "https://archive.apache.org/dist/hadoop/common/".

4.5 Data Collection

Initially JIRA Extractor tool extracted 4619, 2022, and 1566 defect methods from JIRA issues labelled as 'bug' for Hive, Lucene, and Mapreduce, respectively. Table 5 summarizes the data reduction method. Uniqueness of the method was based on the combination of its 'issue id', 'class name', 'method name', 'type signature', and 'version' where:

- 'issue id' is the JIRA issue number
- 'class name' is the name of java class including its package where the method exists
- 'method name' is the name of method
- 'type signature' is the java signature of method
- 'version' is the project's version number

We then filtered methods by their package name to make sure we include only those methods that belonged to the software projects in which we are interested. For Hive, all those methods were included whose package name contained "*org.apache.hadoop.hive*" or "*org.apache.hive*". Similarly, Lucene and Mapredure were filtered by "*org.apache.lucene*", and "*org.apache.hadoop.mapred*", respectively.

Some of the versions had very few methods. So we removed all versions that had less than 30 defect methods. Consequently, we had 11, 17, and 9 versions with 4612, 1868, and 613 methods for Hive, Lucene, and Mapreduce, respectively. All other methods were excluded from the study.

5 Results

This section will present the experimental results. We have two sets of results. The first set of results relates to the distribution and likelihood of nano-patterns in defect methods. The second set of results assists in understanding which nano-patterns are more defect-prone.

Table 4: Hadoop MapReduce details

	Mapreduce 1.0.0	Mapreduce 1.2.0	Mapreduce 2.0.0 - alpha	Mapreduce 2.0.2 - alpha
Classes	624	723	970	715
Code Lines	47,231	54,266	74,576	57,898
Executable Statements	17,157	19,584	27,842	22,396
Files	311	370	469	326
Functions	4,430	5,006	5,280	4,075

Table 5: Data reduction

	Hive	Lucene	Mapreduce
Initial count of defect methods	4619	2022	1566
Count of defect methods after filtering by package name	4612	2015	915
Count of versions with defect count > 30	11	17	9
Cumulative count of defect method for version with defect count>30	4612	1868	613

RQ1: How frequently do nano-patterns occur in software defects?

We counted the occurrence of each pattern in defect methods and calculated their proportion for all projects under study (Hive, Lucene, and Mapreduce). Table 6 shows, for each nano-pattern, its count and its proportion in percentage. It can be seen from the table that the presence of Recursive and Leaf nano-patterns is negligible. The likelihood of patterns in all the three projects shows a similar trend. With respect to data, the following patterns have a high presence:

- LocalReader with 95.97%, 98.93%, and 94.62% presence in Hive, Lucene, and Mapreduce, respectively.
- LocalWriter with 76.52%, 72.16%, and 83.69% presence in Hive, Lucene, and Mapreduce, respectively.
- ObjectCreator with 70.21%, 68.47%, and 76.02% presence in Hive, Lucene, and Mapreduce, respectively.
- FieldReader with 75.05%, 86.24%, and 68.19% presence in Hive, Lucene, and Mapreduce, respectively.

RQ2: How are nano-patterns distributed in software defects?

For each defect method, we counted the number of nano-patterns found in a method. Figure 5, 6, and 7 show the

Table 6: Occurrence percentage of each nano-pattern in defect method list

	Hive		**Lucene**		**Mapreduce**	
Nano-Patterns	**Patterns Count**	**Percentage**	**Patterns Cou**	**Percentage**	**Patterns Count**	**Percentage**
NoParams	883	19.15	700	37.47	286	46.66
NoReturn	1984	43.02	939	50.27	426	69.49
Recursive	160	3.47	7	0.37	1	0.16
SameName	689	14.94	326	17.45	53	8.65
Leaf	288	6.24	86	4.60	7	1.14
ObjectCreator	3238	70.21	1279	68.47	466	76.02
FieldReader	3461	75.04	1611	86.24	418	68.19
FieldWriter	1132	24.54	529	28.32	84	13.70
TypeManipulator	2377	51.54	641	34.31	192	31.32
StraightLine	1146	24.85	516	27.62	199	32.46
Looping	1823	39.53	703	37.63	218	35.56
Exceptions	2879	62.42	1159	62.04	422	68.48
LocalReader	4426	95.97	1848	98.93	580	94.62
LocalWriter	3529	76.52	1348	72.16	513	83.69
ArrayCreator	814	17.65	314	16.81	130	21.21
ArrayReader	969	21.01	346	18.52	108	17.62
ArrayWriter	623	13.51	291	15.58	111	18.11

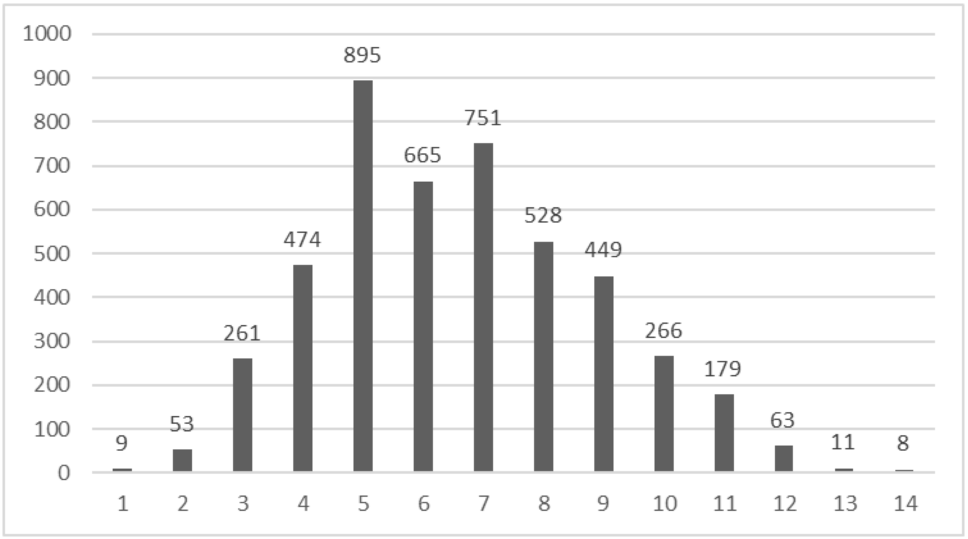


Figure 5: Frequency distribution of Hive (number of methods [y-axis] that contain the specified number of nano-patterns [x-axis])

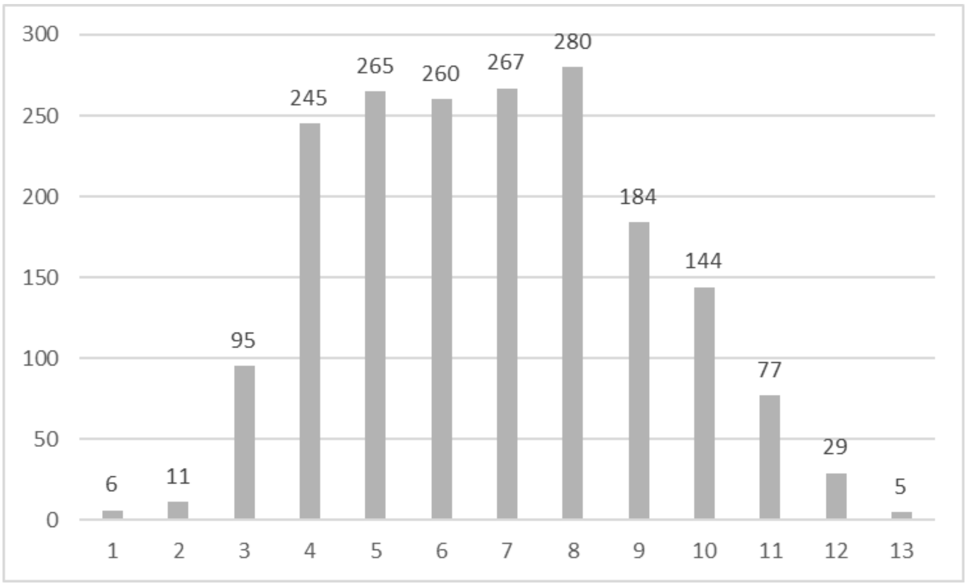


Figure 6: Frequency distribution of Lucene (number of methods [y-axis] that contain the specified number of nano-patterns [x-axis])

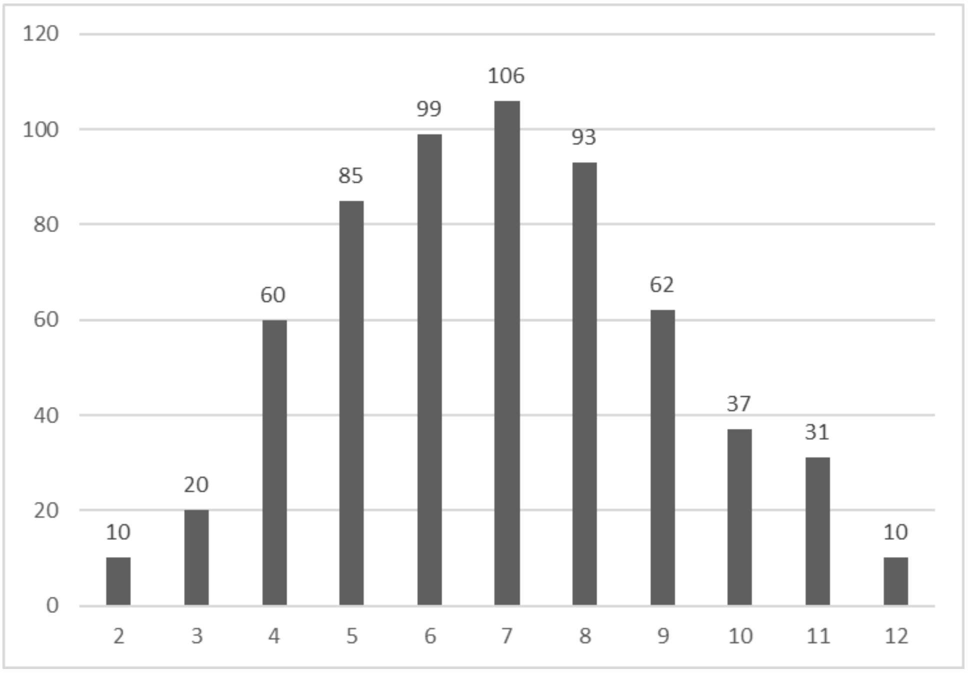


Figure 7: Frequency distribution of Mapreduce (number of methods [y-axis] that contain the specified number of nano-patterns [x-axis])

distribution of the number of nano-patterns in a method for Hive, Lucene, and Mapreduce, respectively. The x-axis represents the number of nano-patterns contained in a method. The y-axis shows the number of methods that contain that number of nano-patterns. All the methods have at least one nano-pattern (i.e., there is not a method without any of the 17 defined nano-patterns). The number of patterns in a majority of the methods is between 4 and 9. In very few methods, the number of patterns is 1 or 2 or >12. In Hive, no methods have more than 14 nano-patterns and in Lucene, no methods have more than 13 patterns. Similarly, in Mapreduce, no methods have more than 12 nano-patterns. No method across all 3 projects has all the 17 nano-patterns.

RQ3: What is the co-occurrence between various nano-patterns in software defects?

A Pearson Chi-Square test at 95% confidence interval was conducted. Tables 7, 8, and 9 represent the Chi-Square test value for Hive, Lucene, and Mapreduce, respectively. All the values significant at 95% are shown highlighted in red. In most of the cases nano-patterns are significantly correlated in all the 3 projects under study.

The value of Chi Square is a function of both the proportion of observation in each cell of the contingency table and the total sample size [28]. In fact, the magnitude of the chi square value is directly proportional to the total sample size [28]. However, the degree of association between variables is only a function of the cell proportion and is independent of the sample size [28]. Because of the large sample size across all the three systems in our study, it is very likely the chi square

Table 7: Pearson Chi-Square test for Hive

	Noparams	NoReturn	Recursive	Samename	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	StraightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
Noparams		1.44	6.68	0.05	349.47	191.12	1.14	0.04	264.3	329.49	63.41	241.74	569	443.98	118.41	48.14	75.35
NoReturn	1.44		1.76	2.68	3.45	70.3	28.84	343.13	3.75	0.68	0.29	76.58	4.49	11.4	2.9	1.7	1.48
Recursive	6.68	1.76		0.18	11.04	7.6	0.3	21.38	46.9	54.8	81.21	4.75	6.97	40.61	0.03	16.21	7.19
SameName	0.05	2.68	0.18		53.95	16.32	51.18	2.33	0.54	18.34	2.15	32.55	22.89	1.92	6	8.51	0.23
Leaf	349.47	3.45	11.04	53.95		723.91	262.09	8.25	313.63	780.95	197.27	226.53	781.73	947.2	65.84	63.9	40.86
ObjectCreator	191.12	70.3	7.6	16.32	723.91		138.35	25.31	407.01	846.88	403.71	365.82	101.59	1212.07	240.17	101.83	91.59
FieldReader	1.14	28.84	0.3	51.18	262.09	138.35		134.18	267.49	448.61	182.21	116.3	471.75	229.48	139.1	194.16	92.24
FieldWriter	0.04	343.13	21.38	2.33	8.25	25.31	134.18		4.1	79.04	67.68	6.6	57.64	34.55	11.76	115.87	52.07
TypeManipulator	264.3	3.75	46.9	0.54	313.63	407.01	267.49	4.1		818.67	1287.64	172.96	181.14	808.9	389.81	236.39	245.88
StraightLine	329.49	0.68	54.8	18.34	780.95	846.88	448.61	79.04	818.67		996.74	225.38	561.29	1719.87	148.33	328.78	124.23
Looping	63.41	0.29	81.21	2.15	197.27	403.71	182.21	67.68	1287.64	996.74		164.12	126.69	912.21	403.43	688.74	511.78
Exceptions	241.74	76.58	4.75	32.55	226.53	365.82	116.3	6.6	172.96	225.38	164.12		18.87	397.74	194.48	24.34	97.65
LocalReader	569	4.49	6.97	22.89	781.73	101.59	471.75	57.64	181.14	561.29	126.69	18.87		622.72	41.54	48.95	30.27
LocalWriter	443.98	11.4	40.61	1.92	947.2	1212.07	229.48	34.55	808.9	1719.87	912.21	397.74	622.72		215.6	187.41	139.69
ArrayCreator	118.41	2.9	0.03	6	65.84	240.17	139.1	11.76	389.81	148.33	403.43	194.48	41.54	215.6		250.62	2263.55
ArrayReader	48.14	1.7	16.21	8.51	63.9	101.83	194.16	115.87	236.39	328.78	688.74	24.34	48.95	187.41	250.62		650.05
ArrayWriter	75.35	1.48	7.19	0.23	40.86	91.59	92.24	52.07	245.88	124.23	511.78	97.65	30.27	139.69	2263.55	650.05	

Table 8: Pearson Chi-Square test for Lucene

	NoParams	NoReturn	Recursive	SameName	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	StraightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
NoParams		72.6	1.61	0.23	6.04	19.83	43.09	0.06	0.68	12.17	0.02	25.92	0.05	0.27	0.73	0.03	10.05
NoReturn	72.6		0.15	11.55	26.31	13.64	3.63	40.66	26.47	6.9	36.93	202.96	0.77	16.54	46.59	5.72	25.69
Recursive	1.61	0.15		1.49	0.34	0.42	0	0	8.24	2.68	3.42	1.1	0.08	2.71	1.42	2.76	0.01
SameName	0.23	11.55	1.49		19.06	48.73	7.19	0.4	1.95	7.4	2.55	2.98	4.27	28.5	8.42	0	4.62
Leaf	6.04	26.31	0.34	19.06		195.76	23.64	2.42	22.75	108.8	38.89	92.87	167.91	140.15	18.22	1.96	14.24
ObjectCreator	19.83	13.64	0.42	48.73	195.76		24.11	0	114.71	122.3	102.84	112.68	13.86	332.16	66	30.52	24.1
FieldReader	43.09	3.63	0	7.19	23.64	24.11		26.89	40.88	126.97	23.17	8.82	86.48	96.24	5.62	7.28	3.46
FieldWriter	0.06	40.66	0	0.4	2.42	0	26.89		2.12	104.78	45.9	15.99	5.42	1.38	28.8	7.72	64.22
TypeManipulator	0.68	26.47	8.24	1.95	22.75	114.71	40.88	2.12		236.39	319.78	101.44	10.56	250.08	61.66	17.42	19.84
StraightLine	12.17	6.9	2.68	7.4	108.8	122.3	126.97	104.78	236.39		430.21	39.78	45.9	438.42	75.36	129.93	67.04
Looping	0.02	36.93	3.42	2.55	38.89	102.84	23.17	45.9	319.78	430.21		112.6	12.2	387.32	182.68	405.43	189.34
Exceptions	25.92	202.96	1.1	2.98	92.87	112.68	8.82	15.99	101.44	39.78	112.6		0.07	151.31	87.05	12.95	38.92
LocalReader	0.05	0.77	0.08	4.27	167.91	13.86	86.48	5.42	10.56	45.9	12.2	0.07		45.4	4.08	4.6	3.73
LocalWriter	0.27	16.54	2.71	28.5	140.15	332.16	96.24	1.38	250.08	438.42	387.32	151.31	45.4		117.16	92.35	105.06
ArrayCreator	0.73	46.59	1.42	8.42	18.22	66	5.62	28.8	61.66	75.36	182.68	87.05	4.08	117.16		40.26	1018.83
ArrayReader	0.03	5.72	2.76	0	1.96	30.52	7.28	7.72	17.42	129.93	405.43	12.95	4.6	92.35	40.26		104.01
ArrayWriter	10.05	25.69	0.01	4.62	14.24	24.1	3.46	64.22	19.84	67.04	189.34	38.92	3.73	105.06	1018.83	104.01	

Table 9: Pearson Chi-Square test for Mapreduce

	Noparams	NoReturn	Recursive	Samename	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	straightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
Noparams		78.06	0.88	0.13	2.98	0.70	6.78	17.58	2.24	0.52	0.05	2.01	0.05	0.02	0.00	0.87	4.61
NoReturn	78.06		2.28	1.43	0.51	4.35	0.72	4.84	2.54	0.18	0.01	7.79	0.65	1.70	0.02	1.94	11.46
Recursive	0.88	2.28		0.09	0.01	3.18	2.15	0.16	2.20	0.48	1.81	2.21	0.06	0.20	0.27	4.68	0.22
Samename	0.13	1.43	0.09		0.67	12.00	3.27	32.96	3.93	0.74	0.06	4.05	3.30	13.24	6.48	7.66	4.36
Leaf	2.98	0.51	0.01	0.67		22.45	15.18	1.12	3.23	14.73	3.91	5.35	124.45	36.32	0.20	1.51	1.57
ObjectCreator	0.70	4.35	3.18	12.00	22.45		9.58	2.00	18.42	26.08	24.95	17.02	39.99	104.98	21.80	24.41	20.92
FieldReader	6.78	0.72	2.15	3.27	15.18	9.58		32.83	40.60	106.41	19.45	4.85	31.05	24.73	4.82	7.91	2.71
FieldWriter	17.58	4.84	0.16	32.96	1.12	2.00	32.83		3.79	7.99	2.08	3.94	3.36	0.05	11.52	0.74	6.27
TypeManipulator	2.24	2.54	2.20	3.93	3.23	18.42	40.60	3.79		67.97	95.50	3.66	15.91	44.56	13.56	33.11	1.99
StraightLine	0.52	0.18	0.48	0.74	14.73	26.08	106.41	7.99	67.97		162.62	1.25	66.19	168.10	14.75	37.54	9.88
Looping	0.05	0.01	1.81	0.06	3.91	24.95	19.45	2.08	95.50	162.62		15.95	19.25	62.29	51.50	130.55	39.06
Exceptions	2.01	7.79	2.21	4.05	5.35	17.02	4.85	3.94	3.66	1.25	15.95		3.32	9.19	21.05	31.84	15.86
LocalReader	0.05	0.65	0.06	3.30	124.45	39.99	31.05	3.36	15.91	66.19	19.25	3.32		178.92	4.79	7.46	5.35
LocalWriter	0.02	1.70	0.20	13.24	36.32	104.98	24.73	0.05	44.56	168.10	62.29	9.19	178.92		16.54	25.55	13.84
ArrayCreator	0.00	0.02	0.27	6.48	0.20	21.80	4.82	11.52	13.56	14.75	51.50	21.05	4.79	16.54		78.20	469.64
ArrayReader	0.87	1.94	4.68	7.66	1.51	24.41	7.91	0.74	33.11	37.54	130.55	31.84	7.46	25.55	78.20		70.25
ArrayWriter	4.61	11.46	0.22	4.36	1.57	20.92	2.71	6.27	1.99	9.88	39.06	15.86	5.35	13.84	469.64	70.25	

value was affected by the sample size. As a part of further analysis, we computed the phi coefficient (φ) to measure a degree of association between nano-patterns. The Phi coefficient is a measure of association for 2x2 contingency table and is independent of sample size [28]. The use of phi coefficient is most commonly endorsed for 2x2 contingency table. The strength of association is determined by the following criteria [28]:

- Small association (.10 <= φ <.30)
- Medium association (.30 <= φ < .50)
- High association (φ >= .50)

Tables 10, 11, and 12 represent the phi coefficient values for Hive, Lucene, and Mapreduce, respectively. Values in red represent high association and values in green represent medium association. With respect to data in Tables 10, 11, and 12, the following nano-patterns have high association:

- ArrayWriter and ArrayCreator nano-patterns have high association across all the three software systems.
- LocalWriter and StraightLine nano-patterns have high negative association in Hive and Mapreduce.
- Looping and TypeManipulator, and LocalWriter and ObjectCreator nano-patterns have high association only in Hive whereas LocalWriter and LocalReader, and Looping and StraightLine nano-patterns have high association only in Mapreduce.

The results from RQ4 suggest that ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalReader, and LocalWriter nano-patterns are highly prone to defects. Tables 13, 14, and 15 (subset of Tables 10, 11, and 12, respectively) present the association between these 7 defect-prone nano-patters for Hive, Lucene, and MapReduce, respectively. Values in red are high associations and values in green are medium associations. With respect to data in 0, 14, and 15, we see the following observations:

- Almost all these 7 nano-patterns have positive association
- ObjectCreator and LocalWriter nano-patterns have high association in Hive and medium association in Lucene, and Mapreduce.
- TypeManipulator and Looping nano-patterns have high association in hive and medium association in Lucene, and Mapreduce.
- Looping and LocalWriter nano-patterns have medium association across all the three software systems.

- **RQ4: Are certain nano-patterns disproportionately involved with defective methods?**
- Some nano-patterns are abundantly present in the source code and therefore the likelihood of their presence in defect code is also abundant. To avoid this bias, we compared the proportion of each nano-pattern with itself - between defect methods and the entire source code. Figure 8, 9, and 10 show the comparison of the average defect proportion of each nano-pattern between defect methods and the entire source code for Hive, Lucene, Mapreduce, etc. The average defect proportion is the average of the proportion of each nano-pattern in the selected version's defect methods. Similarly, average source code proportion is the average of proportion of each nano-pattern in the selected version's source code. It

can be observed in the Figures 8, 9, and 10 that proportions of ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalWriter, ArrayCreator, ArrayReader, and ArrayWriter are much higher in defect methods than the entire source code across all projects under study. Similarly, the proportion of SameName and StraightLine looks much lower in defect methods than the entire source code across all the projects under study. There is no significant difference in the proportion of other nano-patterns between defect code and entire source code.

We conducted the z-test to analyze if the proportions of nano-patterns in defect methods are significantly different from the overall source code at 95% confidence (i.e. α = 0.05).

Tables 16, 17, and 18 show the result of z-test for proportion for Hive, Lucene, and Mapreduce, respectively where:

Table 10: The phi coefficient for Hive

	NoParams	NoReturn	Recursive	samename	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	straightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
NoParams		-0.02	-0.04	0.00	0.28	-0.20	0.02	0.00	-0.24	0.27	-0.12	-0.23	-0.35	-0.31	-0.16	-0.10	-0.13
NoReturn	-0.02		0.02	0.02	0.03	-0.12	0.08	0.27	-0.03	-0.01	0.01	0.13	0.03	-0.05	0.03	-0.02	-0.02
Recursive	-0.04	0.02		-0.01	-0.05	0.04	0.01	0.07	0.10	-0.11	0.13	0.03	0.04	0.09	0.00	0.06	0.04
Samename	0.00	0.02	-0.01		-0.11	-0.06	0.11	0.02	0.01	0.06	-0.02	0.08	0.07	-0.02	-0.04	-0.04	0.01
Leaf	0.28	0.03	-0.05	-0.11		-0.40	-0.24	0.04	-0.26	0.41	-0.21	-0.22	-0.41	-0.45	-0.12	-0.12	-0.09
ObjectCreator	-0.20	-0.12	0.04	-0.06	-0.40		0.17	0.07	0.30	-0.43	0.30	0.28	0.15	0.51	0.23	0.15	0.14
FieldReader	0.02	0.08	0.01	0.11	-0.24	0.17		0.17	0.24	-0.31	0.20	0.16	0.32	0.22	0.17	0.21	0.14
FieldWriter	0.00	0.27	0.07	0.02	0.04	0.07	0.17		0.03	-0.13	0.12	0.04	0.11	0.09	0.05	0.16	0.11
TypeManipulator	-0.24	-0.03	0.10	0.01	-0.26	0.30	0.24	0.03		-0.42	0.53	0.19	0.20	0.42	0.29	0.23	0.23
StraightLine	0.27	-0.01	-0.11	0.06	0.41	-0.43	-0.31	-0.13	-0.42		-0.46	-0.22	-0.35	-0.61	-0.18	-0.27	-0.16
Looping	-0.12	0.01	0.13	-0.02	-0.21	0.30	0.20	0.12	0.53	-0.46		0.19	0.17	0.44	0.30	0.39	0.33
Exceptions	-0.23	0.13	0.03	0.08	-0.22	0.28	0.16	0.04	0.19	-0.22	0.19		0.06	0.29	0.21	0.07	0.15
LocalReader	-0.35	0.03	0.04	0.07	-0.41	0.15	0.32	0.11	0.20	-0.35	0.17	0.06		0.37	0.09	0.10	0.08
LocalWriter	-0.31	-0.05	0.09	-0.02	-0.45	0.51	0.22	0.09	0.42	-0.61	0.44	0.29	0.37		0.22	0.20	0.17
ArrayCreator	-0.16	0.03	0.00	-0.04	-0.12	0.23	0.17	0.05	0.29	-0.18	0.30	0.21	0.09	0.22		0.23	0.70
ArrayReader	-0.10	-0.02	0.06	-0.04	-0.12	0.15	0.21	0.16	0.23	-0.27	0.39	0.07	0.10	0.20	0.23		0.38
ArrayWriter	-0.13	-0.02	0.04	0.01	-0.09	0.14	0.14	0.11	0.23	-0.16	0.33	0.15	0.08	0.17	0.70	0.38	

Table 11: The phi coefficient for Lucene

	NoParams	NoReturn	Recursive	SameName	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	StraightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
NoParams		0.20	-0.03	0.01	0.06	-0.10	0.15	-0.01	-0.02	0.08	0.00	0.12	0.01	0.01	-0.02	0.00	-0.07
NoReturn	0.20		-0.01	-0.08	-0.12	0.09	0.04	0.15	0.12	-0.06	0.14	0.33	-0.02	0.09	0.16	0.06	0.12
Recursive	-0.03	-0.01		-0.03	-0.01	-0.01	0.00	0.00	0.07	-0.04	0.04	-0.02	0.01	0.04	-0.03	0.04	0.00
SameName	0.01	-0.08	-0.03		-0.10	-0.16	-0.06	0.01	-0.03	0.06	-0.04	0.04	0.05	-0.12	-0.07	0.00	-0.05
Leaf	0.06	-0.12	-0.01	-0.10		-0.32	-0.11	-0.04	-0.11	0.24	-0.14	-0.22	-0.30	-0.27	-0.10	-0.03	-0.09
ObjectCreator	-0.10	0.09	-0.01	-0.16	-0.32		0.11	0.00	0.25	-0.26	0.23	0.25	0.09	0.42	0.19	0.13	0.11
FieldReader	0.15	0.04	0.00	-0.06	-0.11	0.11		0.12	0.15	-0.26	0.11	0.07	0.22	0.23	0.05	0.06	0.04
FieldWriter	-0.01	0.15	0.00	0.01	-0.04	0.00	0.12		0.03	-0.24	0.16	0.09	0.05	0.03	0.12	0.06	0.19
TypeManipulator	-0.02	0.12	0.07	-0.03	-0.11	0.25	0.15	0.03		-0.36	0.41	0.23	0.08	0.37	0.18	0.10	0.10
StraightLine	0.08	-0.06	-0.04	0.06	0.24	-0.26	-0.26	-0.24	-0.36		-0.48	-0.15	-0.16	-0.48	-0.20	-0.26	-0.19
Looping	0.00	0.14	0.04	-0.04	-0.14	0.23	0.11	0.16	0.41	-0.48		0.25	0.08	0.46	0.31	0.47	0.32
Exceptions	0.12	0.33	-0.02	0.04	-0.22	0.25	0.07	0.09	0.23	-0.15	0.25		-0.01	0.28	0.22	0.08	0.14
LocalReader	0.01	-0.02	0.01	0.05	-0.30	0.09	0.22	0.05	0.08	-0.16	0.08	-0.01		0.16	0.05	0.05	0.04
LocalWriter	0.01	0.09	0.04	-0.12	-0.27	0.42	0.23	0.03	0.37	-0.48	0.46	0.28	0.16		0.25	0.22	0.24
ArrayCreator	-0.02	0.16	-0.03	-0.07	-0.10	0.19	0.05	0.12	0.18	-0.20	0.31	0.22	0.05	0.25		0.15	0.74
ArrayReader	0.00	0.06	0.04	0.00	-0.03	0.13	0.06	0.06	0.10	-0.26	0.47	0.08	0.05	0.22	0.15		0.24
ArrayWriter	-0.07	0.12	0.00	-0.05	-0.09	0.11	0.04	0.19	0.10	-0.19	0.32	0.14	0.04	0.24	0.74	0.24	

Table 12: The phi coefficient for MapReduce

	NoParams	NoReturn	Recursive	SameName	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulato	StraightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
NoParams		0.36	-0.04	0.01	-0.07	-0.03	0.11	0.17	-0.06	0.03	0.01	0.06	-0.01	0.01	0.00	-0.04	0.09
NoReturn	0.36		-0.06	-0.05	-0.03	0.08	0.03	0.09	-0.06	-0.02	0.00	0.11	-0.03	0.05	0.01	-0.06	0.14
Recursive	-0.04	-0.06		-0.01	0.00	-0.07	-0.06	-0.02	0.06	-0.03	0.05	-0.06	0.01	0.02	-0.02	0.09	-0.02
SameName	0.01	-0.05	-0.01		-0.03	-0.14	0.07	0.23	0.08	0.03	-0.01	-0.08	0.07	-0.15	-0.10	-0.11	-0.08
Leaf	-0.07	-0.03	0.00	-0.03		-0.19	-0.16	-0.04	-0.07	0.16	-0.08	-0.09	-0.45	-0.24	-0.02	-0.05	-0.05
ObjectCreator	-0.03	0.08	-0.07	-0.14	-0.19		0.13	0.06	0.17	-0.21	0.20	0.17	0.26	0.41	0.19	0.20	0.18
FieldReader	0.11	0.03	-0.06	0.07	-0.16	0.13		0.23	0.26	-0.42	0.18	-0.09	0.23	0.20	0.09	0.11	0.07
FieldWriter	0.17	0.09	-0.02	0.23	-0.04	0.06	0.23		0.08	-0.11	-0.06	-0.08	0.07	0.01	-0.14	-0.03	-0.10
TypeManipulator	-0.06	-0.06	0.06	0.08	-0.07	0.17	0.26	0.08		-0.33	0.39	-0.08	0.16	0.27	0.15	0.23	0.06
straightLine	0.03	-0.02	-0.03	0.03	0.16	-0.21	-0.42	-0.11	-0.33		-0.52	0.05	-0.33	-0.52	-0.16	-0.25	-0.13
Looping	0.01	0.00	0.05	-0.01	-0.08	0.20	0.18	-0.06	0.39	-0.52		0.16	0.18	0.32	0.29	0.46	0.25
Exceptions	0.06	0.11	-0.06	-0.08	-0.09	0.17	-0.09	-0.08	-0.08	0.05	0.16		0.07	0.12	0.19	0.23	0.16
LocalReader	-0.01	-0.03	0.01	0.07	-0.45	0.26	0.23	0.07	0.16	-0.33	0.18	0.07		0.54	0.09	0.11	0.09
localWriter	0.01	0.05	0.02	-0.15	-0.24	0.41	0.20	0.01	0.27	-0.52	0.32	0.12	0.54		0.16	0.20	0.15
ArrayCreator	0.00	0.01	-0.02	-0.10	-0.02	0.19	0.09	-0.14	0.15	-0.16	0.29	0.19	0.09	0.16		0.36	0.88
ArrayReader	-0.04	-0.06	0.09	-0.11	-0.05	0.20	0.11	-0.03	0.23	-0.25	0.46	0.23	0.11	0.20	0.36		0.34
ArrayWriter	0.09	0.14	-0.02	-0.08	-0.05	0.18	0.07	-0.10	0.06	-0.13	0.25	0.16	0.09	0.15	0.88	0.34	

Table 13: The phi coefficient of seven defect-prone nano-patterns in Hive

	ObjectCreator	FieldReader	TypeManipulator	Looping	Exceptions	localReader	LocalWriter
ObjectCreator		0.17	0.30	0.30	0.28	0.15	0.51
FieldReader	0.17		0.24	0.20	0.16	0.32	0.22
TypeManipulator	0.30	0.24		0.53	0.19	0.20	0.42
Looping	0.30	0.20	0.53		0.19	0.17	0.44
Exceptions	0.28	0.16	0.19	0.19		0.06	0.29
LocalReader	0.15	0.32	0.20	0.17	0.06		0.37
LocalWriter	0.51	0.22	0.42	0.44	0.29	0.37	

Table 14: The phi coefficient of seven defect-prone nano-patterns in Lucene

	ObjectCreator	FieldReader	TypeManipulator	Looping	Exceptions	LocalReader	LocalWriter
ObjectCreator		0.11	0.25	0.23	0.25	0.09	0.42
FieldReader	0.11		0.15	0.11	0.07	0.22	0.23
TypeManipulator	0.25	0.15		0.41	0.23	0.08	0.37
Looping	0.23	0.11	0.41		0.25	0.08	0.46
Exceptions	0.25	0.07	0.23	0.25		-0.01	0.28
LocalReader	0.09	0.22	0.08	0.08	-0.01		0.16
LocalWriter	0.42	0.23	0.37	0.46	0.28	0.16	

Table 15: The phi coefficient of seven defect-prone nano-patterns in MapReduce

	ObjectCreator	FieldReader	TypeManipulator	Looping	Exceptions	LocalReader	LocalWriter
ObjectCreator		0.13	0.17	0.20	0.17	0.26	0.41
FieldReader	0.13		0.26	0.18	-0.09	0.23	0.20
TypeManipulator	0.17	0.26		0.39	-0.08	0.16	0.27
Looping	0.20	0.18	0.39		0.16	0.18	0.32
Exceptions	0.17	-0.09	-0.08	0.16		0.07	0.12
LocalReader	0.26	0.23	0.16	0.18	0.07		0.54
LocalWriter	0.41	0.20	0.27	0.32	0.12	0.54	

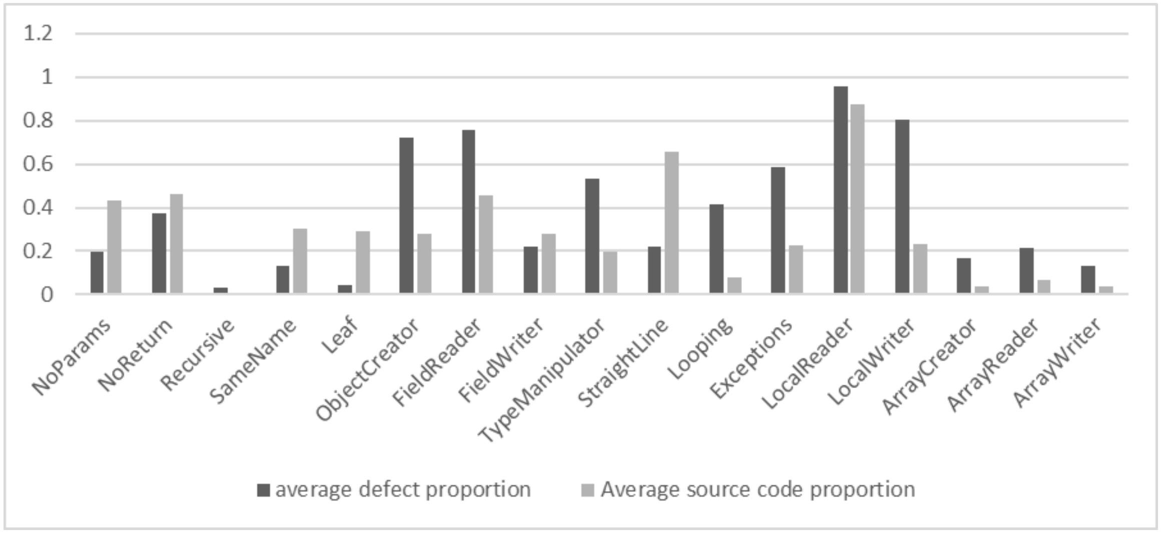


Figure 8: Comparison of average defect proportion vs. average source code proportion for each nano-pattern in Hive

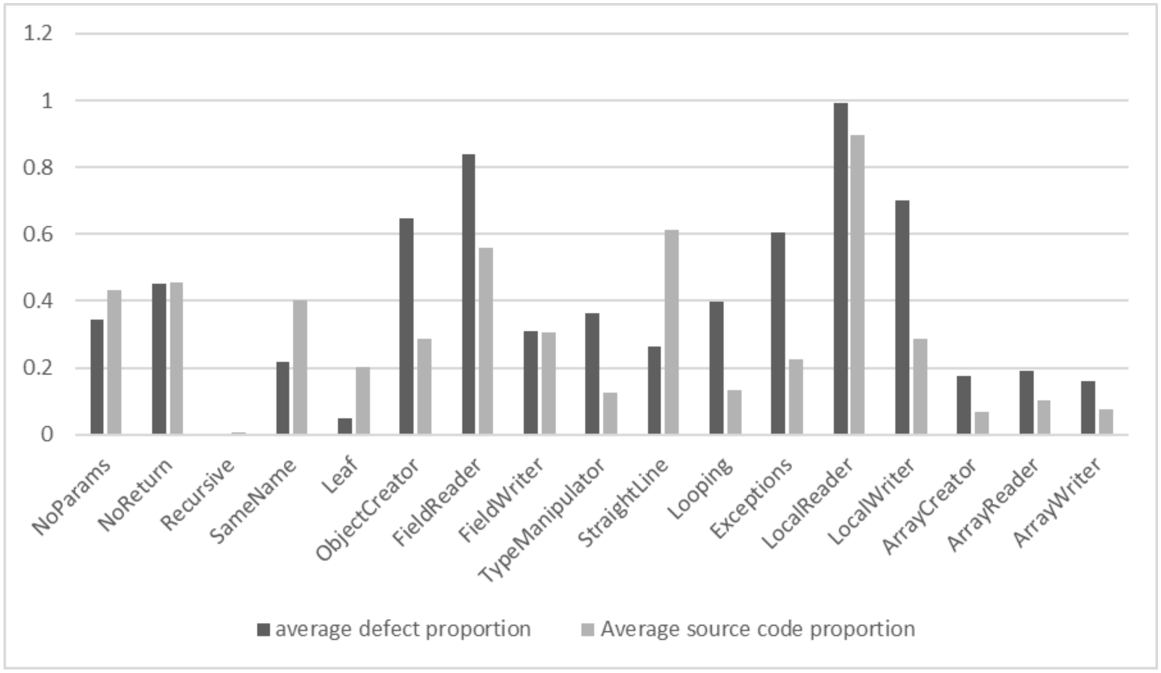


Figure 9: Comparison of average defect proportion vs. average source code proportion for each nano-pattern in Lucene

- L= proportion of nano-pattern that is significantly lower in defect methods
- H = proportion of nano-pattern that is significantly higher in defect methods
- NS = z-test result not significant
- F = Normality check for z-test failed

From data in Tables 16, 17, and 18, the following results are observed:

- Proportion of SameName and StraightLine nano-patterns is significantly lower in defect methods in a majority of the cases
- Proportion of ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalReader, and LocalWriter are significantly higher in defect method in a majority of the cases
- For all other nano-patterns, no specific trend is observed

RQ5: How are nano-patterns associated with a defect's priority?

There are 5 levels of priority for issues in JIRA: Blocker, Critical, Major, Minor, and Trivial. To avoid any confusion and misunderstanding while creating an entry for defect in

JIRA, we grouped closely related priorities together. We categorized priority into 3 levels: High, Medium, and Low. Blocker and Critical were grouped as High, Major was Medium, and Minor and Trivial were grouped as Low. We calculated Pearson Chi-Square test between each nano-pattern and defect's priority (High, Medium, and Low). Table 19 gives Pearson Chi-Square test value between each nano-pattern and each defect's priority. The test was conducted at 95% confidence interval. Significant Chi-Square values (i.e., values greater than critical value (3.841) are shown as bold in Table 19. It can be seen that in many cases, nano-patterns are significantly associated with a defect's priority. Most of the patterns have association with priority across all levels (high, medium, low). However, no clear pattern is observed between any particular nano-pattern and particular defect's priority except for TypeManipulator. TypeManipulator is only

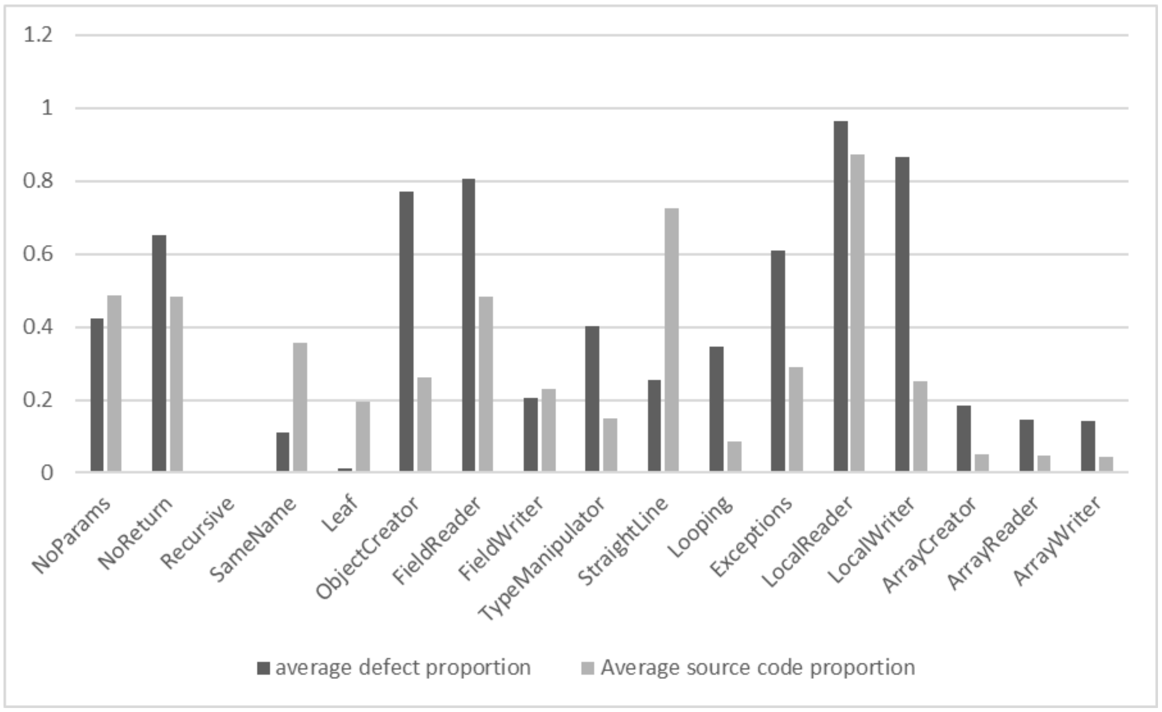


Figure 10: Comparison of average defect proportion vs average source code proportion for each nano-pattern in Mapreduce

Table 16: z-test proportion result for Hive

Patterns/Versions	0.13.1	0.13.0	0.12.0	0.11.0	0.10.0	0.9.0	0.8.1	0.8.0	0.7.1	0.7.0	0.6.0
NoParams	L	L	L	L	L	L	L	L	L	L	L
NoReturn	NS	L	L	L	NS	H	L	L	L	NS	NS
Recursive	H	F	F	F	F	F	F	F	F	F	F
SameName	L	L	L	L	L	L	NS	L	L	L	L
Leaf	L	L	L	L	L	L	L	L	L	L	L
ObjectCreator	H	H	H	H	H	H	H	H	H	H	H
FieldReader	H	H	H	H	H	H	H	H	H	H	H
FieldWriter	H	H	L	NS	NS	L	L	L	L	NS	NS
TypeManipulator	H	H	H	H	H	H	H	H	H	H	H
StraightLine	L	L	L	L	L	L	L	L	L	L	L
Looping	H	H	H	H	H	H	F	H	H	F	H
Exceptions	H	H	H	H	H	H	H	H	H	H	H
LocalReader	H	H	H	H	H	H	F	NS	NS	F	H
LocalWriter	H	H	H	H	H	H	H	H	H	H	H
ArrayCreator	H	H	H	H	H	H	F	F	H	F	F
ArrayReader	H	H	H	H	H	H	F	H	H	F	H
ArrayWriter	H	H	H	H	H	H	F	F	H	F	F

Table 17: z-test proportion result for Lucene

Patterns/Versions	4.9.0	4.8.1	4.8.0	4.7.1	4.7.0	4.6.0	4.5.0	4.4.0	4.3.1	4.2.1	4.2.0	4.1.0	4.0-BET	4.0-LPHA	4.0.0	3.6.0	3.5.0
NoParams	NS	NS	NS	H	NS	L	L	NS	L	NS	L	L	L	NS	NS	L	NS
NoReturn	H	NS	NS	H	NS	NS	NS	L	L	NS	L	NS	NS	NS	NS	H	L
Recursive	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
SameName	L	NS	NS	L	NS	L	L	L	L	NS	L	L	L	NS	L	L	L
Leaf	L	L	L	L	L	L	L	L	L	L	L	L	L	NS	L	L	NS
ObjectCreator	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
FieldReader	H	H	H	H	H	NS	H	H	NS	H	H	H	H	H	H	H	H
FieldWriter	L	NS	H	L	NS	NS	NS	NS	L	H	NS	NS	NS	NS	H	NS	NS
TypeManipulator	H	H	F	H	H	H	H	NS	F	H	H	H	H	H	H	H	NS
StraightLine	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L
Looping	H	H	F	H	H	H	H	H	F	H	NS	H	H	H	H	H	H
Exceptions	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
LocalReader	H	H	F	H	F	H	F	H	F	F	H	H	H	H	H	H	H
LocalWriter	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
ArrayCreator	H	H	F	H	F	F	F	NS	F	F	F	H	H	F	F	H	NS
ArrayReader	H	H	F	H	F	H	F	NS	F	F	NS	NS	NS	NS	NS	H	H
ArrayWriter	H	H	F	H	F	F	F	NS	F	F	F	H	H	F	F	NS	NS

Table 18: z-test proportion result for Mapreduce

Patterns/Versions	2.4.0	2.3.0	2.1.1-bet	1.1.2	1.1.1	1.0.0	0.23.7	0.23.4	0.23.1
NoParams	NS	L	L	NS	H	H	L	NS	NS
NoReturn	H	H	H	NS	H	H	NS	L	L
Recursive	F	F	F	F	F	F	F	F	F
SameName	L	NS	L	L	L	L	L	L	L
Leaf	L	L	L	L	L	L	L	L	L
ObjectCreator	H	H	H	H	H	H	H	H	H
FieldReader	H	H	H	H	H	NS	H	H	H
FieldWriter	NS	H	NS	NS	NS	L	NS	NS	NS
TypeManipulator	H	H	H	H	H	H	H	H	H
StraightLine	L	L	L	L	L	L	L	L	L
Looping	F	F	F	H	H	H	F	F	F
Exceptions	H	H	H	H	H	H	H	NS	NS
LocalReader	F	F	F	H	NS	NS	F	NS	NS
LocalWriter	H	H	H	H	H	H	H	H	H
ArrayCreator	F	F	F	F	F	H	F	F	F
ArrayReader	F	F	F	F	F	H	F	F	F
ArrayWriter	F	F	F	F	F	H	F	F	F

Table 19: Pearson Chi-Square test (nano-patterns vs. defect's priority)

	Hive			Lucene			Mapreduce		
Patterns/Priority	High	Medium	Low	High	Medium	Low	High	Medium	Low
NoParams	1.65	1.9	0.36	2.13	12.68	10.92	10.21	7.78	0.12
NoReturn	5.5	0.74	9.84	1.08	4.34	13.15	14.60	15.76	0.81
Recursive	2.09	2.51	0.51	0.95	5.53	4.72	0.22	0.29	0.04
SameName	1.98	0.01	1.89	1.29	0.69	0	3.91	7.71	3.85
Leaf	8.59	22.72	11.57	0.26	12.59	17.13	0.08	0.27	0.31
ObjectCreator	0.64	0.1	1.22	4.84	3.36	0.16	5.01	2.45	1.10
FieldReader	0.16	0	0.12	1.16	4.45	13.65	0.13	2.05	5.14
FieldWriter	12.56	2.54	1.25	2.89	1.97	0.09	0.65	6.53	14.07
TypeManipulator	3.41	7.02	2.89	3.34	6.34	2.64	0.48	0.78	0.24
StraightLine	2.04	2.29	9.93	0.59	0.02	0.26	0.09	0.03	0.06
Looping	7.95	4.82	0.08	1.66	0.05	2.07	2.92	4.14	0.88
Exceptions	0.82	13.55	29.29	8.71	14.4	5.16	4.22	6.28	1.57
LocalReader	9.03	0	6.59	0.02	0.07	0.22	3.48	3.60	0.13
LocalWriter	4.01	0.91	0.32	12.97	20.29	6.82	0.41	0.02	2.24
ArrayCreator	6.95	24.03	14.67	0.07	2.08	2.67	2.98	8.42	7.31
ArrayReader	31.07	11.78	0.35	0.96	0.13	0.17	1.05	1.82	0.69
ArrayWriter	30.22	39.06	9.04	1.64	5.06	3.11	2.91	6.29	3.72

only associated with Medium priority for Hive and Lucene.

RQ6: Does the frequency of nano-patterns in defect methods affect priority rating?
As explained in RQ5, the 5 levels of priority were categorized into 3 levels (High, Medium, and Low). Figure 11 gives the proportion of each priority in defect methods for Hive, Lucene, and Mapreduce. The proportion of medium priority is much more than high and low priority. Their proportion is more than 78% in all projects. The proportion of low and high priority is comparatively small. In two cases, the proportion of low priority is slightly higher than the proportion of high priority and, in one case, the proportion of high priority is fairly higher than the proportion of low priority.

Figure 12 represents a line chart for the number of nano-patterns in a method and their count in the defect method list. X-axis is the number of nano-patterns in a method and Y-axis is the count of such method in defect method list. It is obvious to notice that medium priority dominates the frequency of defects. However, if we see a trend of the line moving from one point to another along X-axis, it looks similar for all three priority levels (high, medium, and low).

Analysis of Variance (ANOVA) test was conducted to see if the mean number of nano-patterns in a method is significantly different for high, medium, and low priority defect methods. The test was conducted at 95% confidence interval with the help of statistical software SPSS version 22. Table 20 shows the result of ANOVA test for Hive, Lucene, and Mapreduce where:

- S = significantly different
- NS = not significantly different

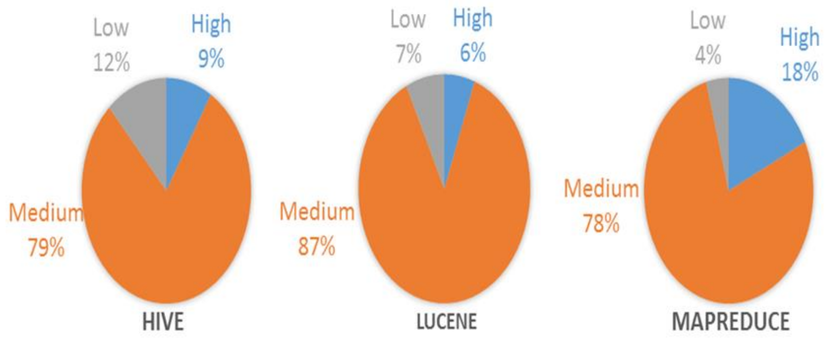


Figure 11: Pie chart of defect's priority proportion

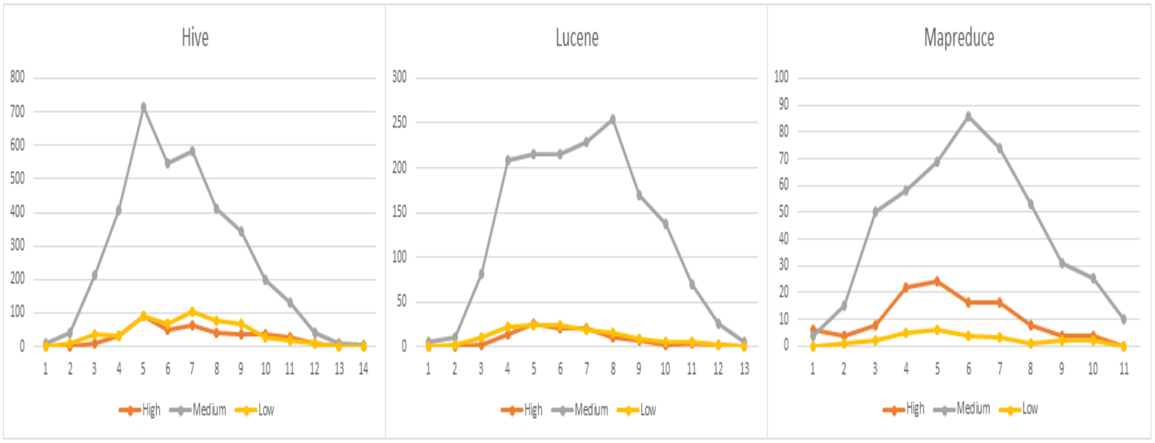


Figure 12: Line chart of number of nano-patterns in a method vs. their count in defect methods list

From Table 20, we can see that for Hive, the mean number of nano-patterns was significantly different between high and medium priority defect methods than between medium and low priority defect methods. For Lucene, the mean number of nano-patterns was significantly different between low and medium priority defects while the mean number of nano-patterns for Mapreduce was significantly different between high and medium priority levels.

Table 20: ANOVA test result for Hive, Lucene, and MapReduce

	Hive	Lucene	Mapreduce
High – Medium	S	NS	S
High – Low	NS	NS	NS
Medium – Low	S	S	NS

6 Discussion

In this section, we interpret and analyze the results as well as describe the significance of our findings. The results from Tables 16, 17, and 18 show the proportion of ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalReader, and LocalWriter nano-patterns to be significantly higher in the defect code compared to the entire source code. This implies that methods with these nano-patterns are more likely to have defects. Further, a high presence of ObjectCreator, LocalReader LocalWriter, and FieldReader (see Table 6) in defect code makes them even more susceptible to defects. Below, we describe the possible reasons why some of the nano-patterns are more vulnerable to defects.

As reported by Shah et al. [27] novice developers dislike the forced handling of exceptions imposed by Java. Developers often ignore exception handling code. Therefore, the complexity involved in handling exceptions could be one of the reasons for Exception nano-patterns (e.g., exception handling code) are more vulnerable to defects. A study by Sawadpong et al. [26] showed the exception handling code to have approximately three times higher defect density than the overall defect density of the system. In light of high susceptibility of the exception handling code towards software defects, we advise exception handling code should be carefully monitored. To check exception handling defects, we suggest measures advised by Shah et al. [27], like training of novice developers in perceiving exception handling code the way expert developers perceive (e.g., giving equal importance to both exception and non-exception functionality, training developers to think about both usual and exceptional behavior simultaneously, and proper use of exception handling).

The high defect rate for the TypeManipulator nano-pattern (i.e., code with type casts or instance of operations) could be because of the difficulty involved in understanding type manipulator code. Concepts like type erasure (Java Generics), type casting, etc. involved with Type Manipulator are often difficult to comprehend. Also, errors with type manipulator can be difficult to spot as class cast exceptions, which is very helpful in identifying errors with expression involving type cast.

Our results from Tables 16, 17, and 18 also show that the patterns like SameName and StraightLine are less prone to defects compared to other program constructs like Looping (i.e., one or more control flow loops in method body). This may be due to the fact that the flow of instruction in a straight line without any method call is easier to follow compared to the other program constructs like looping. Hyland et al. [19] stated that looping is one of the most difficult topics to understand along with arrays, threads, polymorphism, and exceptions.

Although none of the defect-prone nano-patterns have high associations across all three software systems, ObjectCreator and LocalWriter nano-patterns have a high association in Hive and falls in the upper limit of the medium associations identified in Lucene and Mapreduce (see Tables 13, 14, and 15). Since, both of these nano-patterns are widely present and defect-prone, and have strong associations, we suggest extra caution while using these two nano-patterns.

It is interesting to note that of the four categories of nano-patterns (see Table 1), our results indicate that the object-oriented and control flow categories are more vulnerable to defects. Three out of four patterns (i.e., ObjectCreator, FieldReader, and TypeManipulator) from the object-oriented category and two out of the three patterns (i.e., Looping and Exceptions) from the control flow category are associated with a high number of defects. Similar results were seen in the study by Basili et al. [3], where they reported both interface errors and control structure errors to be the major problem with the systems analyzed. With the claim that the pre-existing cognitive capabilities of humans better align with procedural languages than non-procedural language as shown by S. Papert [25], we can contend that object-oriented concepts like objects, inheritance, generics, etc. of Java are more difficult to understand and implement compared to other Java fundamental programming concepts like variables, arrays etc. This research could provide an explanation why object oriented patterns have a larger number of defects.

We did not observe any significant relationships between nano-patterns and the priority of each defect. In the light of these observations, we are unable to establish any significant relationship between nano-patterns and defect priority.

In this study, all the methods were treated in the same way regardless of their size. Insight into the effect of method length on nano-patterns would be interesting. It can be argued that the change in a method's length might change the number of nano-patterns in a method. However, this change should not have any influence on the overall defect rate. The result for RQ2 (see Figures 5, 6, and 7), shows that there is no specific trend on the increase/decrease of number of defects with the increase/decrease of number of nano-patterns in a method.

Existing literature on software engineering is inconclusive regarding the correlation between defect density and method length. Lipow [22] concluded that the number of defects per line of code increases with the number of lines of code in the program. However, in the study by Basili et al. [3], it was found that the larger the modules the lower the number of defects found. This happened even in the case when larger modules were more complex. Moller et al. [23] found that smaller modules have higher defect density and, for larger modules, size has apparently no effect on the fault rate. In light of these findings, even if the method length has an effect on number of nano-patterns in a method, it seems less likely to have any correlation with software defects. For more clarity on this, a separate study on how method length affects nano-patterns and overall defects is warranted.

7 Threats to Validity

We observed the following threats to validity for this research.

7.1 Construct Validity

A method can have more than one nano-pattern. In our study, any error encountered in a method is accounted to all nano-patterns present in a method. However, it might be the case that defect is actually related to only one or more of the present nano-patterns and not all. Our current methodology makes it difficult to distinguish the exact nano-pattern that is responsible for the defect. However, a refinement of our strategy would allow this measure and is left for future work.

7.2 External Validity

Our results are based on the study of three open source software projects. All the projects belong to Apache Software Foundation. Generalizing these results incorporates some bias due to all projects having similar development environments and following similar processes. We saw these projects as a proxy for commercially developed applications because each of them have an active developer community and support industrial applications.

7.3 Internal Validity

We have correlated the errors present in a method with the nano-patterns present in them. However, software development is such a complex process and many variables influence the code. In this study, we have not considered any other variables apart from nano-patterns.

8 Conclusions and Future Work

In this research, we demonstrated that certain categories of nano-patterns are more fault-prone than others. We prescribed a methodology for extracting modified classes and methods from a repository based on the issues specified in the repository's issue tracking systems. Using this methodology, we developed a JIRA Extractor tool that was used to extract data for this study. The JIRA Extractor tool is flexible and can be used in extracting data for other research questions where class and method-level changes are required.

The research was conducted in two steps. In the first step, issue tracking systems were mined for defects, and changed methods containing defects were extracted from their source code repositories. In the second step, the nano-patterns of each of these methods were calculated.

We found every defective method to have at least one nano-pattern. While the presence of FieldWriter, TypeManipulator, Looping, and Exception nano-patterns were extremely high, ObjectCreator, FieldReader, TypeManipulator, Looping, Exceptions, LocalReader, and LocalWriter nano-patterns were more defect-prone. We did not observe any significant relationship between any nano-pattern and the priority of the defect (i.e., high, medium, and low).

In considering the high affiliation between defects with object-oriented nano-patterns and control flow nano-patterns, we recommend developers take care when writing code that contains nano-patterns in these two categories. Classes and methods with a high number of nano-patterns of these two categories should be more heavily tested. We encourage the training of novice developers on the understanding and proper use of complex constructs like exception handling code. Using preventive measures like an "instanceOf" check (which prevents code from run-time errors from inappropriate type casts) yields better results.

Similar to code smells and micro-patterns, nano-patterns can be helpful in assessing defects and software quality. Since identification of nano-patterns can be automated and the presence of a nano-pattern is binary, conclusions drawn using nano-pattern data is deemed highly reliable.

Per our knowledge, after Singer et al. [29] defined nano-patterns, this is the first study involving nano-patterns. The knowledge on nano-patterns unveiled in this study will contribute to future research on various aspects of nano-patterns and how they can be used to evaluate code. Further, similar studies (to the ones described in this study) can be carried out to validate our findings on other systems. This study can be taken forward to analyze how nano-pattern affects change likelihood. Likewise, a study on how method size affects nano-patterns and overall defects would be insightful.

References

[1] "Apache Lucene," 21-Apr-2014. [Online]. Available: http://lucene.apache.org/. [Accessed: 22-Apr-2014].

[2] "Apache Hive," 22-Jun-2014. [Online]. Available: https://hive.apache.org/. [Accessed: 22-Jun-2014].

[3] V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Commun ACM*, 27(1)42-52, Jan. 1984.

[4] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Softw. Eng.*, 22(10):751-761, Oct. 1996.

[5] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, pp. 73-87, 2000.

[6] J. M. Bieman, N. Fenton, D. Gustafson, A. Melton, and L. Ott, *Fundamental Issues in Software Measurement*. Kansas State University, Department of Computing and Information Sciences, 1991.

[7] F. Brito e Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," *Software Metrics Symposium, 1996, Proceedings of the 3rd International*, pp. 90-99, 1996.

[8] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," *Adapt. Extensible Compon. Syst.*, 30:19, 2002.

[9] G. Concas, G. Destefanis, M. Marchesi, M. Ortu, and R. Tonelli, "Micro Patterns in Agile Software," *Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister and B. Weber, Eds. Springer Berlin Heidelberg, pp. 210-222, 2013.

[10] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the Impact of Design Flaws on Software Defects," *2010 10th International Conference on Quality Software (QSIC)*, pp. 23-31, 2010.

[11] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, "Micro Pattern Fault-Proneness," *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 302-306, 2012.

[12] G. Destefanis, *Assessing Sofware Quality by Micro Patterns Detection*, Doctoral Thesis, Universita' degli Studi di Cagliari, 2013.

[13] "FishEye," *FishEye*, 03-Jun-2014. [Online]. Available: https://www.atlassian.com/software/fisheye/overview. [Accessed: 18-Jun-2014].

[14] F. A. Fontana, B. Walter, and M. Zanoni, "Code Smells and Micro Patterns Correlations," RefTest 2013 Workshop, 2013.

[15] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, Boston, MA, 1999.

[16] J. (Yossi) Gil and I. Maman, "Micro Patterns in Java Code," *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, pp. 97-116, 2005.

[17] "Hadoop," 22-Jun-2014. [Online]. Available: http://hadoop.apache.org/. [Accessed: 22-Jun-2014].

[18] E. W. Host and B. M. Ostvold, "The Programmer's Lexicon, Volume I: The Verbs," *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pp. 193-202, 2007.

[19] E. Hyland and G. Clynch, "Initial Experiences Gained and Initiatives Employed in the Teaching of Java Programming in the Institute of Technology Tallaght," *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, Maynooth, County Kildare, Ireland, Ireland, pp. 101-106, 2002.

[20] "javaparser," *javaparser*, 03-Jun-2014. [Online]. Available: https://code.google.com/p/javaparser/. [Accessed: 18-Jun-2014].

[21] F. Khomh, M. Di Penta, and Y. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-Proneness," *16th Working Conference on Reverse Engineering, WCRE '09*, 2009, pp. 75-84, 2009.

[22] M. Lipow, "Number of Faults per Line of Code," *IEEE Trans. Softw. Eng.*, SE-8(4):437-439, July 1982.

[23] K.-H. Moller and D. J. Paulish, "An Empirical Investigation of Software Fault Distribution," *Software Metrics Symposium, 1993, Proceedings., First International*, pp. 82-90, 1993.

[24] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems," *2010 IEEE International Conference on Software Maintenance (ICSM)*, pp. 1-10, 2010.

[25] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., New York, NY, USA, 1980.

[26] P. Sawadpong, E. B. Allen, and B. J. Williams,

"Exception Handling Defects: An Empirical Study," *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 90-97, 2012
[27] H. B. Shah, C. Gorg, and M. J. Harrold, "Understanding Exception Handling: Viewpoints of Novices and Experts," *IEEE Trans. Softw. Eng.*, 36(2):150-161, Mar. 2010.
[28] D. J. Sheskin, *Parametric and Nonparametric Statistical Procedures*. United States: Chapman & Hall/CRC, 2000.
[29] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, "Fundamental Nano-Patterns to Characterize and Classify Java Methods," *Electron. Notes Theor. Comput. Sci.*, 253(7):191-204, Sep. 2010.
[30] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Trans. Softw. Eng.*, 39(8):1144-1156, Aug. 2013.
[31] L. Tahvildari and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations," *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings*, pp. 183-192, 2003.
[32] M. M. T. Thwin and T.-S. Quah, "Application of Neural Networks for Software Quality Prediction using Object-Oriented Metrics," *J. Syst. Softw.*, 76(2):147-156, May 2005.
[33] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric (GQM) Approach," *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., pp. 578-583, 2002.
[34] A. Yamashita and L. Moonen, "Do Code Smells Reflect Important Maintainability Aspects?," *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 306-315, 2012.
[35] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising Refactoring Using Code Bad Smells," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 458-464, 2011.

Ajay Deo holds a MS degree in Computer Science and Engineering from Mississippi State University, Mississippi, USA, and a BTech degree in Computer Science and Engineering from National Institute of Technology (NIT) Bhopal, India. He is passionate about translating academic research into industrial practice. He has particular interest in software architecture and design, software patterns, and technical debt.

Zadia Codabux is a PhD candidate in Computer Science at Mississippi State University (MSU). She holds a bachelor's and master's degree from The University of Technology Mauritius (UTM) and University of Mauritius (UOM) respectively. Prior to coming to MSU, she also worked as a faculty at UOM. Zadia is a recipient of the Fulbright Foreign Student Program Fellowship and the IBM PhD fellowship. Her research interests are Empirical Software Engineering, Technical Debt, Software Metrics, Predictive Analytics, Software Quality, Software Maintenance and Computer Science Education.

Kazi Zakia Sultana is currently a PhD student in the Department of Computer Science and Engineering, Mississippi State University, MS, USA. She joined here as a graduate student in August, 2014. She received her M.S. degree from the Department of Computer Science of Wayne State University, Detroit, MI, USA in 2011, and her B.Sc. (Engineering) from the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Bangladesh in 2006. Her research concentrations include Software Security, Code Patterns, and Data Mining. Her email address: ks2190@msstate.edu.

Byron J. Williams is an Assistant Professor in the Department of Computer Science and Engineering at Mississippi State University (MSU). He received his Ph.D. from MSU and worked as the Associate Director and Chief Software Engineer at the Center for Defense Integrated Data, in Jackson, MS. His research interests include taking an empirical approach to software maintenance, software security, agile methods and software development operations. He is an IEEE Computer Society Certified Software Development Professional (CSDP) and a Senior Member of the ACM.

Transforming C Applications with Meta-Programming

Songqing Yue[†]
University of Central Missouri, Warrensburg, Missouri 64093, USA

Jeff Gray[‡]
University of Alabama, Tuscaloosa, Alabama 35401, USA

Abstract

Computational reflection has shown much promise for improving the quality of software by providing programming language techniques to address issues of modularity, reusability, maintainability, and extensibility. The meta-object protocol (MOP) is a powerful tool to provide the capability of computational reflection by means of object-oriented and reflective techniques to organize a meta-level architecture. In this paper, we describe how to bring the power of computational reflection to the C programming language through a MOP that can be used to build arbitrary source-to-source program transformation libraries for large software systems in C. To assist application programmers with harnessing the power offered by meta-programming and reflection, a Domain-Specific Language (DSL) called SPOT (Specifying PrOgram Transformation) was extended to allow developers to specify direct manipulation of C programs.

This work is mainly motivated by the observation that C is one of the most widely used programming languages and there is a vast body of legacy C programs that are very costly to maintain and evolve. The design focus of the work presented in this paper is to automate program transformations through techniques of code generation, so that developers only need to specify desired transformations while being oblivious to the details about how the transformations are performed. The paper provides a general motivation for using meta-programming and reflection and explains the design and implementation of our MOP for C (called OpenC) and SPOT.

General Terms: Program transformation and domain-specific language.

Key Words: Computational reflection, program transformation, meta-object protocol, domain-specific language, and abstraction.

[†] Department of Mathematics and Computer Science. Email: syue@ucmo.edu.

[‡] Department of Computer Science. Email: gray@cs.ua.edu.

1 Introduction

Computational reflection was introduced into the context of computer science by Brian Smith as a way to extend the semantics of programming languages. According to Smith, a reflective system is able to reason about and manipulate itself based on an explicit and principled means of representing its implementation [21]. Maes [14] presented a formal definition of computational reflection as "*a computational system which is about itself in a causally connected way*." A computational system refers to a system running on a computer to solve problems in a specific domain. In order to achieve this, a system must have internal structures used to describe its domain (e.g., using data to represent entities and their relations and algorithms to operate on those data). Given this definition, every executing program can be considered a computational system because it manipulates abstractions for a specific problem domain.

"*Causally connected*" implies that the computational system and its domain are linked in such a way that if one changes, a corresponding change can be seen in the other. A reflective system is depicted as a computational system whose domain is itself (i.e., a reflective system has internal structures to describe itself). Its internal structures and its external behaviors are causally connected so that it is possible to change its behavior through manipulating its internal structures.

Computational reflection, in the realm of programming languages, provides the power to extend the semantics of a language by representing and modifying a program in the same way that a program represents and modifies the data that it processes [21]. Usually a system with the power of computational reflection includes a base-level and a meta-level. The base-level is responsible for dealing with computing results from the main domain of the application (this is the typical program written by programmers), and the meta-level addresses problems and returns information about the base-level program.

Reflection can be distinguished as structural reflection and behavioral reflection based on the dimension that the objects of the meta-level program operate [4]. Structural reflection is about the manipulation of the static structure of a program. With structural reflection, the definition of data structures, such as classes and methods can be retrieved and even modified (e.g., getting a list of all public methods available in a class definition, or adding a new method). Behavioral reflection focuses on the semantics of an executing system and provides a complete reification of both the semantics of the language and the execution states [7]. Behavioral reflection makes it possible to intercept and alter operations during run-time (e.g., field access and method invocation). Behavioral reflection allows for modifying the behavior of an operation, and structural reflection provides an ability to inspect and modify static data structures of the program. However, it is much easier to implement structural reflection and many languages have already integrated this feature, e.g., Java and Python. On the contrary, it is more challenging to realize complete behavioral reflection because it is especially difficult to incorporate behavioral properties without adversely affecting performance.

1.1 Meta-Object Protocol Overview

Meta-programming is a paradigm for building software that is able to automate program transformations through code generation or manipulation [22]. Computational reflection is a special case of meta-programming where the meta-program and the base-program are usually coded in the same programming language and sometimes the base-program itself is a meta-program [22].

In this work, we intend to bring the capacity of structural computational reflection to C programs through a technique of meta-programming named meta-object protocol (MOP). A MOP enables the extension or redefinition of a language's semantics to make it open and extensible by providing a set of interfaces to access the underlying language implementation [11]. With the MOP technique, the restraint that the meta-program and the base-program have to be coded in the same language can be loosed.

To allow transformation from a meta-level, there must be a clear representation for the base-program of its internal structure and entities (e.g., the classes and methods defined within an object-oriented program) and well-defined interfaces through which these entities and their relations can be manipulated [11]. Through the interfaces, a software developer can change the implementation and the behavior of the program incrementally to better suit their needs.

In a MOP, each entity in the base-program is represented with a meta-object in the meta-level program. The class from which the meta-object is instantiated is called the meta-class. For instance, for a function defined in C, a corresponding meta-object will be constructed in the meta-level program. The meta-object for the function holds adequate information to describe the structure and behavior of the function and interfaces carefully designed to alter them. The interfaces may manifest as a set of classes or methods so that users can create variants of the default language implementation incrementally by sub-classing, specialization, or method combination [8].

Based on the time when the meta-objects exist, a MOP may be run-time or compile-time. Run-time MOPs function while a program is executing and can be used to perform real-time adaptation, e.g., the Common Lisp Object System (CLOS) [4] that allows the mechanisms of inheritance, method dispatching, class instantiation and other language implementation details to be modified during program execution. As an example, a language called 3-KRS [14] has complete self-representation at run-time via meta-objects to affect the run-time execution. In comparison, meta-objects in compile-time MOPs only exist during compilation and may be used to manipulate the compiling process. Two examples of compile-time MOPs are OpenC++ [5] and OpenJava [23]. We have also implemented a compile-time MOP for Fortran named OpenFortran [27]. Though not as powerful as run-time MOPs, compile-time MOPs are simpler to implement and offer an advantage in reducing run-time overhead because of the source-to-source translation, which uses a standard language compiler for the final compilation.

1.2 Main Contributions: OpenC and SPOT

The key contributions of this paper include two aspects: (1) a MOP, named OpenC, for building transformation libraries that can be applied in a transparent manner for C programs, and (2) the SPOT DSL that is built on top of OpenC that is focused on the meta-meta-level to provide a higher level of abstraction for expressing program transformations. The SPOT DSL can assist in bridging the gap between the traditional programming style and the intensive meta-programming techniques involved in using a MOP.

OpenC works at compile-time and provides the capability of meta-programming to C programs. However, it is still a challenge for most developers to program with the concept of meta-programming. To assist developers in accessing the capabilities of OpenC, we have extended SPOT [28], which was created originally to provide a higher level of abstraction for expressing program transformations for Fortran [27], to make it compatible with OpenC. With SPOT, source-to-source program transformation can be performed transparently, whereby developers do not need to know the details on how the transformations are performed. In addition, with constructs and actions provided, SPOT allows users to express the intent of modifying C programs in a direct manner, therefore coding with it more aligns with a developer's comprehension of program transformation than coding with MOP capabilities or manipulating an abstract syntax tree (AST) as practiced by most program transformation engines (PTEs).

The paper is organized as follows. Section 2 describes the design and implementation of OpenC. Section 3 illustrates two case studies of using OpenC to first develop a simple profiling library and then facilitate parallelization of sequential C programs with OpenMP. Section 4 elaborates the extension of SPOT in order to accommodate OpenC. Section 5 shows related work. We present our future work and conclude the paper in Section 6.

2 The Implementation of OpenC Mop

C is one of the most widely used programming languages and there is a vast body of legacy C programs in use today, especially in the area of embedded systems and High Performance Computing [24]. It is often very expensive to make changes to legacy code on a large scale [3]. The procedural paradigm and lower-level programming constructs make C code even more difficult to maintain and evolve. In order to automate program translations for large-scale legacy C programs, we have implemented a MOP, named OpenC, which allows programmers to specify source-to-source program transformation for applications written in C. The benefit to application programmers is that they can use the OpenC libraries to translate their application code in a transparent and repeated way by only adding simple annotations. To the best of our knowledge, OpenC is one of the first MOPs to bring the power of static computational reflection to C.

Even though the MOP mechanism may assume an object-oriented meta-level language, the base-level language is not required to be object-oriented [11]. To implement OpenC, the base-level program refers to C applications to be manipulated and the meta-level program is written in C++, which is the language used in the underlying transformation engine ROSE [18, 22].

The libraries developed with OpenC work at the meta-level providing the capability of structural reflection to inspect and modify internal static data structures. OpenC also supports partial behavioral reflection, which assists in intercepting function calls and variable accesses to add new behavior to base-level programs. Considering that system performance should not be affected adversely by applying libraries, OpenC performs program transformation at compile-time to improve run-time performance.

2.1 OpenC Design Architecture

Figure 1 shows the high-level infrastructure where OpenC is used to fulfill source-to-source program translations. The base-level application is C source code and the meta-level library is developed with facilities provided by OpenC to perform transformations on the base-level code. OpenC takes the meta-level transformation and base-level C code as input and generates the extended C code to address the concerns expressed in the meta-program. The generated C code, which can be compiled by a traditional C compiler, is composed of both the original and newly translated C code placed in specific program locations.

In our approach, the low-level support is from a program transformation engine called ROSE [18] that integrates EDG [10] as the frontend for C programs. ROSE is an open source compiler infrastructure that allows users to build source-to-source transformation tools that read and translate programs in large-scale systems [18]. ROSE provides a rich set of interfaces for constructing an AST from the input source code, traversing and manipulating and regenerating source code from the AST.

Though ROSE is powerful in supporting specified program transformations, it is quite a challenge for most application developers to learn and use. Manipulation of an AST is greatly different than a programmer's intuitive understanding of programs. In contrast, the MOP mechanism resembles a developer's comprehension of program transformation by allowing direct manipulation of language constructs (e.g., functions, statements, structs) in the base-level code via the interfaces provided. Through a MOP, some language constructs that are not a first-class citizen can be promoted to first-class to allow for construction, modification and deletion [20].

2.2 OpenC Implementation Details

In OpenC, the top-level entities in the base-level code, such as struct definitions, variables and functions, are represented by meta-objects in the meta-level program. For instance, a function meta-object contains sufficient information about the structure and behavior of the function and interfaces carefully designed to alter them. With OpenC the source-to-source program transformations are performed in the steps described in the following paragraphs.

The base-level C source code is parsed and the top-level definitions are identified. The parse tree is traversed. For any applicable top-level definitions where a meta-program has been specified, a corresponding meta-object is constructed. The member function of the meta-object,

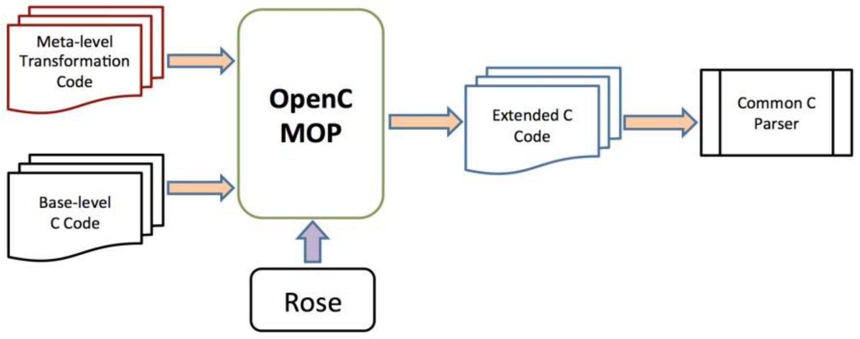


Figure 1: Overview of OpenC transformation process

OCExtendDefinition() is called to modify the AST to perform transformations. The parse trees generated from all meta-objects are synthesized and transformed back to C code, which is then passed on to a traditional C compiler.

OpenC provides facilities to develop translation libraries that are able to transform C code in multiple scopes (e.g., manipulating a function, a struct, a file or even a whole project including multiple files). As an example, assume a user would like to create a new function *A* and call it from another function *B*. The translation scope can be the file (if function *A* and *B* are in the same file) or the whole project space (if *A* is generated in a different file than *B*).

Four types of meta-objects, as indicated in Table 1, are designed to support transformations of multiple scopes. They are *MetaFunction*, *MetaStruct*, *MetaFile* or *MetaGlobal*. The class from which the meta-object is instantiated is called the meta-class. The four meta-classes are all subclasses of the class named *MetaObject*. The member function *OCExtendDefinition()* declared in *MetaObject* should be overridden by all subclasses to perform called-side adaptions for the definition of a function or a struct (e.g., adding a new variable in a struct, or inserting statements in a function). OpenC also supports caller-side translations by overriding the following member functions defined in *MetaObject*:

- *OCExtendFunctionCall(string funName)*
  intercept function invocation and translate how it is invoked
- *OCExtendVariableRead(string varName)*
  intercept and translate the behavior of a variable reading
- *OCExtendVariableWrite(string varName)*
  intercept and translate the behavior of a variable writing

Translating the definition of a function is the finest level of granularity that OpenC supports. Because a C program is composed of definitions of functions (we ignore struct, union and enum in our discussion here on purpose due to simplicity), the manipulation of a file or a whole project is ultimately delegated to that of function definition. Therefore, in our implementation for OpenC, a *MetaFile* is composed of a group of *MetaFunctions, MetaGlobal* consists of several *MetaFile*s, and most of the facilitating member functions are defined in the class of *MetaFunction*.

Different types of meta-objects are often used collaboratively in a transformation task. If multiple-level translations are involved, the sequence for applying these meta-objects has to be arranged carefully to avoid conflicts that may introduce issues of non-determinacy in the transformation results. Transformation libraries should be written to perform transformations on a low level of the base-program first, and then on a higher level; for example, translating an isolated function contained by a file before performing the file-wide translations.

To allow application developers to apply transformation libraries by simply adding annotations to their base-level programs, OpenC provides a set of keywords to identify the annotations.

Table 1 summarizes the features of these keywords, including the type of the meta-object corresponding to each keyword, the place in the application code where a keyword is added, and the translation scope. For instance, META_FUNCTION is a new keyword designed to designate a meta-function (i.e., the translation scope is function-wide), which is defined in the library code, to a function definition in the base code.

ROSE is able to preserve all comments that appear in the source code, which are saved with the AST and can be obtained later by traversal [18]. We take advantage of this feature to allow application developers to annotate source code in the place of user comments. The annotation is used to specify a meta-object using keywords and special tokens, e.g., "//@OC::META_FUNCTION *metaFunName*."

Table 1: The keywords used as annotations in OpenC

Key Words	***Meta-Objects***	***Location***	***Scope***
META_FUNCTION	MetaFunction	Comment for the Function definition	The function
META_STRUCT	MetaStruct	Comment for the struct definition	The struct
META_FILE	MetaFile	Comment for any Function definition in the file	The whole file
META_GLOBAL	MetaGlobal	Comment for the Main function	The whole project

3 The Application of OpenC Mop

In this section, we first outline the implementation of the initial version of a profiling library that can be used to show the distribution of execution time among all function calls in a system. The main purpose is to illustrate how OpenC can be used to implement a translation library and how the library can then be used to add the profiling capability to any existing C application in a transparent way. Then, we show how OpenC can be used to solve real-world problems encountered in software maintenance and evolution by facilitating the parallelization of sequential C code with OpenMP [15].

3.1 Implementing a Profiling Library

As an example, consider the situation when we would like to know the time spent on executing each function call in the source code, as shown in Figure 2. Profiling is a useful technique to help developers obtain an overview of system performance. A general way to implement this is to create a helper function, say *profiling(char* pidentifier),* that calculates the execution duration by comparing the system time just before and after a function call. The only parameter is the identifier uniquely indicating a function call by splicing the caller's function name and the callee's function name.

```
//@OC::META_GLOBAL profilingMetaClass
1 int main(){
2   int radius;
3   scanf("%d", &radius);
4   if(getArea(radius)>10&&
          getCircumference(radius)<100)
5      return 1;
6   else
7      return 0;
```

Figure 2: Example source code to be transformed

For our purpose, we cannot simply insert profiling before and after every statement containing function calls in the main function because function calls to *getArea* and *getCircumference* are embedded in a condition statement as indicated by line 4 in Figure 2. Instead, we need first to rewrite the original code to normalize the function calls by adding temporary variables to have each function call appear in a standalone assignment statement, and then insert profiling before and after each standalone assignment statement, as shown in Figure 3.

In this example, with only three function calls in the main function, it may not seem like a challenge to code manually for the purpose of implementing the profiling functionality. However, the situation becomes labor-intensive and error-prone when many more functions or more scenarios where function calls are embedded in statements are involved, which is always the case in larger applications. More importantly, after adding the profiling functionality, the original code gets polluted and modifying code back and forth to enable and disable this functionality is extremely tedious.

```
1 int main(){
2    int radius;
3    profiling("main:scanf");
4    scanf("%d", &radius);
5    profiling("main:scanf");
6    profiling("main: getArea");
7    float tempVar1 = getArea(radius);
8    profiling("main: getArea");
9    profiling("main: getCircumference");
10   float tempVar2 = getCircumference(radius);
11   profiling("main: getCircumference");
12   if(tempVar1 >10 && tempVar2 <100)
13       return 1;
14   else
15       return 0;
16 }
```

Figure 3: Example source code after transformation

With OpenC, the process of normalizing function calls and invoking profiling around them in a large-scale system can be automated via code generation techniques. OpenC provides the ability to build a profiling library that automatically generates and integrates a new copy of the original application code and profiling code by manipulating the AST. The original code is kept intact and the application programmer never needs to see the generated code.

To implement the profiling library with the facilities provided by OpenC, we can choose to implement a meta-class inherited from *MetaFunction* to transform method invocations within a function. Or, we can also choose to subclass from *MetaGlobal* to perform file-wide (i.e., any functions within current file containing method invocations will be affected) or even project-wide transformations that translate all the files in a system by merging individual ASTs for each file into a single large AST. In the example, we choose *MetaGlobal* as the superclass.

To build the library, we override *OCExtendDefinition* to specify the translations. Figure 4 shows the code snippet implementing the overridden *OCExtendDefinition*. The *functionList* in line 7 is a member variable defined in *MetaGlobal* as a container holding the *MetaFunction* objects representing all function definitions in the file. The for-loop iterates through these objects to perform translation. Line 8 and line 19 work together to operate on a global scope stack, pushing the current scope (a function body in this case) onto the stack, which implies that all the following operations are done within the current scope and popping the current scope when translation is finished. Line 9 calls a member function *functionNormalization* defined in *MetaFunction* to normalize function calls in the current function. Line 11 collects all function-call expressions and line 12 loops through them to identify the statements in which a function-call expression is embedded. For each statement containing a function call, two additional function-call statements are generated by calling *buildFunctionCallStmt* with the first parameter indicating the function name (*profiling*), and the second parameter as the parameter list. The parameter list here contains only the identifier of the function call, composed by combining the caller's function name (*main*) and the callee's function name (*scanf*, *getArea* and *getCircumference*). The

```
1. class ProfilingMetaClass: public MetaGlobal{
2.    public:
3.       ProfilingMetaClass(string name);
4.       virtual void OCExtendDefinition();
5. };

6. void ProfilingMetaClass:: OCExtendDefinition(){
7.   for(int i=0; i<functionList.size(); i++){
8.     pushScopeStack(functionList[i]->getFunctionBodyScope());
9.     functionList[i]->functionNormalization();
10.    vector<SgFunctionCallExp*> funCallList = functionList[i]->getFunctionCallList();
11.    for(int j=0; j<funCallList.size(); j++){
12.       string callerName = functionList[i]->getName();
13.       string calleeName = get_name(funCallList[j]);
14.       SgStatement* targetStmt = functionList[i]->getStmtsContainFunctionCall(funCallList[j]);
15.       string identifier = callerName + ":" + calleeName;
16.       insertStatementBefore(targetStmt,buildFunctionCallStmt("profiling",\
                                                              buildParaList(identifier)));
17.       insertStatementAfter(targetStmt, buildFunctionCallStmt("profiling",\
                                                              buildParaList(identifier)));
18.    }
19.    popScopeStack();
20.  }
21.}
```

Figure 4: User-defined meta-class inherited from MetaGlobal

generated two function-call statements then are inserted before and after the statement, shown in line 16 and line 17.

The resulting translation of the code in Figure 2 is indicated in the code shown earlier in Figure 3.

As denoted by the user comment highlighted in bold in Figure 2, it is possible to use the profiling library by simply annotating the source code with a user comment starting with *"@OC:*." In the annotation, the keyword *META_GLOBAL* is used to associate a *MetaGlobal* object with the main function to perform file-wide or project-wide translation. With the purpose of getting the distribution of execution time among all function calls in an application, the meta-file object is instantiated from the meta-class *ProfilingMetaClass*, which can be replaced by any other meta-class as required to perform desired transformation.

Profiling is a typical example of a crosscutting concern that cannot be modularized in a single place with traditional programming paradigms such as object-oriented programming, and may be spread across multiple modularity boundaries. As demonstrated by the profiling library, OpenC can be used to support aspect-oriented programming (AOP) [12] in C by separating the implementation of the utility function of profiling with the core application. However, a MOP provides additional support code for transformations and can also be used to express more fine-grained program transformations at arbitrary places. The MOP-based approach is superior over the AOP-based approach in some cases because MOPs provide a richer interface that can be used to deal with a wider range of transformation challenges in more diverse scenarios that are not limited to crosscutting concerns.

3.2 Facilitating Parallelization with OpenMP

OpenMP is a parallel model for developing multithreaded programs in a shared memory setting [15]. It provides a flexible mechanism to construct programs with multithreads in languages like C, C++ and Fortran via a set of compiler directives (in the form of *pragma* directives in C) and run-time library routines.

OpenMP has been adopted in the area of high performance computing (HPC) due to its flexibility and the performance it can provide; however, it has its own set of maintenance issues due to its feature of invasive reengineering of existing programs [1]. It is very challenging to evolve a parallel application where the core logic code is often tangled with the code to accomplish parallelization. This situation often occurs when the computation code must evolve to adapt to new requirements or when the parallelization code needs to be changed according to the advancement in the parallel model being used, or needs to be totally rewritten using a different model.

With our approach, the process of instrumenting directives and calling run-time functions can be automated so that the sequential and parallel code can be managed separately and the parallelized application can be generated on demand with the latest sequential and parallel code. Figure 5 shows a code excerpt from a meta-program we have implemented to parallelize a C application that carries out a molecular dynamics simulation [17] using OpenMP [9]. Instead of manually instrumenting the sequential code, the meta-program can be applied to automatically insert corresponding directives to the places identified.

For example, line 5 identifies a *for* statement before which code is often tangled with the code to accomplish parallelize-tion. This situation often occurs when the computation code must evolve to adapt to new requirements or when the parallelization code needs to be changed line 6 adds a directive "*omp for reduction(...)*" as a pragma. The first parameter in "*getForStatement(...)*" is a tuple containing elements that are essential to constitute a loop statement. For example, "*k, 0, np, 1*" will be used to match a for statement as "*for (k=0; k<np; k++)*." The second parameter "*1*" indicates that the first statement matched is returned in case there are multiple matches

```
1. void ParaMDMetaClass:: OCExtendDefinition(){
2.   for(int i=0; i<functionList.size(); i++){
3.      if(functionList[i]->getName() == "compute"){
4.         pushScopeStack(functionList[i]->getFunctionBodyScope());
5.         SgStatement* targetStmt = getForStatement("k,0,np,1" , 1);
6.         insertPragmaBefore(targetStmt, "for reduction (+ : pe, ke)");
7.         targetStmt = getAssignmentStatement("ke", 0.0, 1);
8.         insertPragmaAfter(targetStmt, "parallel shared (f, nd,…) private ( I, j, k…)");
9.         popScopeStack();
10.     }
11.  }
12.}
```

Figure 5: OpenC code to parallelize molecular dynamics using OpenMP

in the same scope specified. Similarly, line 8 inserts a directive "*omp parallel shared(…) private(…)*" after an assignment statement.

The meta-program is mainly focused on the realization of parallelism and maintained separately from the original sequential code. Whenever necessary, the parallelized code can be generated in a different copy, which prevents the sequential code from being polluted with parallel code. The idea of separating the management of the sequential and parallel code can also help to facilitate simultaneous programming of parallel applications where the domain experts can focus on the core logic of the application while the parallel programmers concentrate on the realization of parallelism [1].

4 Extending Spot for Specifying Program Transformation in C

MOP facilities are straightforward with respect to expressing the design intent of program transformation, compared with the APIs of the underlying transformation engine, which involves much manipulation of ASTs. However, there is also a steep learning curve for library developers when attempting to understand the idea of a MOP and to use the APIs provided by MOPs. In addition, it is usually the case that meta-programs are created to serve as a library for the purpose of enabling certain types of code transformation. Conflicts very likely occur when the functionality provided by a library can no longer satisfy the needs of application programmers. It will be a great benefit for programmers if there is a simpler way to tailor existing libraries to meet their new needs or ideally even build a new library, without having to learn how to program with MOPs.

In order to simplify use of the OpenC MOP, we investigated techniques of code generation with a DSL. To relieve developers from the burden of programming with MOP APIs, we have created a DSL, named SPOT [28], which works on top of OpenC and provides a higher level of abstraction for expressing program transformations. The design goal is to provide language constructs that allow developers to perform direct manipulation on program entities and hide the accidental complexities of using OpenC and ROSE.

SPOT was originally designed to simplify the usage of OpenFortran [27] by raising the abstraction level of program transformation. In this paper, we extended SPOT to make it applicable to specifying program transformations for C. SPOT provides notations and built-in functions for systematic change of a language entity (e.g., adding, updating, or deleting a statement) to model the process of code modification, which makes it readily extensible by adding new language elements to support a new general-purpose programming language (GPL). In the following subsections, we first briefly introduce SPOT and then focus on explaining its extension in order to accommodate OpenC.

4.1 An Example SPOT Program

Figure 6 demonstrates example SPOT code with the basic structure and language constructs to specify code changes in C programs. The code adds a function call to *printInt* after every

```
Transformer PrintResult{
  Within(Function *){
   Statement %stmt=getStatementAssignment();
   IF($stmt.varName=="varName"){
     AddCallStatement(After, $stmt, printInt, "varName", $stmt.assignValue);
   }
  }
}
IncludeCode{
  void printInt(char* varName, int val)
  {
    printf("%s=%d\n", varName, val);
  }
}
```

Figure 6: An example program coded in SPOT

assignment statement whose left-hand side is the variable with the name *varName.* As indicated by the code snippet, a typical SPOT program starts with a keyword "*Transformer*," followed by a user-defined name, "*PrintResult*" in this case, which will be used as the file name of the generated *.cpp* file. A transformer is usually composed of one or more scope blocks where action statements, nested scope blocks or condition blocks are included. As shown in Figure 6, we define a scope block from line 2 to line 7. The wildcard feature is also supported to translate source code in multiple locations with similar scenarios. For instance "*Within(Function *)*" indicates that the following translation would be performed for all function definitions in the current code where "*" acts as a wildcard. Line 3 defines a variable named "*stmt*" with a percent sign that serves as the handler for a set of assignment statements. Lines 4 to 6 define a condition block with the keyword "*IF*." If the left-hand side in an assignment statement is the variable *varName*, line 5 adds a line of code that calls "*printInt(…)*" after the assignment statement. The "*$*" sign is used together with a user-defined variable to reference any element in the list. For example "*$stmt*" in this example iterates all elements held by the handler "*%stmt.*" As indicated by line 2 in the example, location and scope information is expressed in a manner similar to an aspect in AspectJ [13].

The including block in lines 9 to 14 is optional and provides code needed by the transformer. The functions or variables defined within an *Include* block will be directly inserted into the beginning of the current file and before the first function definition, unless otherwise specified. The developers are expected to use this section to implement helper code used by transformers in the same code file. In Figure 6, all keywords are highlighted in bold in the example code.

4.2 The Design of SPOT for OpenC

To raise the level of abstraction for simplifying the usage of a MOP like OpenC, high-level programming entities (e.g., files, functions, structs, variables and statements) are used in SPOT language constructs. Built-in functions are provided to allow systematic actions for programming entities, such as add, delete and update. The excerpt of built-in constructs and APIs is listed in Table 2.

A benefit of SPOT is that it supports string-based translation. Developers are allowed to embed C code in a SPOT program. For example, in Figure 6, line 5 can be replaced with "*AddStatement(After, $stmt, "printInt("var Name", varName)")*" to achieve the same effect of adding a function-call statement after the statement indicated by *$stmt*, where the last parameter "*printInt("varName", varName)*" is actually a C statement. In addition, a real C statement can also be used as the parameter in "*GetStatement("stmt")*" to obtain its handler. For instance, as in "*Statement %st=GetStatement("result=a+b"),*" all statements containing "*result=a+b*" within the current scope are matched and their handlers are put into the list represented by "*st*." One thing that needs to be noted is that all embedded C code should be contained within double quotes.

One side effect of using C statements to match possible translation points occurs if the source code to be transformed has been modified, (e.g., *a* has been renamed to *d as* in "*result=d+b*"). In such a case, the transformer will skip this translation point. Another possible scenario is that instead of matching an exact C statement, the transformer would like to match a pattern, for instance, matching all assignment statements with the right-hand side being a plus expression. In order to overcome this drawback and to support the desired feature, we allow developers to define a pattern with special literals *$var1, $var2, $var3*… that can be used to substitute for real expressions in a C statement. The pattern that matches all assignment statements with their right-hand side being a plus expression can be depicted as "*$var1=$var2+$var3.*"

4.3The Implementation of SPOT for OpenC

Figure 7 shows the transformation process after integrating SPOT with OpenC. A SPOT program represents desired translation tasks specified directly with built-in constructs by developers for source code written in C. A code generator is used to automate the translation from the SPOT program to C++ meta-level transformation code. The MOP is responsible for carrying out the specified transformations on source code in C with the assistance of the low-level transformation engine ROSE.

The main purpose of the code generator is to translate a SPOT program to the corresponding C++ meta-level code through code generation. As shown in Figure 8, the code generator consists of a parser that is able to recognize the syntax of both SPOT and C and to build an AST for the recognized program, and a template engine that is used to generate C++ code from traversing the AST. The parser is generated with ANTLR [16] from the grammar of SPOT and C expressed in Extended Backus-Naur Form (EBNF). We have chosen ANTLR because the code generator needs the grammar of C for recognizing C source code. A free C grammar for ANTLR is available for use with little adaptation. To implement the generator, we combined the SPOT grammar with the C grammar. For each rule in the grammar we use annotations to direct ANTLR to build ASTs. The annotations indicate which tokens are to be treated as the root of a sub-tree and which are leaves. We have also implemented a tree grammar, the rules of which match desired sub-trees and map them to the output models. The output models used in our code generator are built with StringTemplate [16], a template engine for generating formatted text output. To support string-based transformation, for the same rule in the tree grammar which matches a statement or a construct, two different types of output models (i.e., two different implementations in the meta-level code) are provided to either locate a place for code translation or to add new language constructs in the base-level code.

When programming with SPOT, developers can be more focused on their design intention of transformations with constructs and actions provided. The underlying generation and translation are performed in a transparent manner. Moreover, SPOT provides a mechanism for developers to specify the translation scope and to pick up a point of translation using a

Table 2: Overview of SPOT syntax and semantics for OpenC

Language Constructs		
Virtual Constructs	**Project**	project-wide transformation
	File	file-wide transformation
User Defined Type	**Struct**	Indicate struct definition
	Union	Indicate union definition
Basic Constructs	**Function**	Indicate function definition
	FunctionCall	Indicate function call expression
	VariableRead	Indicate reading a variable
	VariableWrite	Indicate writing a variable
	VariableDecl	Indicate declaring a variable
	Statement*	Indicate different types of statements
Keywords for Scope Block		
Within(para*)	Get the scope of transformation. Supported scopes include a project, a file, a function, a struct, a union, and statements implying a scope, e.g. if-else statement, for-loop statement	
Before(para)/**Before**	Perform transformation before an entity	
After(para)/**After**	Perform transformation after an entity	
Keywords for Control Flow		
IF(*expr*) **ELSE**	Proceed based on the value of *expr*	
FORALL(**Construct** *name*)	List all constructs specified with *name*	
Primary Actions		
Function	**RenameFunction**(oldName, newName)	
	FindFunctionCall(funName)	
Variable	**AddVariable**(type, name, intialValue)	
	DeleteVariable(name)	
	RenameVariable(oldName, newName)	
	FindVariableRead(name)	
	FindVariableWrite(name)	
Statement	**AddStatement**("*stmt*")/ **AddStatement**(loc, targetStmt ,"*stmt*")	
	AddCallStatement(loc, targetStmt, funName, parameterList)	
	DeleteStatement("*stmt*")/ **DeleteStatement**(loc, targetStmt, "*stmt*")	
Auxiliary Functionality		
Retrieve Functions	**Variable** v = **GetVariableDecl(**name**)**	
	Function f = **GetFunctionDef**(name)	
	Struct s = **GetStructDef**(name)	
	Statement*Type* **%**st **= getStatement*Type*()**	
	Statement %st **= getStatement(**"*stmt*"**)** **Statement** st **= getStatement(**lineNumber**)** //used in a file **Statement %**st **= getStatement(**pattern**)**	
	VariableWrite %vw**=getVariableWrite**(varName)	
	VariableRead %vr**=getVariableRead**(varName)	
Include Block	**IncludeCode** { *source code in c*}	
	IncludeCode { *source code in c*} **into** fileName	

Note:
1. **para** can be a construct name or an expression (*expr*) or statement (*stmt*)
2. ***stmt*** indicates a C statement (within double quotes) or a pattern described with ***%var*** substituting for real expressions within a statement
3. ***expr*** indicates an actual C expression or a pattern described with ***%var***
4. ***%var*** is a user-defined variable representing a collection of entities, using ***$var*** to access an element in the collection

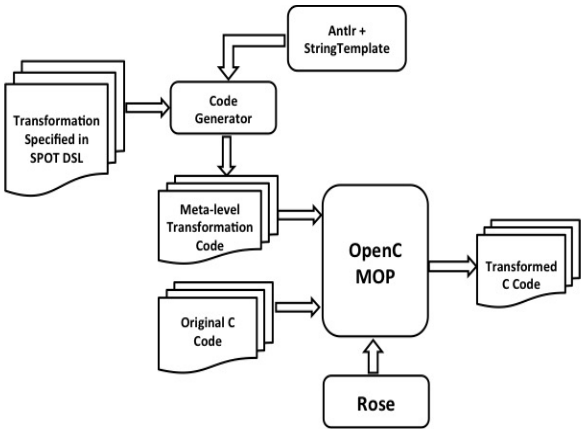


Figure 7: Overview of the transformation process with SPOT and OpenC

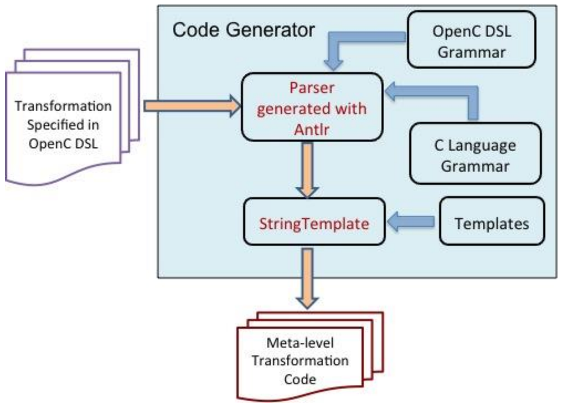

Code Generator
OpenC DSL Grammar
Transformation Specified in OpenC DSL
Parser generated with Antlr
C Language Grammar
StringTemplate
Templates
Meta-level Transformation Code

Figure 8: The implementation structure of the Code Generator

specific construct name or a wildcard to match multiple points. Therefore, no annotation to the source code is necessary to use libraries developed in SPOT, which makes the solution non-intrusive because translations are performed on a generated copy of the original code and the original code is kept intact.

4.4 Applications of SPOT for OpenC

The same transformations specified by the profiling library in Section 3 can be achieved with SPOT in a more straightforward way. Figure 9 demonstrates how to address the translation challenge with constructs provided by SPOT. The Code Generator in Figure 7 is responsible for transforming the DSL code to the meta-level implementation in SPOT (shown in Figure 4). The generated code will be saved in *Profiling.cpp* whose name is from the Transformer's name specified in line 1. Line 2 uses a wildcard to make the transformation applicable to all source files. Line 3 inserts an include directive (i.e., "*#include profiling.h*") at the beginning of the current file. Line 4 loops over all function definitions within the current file by calling *FORALL(…)*. From line 5 to line 8 the code matches all statements containing a function call and then adds two new function calls before and after the statement by invoking *AddCallStatement(…)*, where the first argument indicates the relative location (*Before* or *After*), the second one the handler of the statement matched, the third one the function name to be added and all the rest are interpreted as the parameters passed to the added function call. In the code, all built-in constructs are highlighted in bold. Figure 10 shows the corresponding SPOT code that can be translated into the meta-program as shown in Figure 5, where *insertPragma* is called to instrument OpenMP directives into the desired places.

5 Related Work

Many development concepts of MOPs occurred in the context of the Common Lisp Object System (CLOS) [4]. The initial design objective of the MOP for CLOS was to allow object-oriented Lisp to meet the ever-increasing user demands for extension. As a result, the MOP concept itself became a powerful tool that can also be used to solve many different problems emerging in other high-level languages. OpenC++ was proposed by Chiba to bring the power of meta-programming to C++ [5]. The design goal of OpenC++ was to enable client users to develop customized language extensions or compiler optimizations through simple annotations. However, it is still a challenge for most application developers to program with the concept of meta-programming.

OpenJava was designed as a MOP for Java by Tatsubori and Chiba [23]. It is a reflective system that is able to provide both structural and behavioral reflection. Instead of using an AST as the main data structure to perform translation. OpenJava exploits a more advanced macro system that is able to hold the logical and contextual data. Roychoudhury et al. [19] presented the implementation of an aspect weaver for supporting AOP in Fortran using a source transformation engine named DMS [2].

There exist many systems that can be used to perform program transformations. Though powerful in supporting specified program transformations, it is quite a challenge for application developers to learn and use most program transformation systems. In contrast, our work strives to match more closely to the application developers' comprehension of program transformation by allowing direct manipulation of concepts in the core language. We believe that our approach using OpenC and SPOT is easier to use and has a lower learning curve compared to use of program transformation systems.

Many existing transformation systems, such as ASF+SDF [25] and Turing eXtender Language (TXL) [6], mainly depend on pattern-based rewrite rules to locate translation points and to specify particular transformations. OpenC provides constructs for defining arbitrary transformations. For instance, application developers are allowed to identify transformation places with arbitrary control flow such as condition (*IF-ELSE*) and loop (*FORALL*) (recursive is not necessary) and with user-defined variables (single or a list of variables) as handlers to represent particular entities, and perform translation by directly invoking many built-in operations, for instance *addEntity*, *replaceEntity* and *deleteEntity* where Entity may refer to any program entities

```
Transformer Profiling{
 Within(File *){
   AddIncludeStatement(profiling.h);
   FORALL(Function %fun){
     FORALL(FunctionCall %funCall){
       AddCallStatement(Before, $funCall.statement, profiling,$fun.funName+”:”+$funCall.funName);
       AddCallStatement(After, $funCall.statement, profiling, $fun.funName+”:”+$funCall.funName);
     }
   }
 }
}
```

Figure 9: The profiling library specified in SPOT

```
Transformer ParaMolecularDynamics{
  Within(Function compute){
     Statement forStmt = getForStatement(“k,0,np,1” , 1);
     insertPragma(Before, targetStmt, ”for reduction (+ : pe, ke)”);
     Statement assignStmt = getStatement(“ke = 0.0” , 1);
     insertPragma(After, assignStmt, “parallel shared (f, nd,…) private ( I, j, k…)”);
  }
}
```

Figure 10: SPOT code to parallelize molecular dynamics using OpenMP

of a particular language.

POET [26], a scripting language, was originally developed to perform compiler optimizations for performance tuning. As an extension of the ROSE compiler optimizer, POET can be used to parameterize program transformations so that system performance can be empirically tuned. The features of POET were then enriched to support ad-hoc program translation and code generation of DSLs. However, available transformation libraries are mainly predefined for the purpose of performance turning towards particular code constructs such as loops and matrix manipulation. Developers have to know POET well in order to define their own scripts that are able to read particular input code and return the transformed code. Compared with POET's scheme of parameterization in specifying program transformations, our approach that uses OpenC and SPOT raises the abstraction for program translation, and thus is more aligned with a developer's understanding of program transformations by allowing direct manipulation of language constructs.

6 Conclusions and Future Work

The work described in this paper is focused on a summary of the OpenC framework that brings the power of computational reflection to C with a MOP. With OpenC, source-to-source program transformation libraries can be built and then applied in a transparent way. This can be especially suitable for developing libraries dealing with crosscutting issues like logging, profiling and checkpointing. With traditional approaches, library users are usually forced to learn the specifications on how to use a library's interfaces. However, to use transformation libraries developed with OpenC, the only action required is to attach proper annotations to the source code and the underlying transformations are completely transparent to the developers. It is also convenient to unplug the libraries by simply removing the annotation. The application code is kept intact because translations are performed on a generated copy of the original code.

Although it is more straightforward conceptually to use OpenC to implement libraries than directly using APIs of ROSE to manipulate an AST, we believe that there is a learning curve for library developers to become familiar with the MOP idea. We have created a DSL that can be used on top of a MOP (on a meta-meta-level) to improve the ability to specify program transformations. SPOT developers can use carefully designed language constructs to express transformation tasks in a transparent manner, whereby they do not need to know the details on how the transformations are performed underneath. Not only can the SPOT DSL be used to support AOP in C, it can also be used to specify more fine-grained transformations at more diverse locations.

Currently, we have only extended SPOT to support a limited number of pattern matching actions. Gradually, we will enrich it with more features in order to support additional types of translations. Moreover, SPOT is not limited to Fortran and C, but can also be extended to support other languages since it shows a higher abstraction of program composition.

Our experience shows that the MOP mechanism, as a form of program extension, can be used to address a wide range of problems by facilitating the implementation of source-to-source program translators. There is a lack of infrastructure support for language extension in the way of building a meta-object protocol for an arbitrary language. Therefore, we plan to build a generalized framework, suitable for extending an arbitrary programming language by creating a MOP for the language. The design goal is to allow end-users to specify source-to-source program transformation of any kind via the MOP to existing programs written in the language.

References

[1] R. Arora, P. Bangalore, and M. Mernik, "Tools and Techniques for Non-Invasive Explicit Parallelization," *The Journal of Supercomputing*, 62(3):1583-1608, 2012.

[2] I. D. Baxter, C. Pidgeon, M. Mehlich, "DMS®: Program Transformations for Practical Scalable Software Evolution," *Proceedings of the 26th International Conference on Software Engineering*, pp. 625-634, 2004.

[3] K. H. Bennett and V.T Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proceedings of the International Conference on Software Engineering*, pp.73-87, 2000.

[4] D. Bobrow, R. Gabriel, and J. White, "CLOS in Context —The Shape of the Design Space," A. Paepcke, Editor, *Object-Oriented Programming-The CLOS Perspective*, The MIT Press, Chapter 2, 1993.

[5] S. Chiba, "A Metaobject Protocol for C++," *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pp. 285-299, 1995.

[6] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, 61(3):190-210. 2006.

[7] F. Demers and J. Malenfant, "Reflection in Logic, Functional and Object-Oriented Programming: A Short Comparative Study," *IJCAI Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 95:29-38, 1995.

[8] L. DeMichiel and R. Gabriel, "The Common Lisp Object System: An Overview," *Proceedings of European Conference on Object-Oriented Programming*, pp. 151-170, 1987.

[9] http://people.sc.fsu.edu/~jburkardt/c_src/md_openmp/md_openmp.html, 2016.

[10] http://www.edg.com/index.php?location=c_frontend, 2016.

[11] G. Kiczales, J. Rivieres, and D. Bobrow, *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proceedings of European Conference on Object-Oriented Programming*, pp. 220-242. 1997.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting Started with AspectJ," *Communications of the ACM*, 44(10): 59-65, 2001.

[14] P. Maes, "Concepts and Experiments in Computational Reflection," *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pp. 147-155, 1987.

[15] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface Version 2.0*, November 2000.

[16] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, 2007.

[17] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, Cambridge University Press, 2004

[18] ROSE Compiler Project, http://rosecompiler.org, Last Accessed: February 28, 2016.

[19] S. Roychoudhury, J. Gray, F. Jouault, "A Model-Driven Framework for Aspect Weaver Construction," *In Transactions on Aspect-Oriented Software Development*, VIII:1-45, 2011.

[20] L. M. Scott, *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2006.

[21] B. Smith, "Reflection and Semantics in a Procedural Language," Tech. Report 272, MIT, 1982.

[22] D. Spinellis, "Rational Metaprogramming," *IEEE Software*, 25(1):78-79, 2008

[23] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano, "OpenJava: A Class-Based Macro System for Java," *Reflection and Software Engineering*, pp. 117-133, 1999.

[24] TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[25] M. G.van den Brand, A.van Deursen, J. Heering, H. A. De Jong, M. de Jonge, and Visser, J, "The ASF+ SDF Meta-Environment: A Component-Based Language Development Environment," *Compiler Construction*, pp. 365-370, 2001.

[26] Q. Yi, "POET: A Scripting Language for Applying Parameterized Source - to - Source Program Transforma-tions," *Software: Practice and Experience*, *42*(6):675-706, 2012.

[27] S. Yue and J. Gray, "OpenFortran: Extending Fortran with Meta-Programming," *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pp. 1-7, 2013.

[28] S. Yue and J. Gray, "SPOT: A DSL for Extending FORTRAN Programs with Meta-Programming," *Advances in Software Engineering*, 2014:1-23, 2014.

Songqing Yue is an Assistant Professor in the Department of Mathematics and Computer Science at the University of Central Missouri. He has several years of software development experience in industry and his research interests mainly focus on Programming Language, High Performance Computing, and Model Driven Development.

Jeff Gray is a Professor in the Department of Computer Science at the University of Alabama. His research interests include topics in software engineering, with a special focus on model-driven engineering and domain-specific languages. He also has interests in computer science education and K-12 outreach. More details about his work can be found at http://gray.cs.ua.edu.

Investigating the Relationships between Use Cases Attributes and Source Code Size

William Flageol[*], Mourad Badri[*], and Linda Badri[*]
University of Quebec, Trois-Rivières, Quebec, CANADA

Abstract

Software development is a time and resource consuming process. It is, therefore, important to estimate as soon as possible the effort required to develop software, so that activities can be planned and resources can be optimally allocated. Many software development effort estimation methods have been proposed in the literature. Most of them include software "size" as an important parameter. This study aims at investigating empirically the relationships between use cases attributes and source code size. The research question we wanted to address is to explore the potential of predicting the source code size from use cases attributes only, regardless of technical or environmental factors, which may be difficult to objectively measure. We used in the study four metrics to quantify various use cases attributes and four source code size metrics. In order to investigate the relationships between use cases and source code size metrics, we used three correlation analysis techniques and a clustering technique. The use case metrics have been compared, in terms of relationships with the source code size metrics, to the well-known Use Case Points method. An empirical study, using data collected from five Java open source projects, is reported in the paper. Results provide evidence that a subset of the use case metrics suite is better correlated to the source code size metrics than the Use Case Points values.

Keywords: Software development effort, use cases, source code size, metrics, use case points, relationships.

1 Introduction

Software development is a time and resource consuming process. It is, therefore, absolutely necessary to estimate as soon as possible, ideally in the early stages of the software development lifecycle, the effort required to develop software. In this way, activities can be planned and resources can be optimally allocated. Predicting early the software development effort is, in fact, one of the key aspects of successful software development management. Many software development effort estimation methods (models) have been proposed in the literature. Most of them include software "size" as an important parameter.

Typical inputs available at early stages of the software development lifecycle are functional requirements, which describe what a software system is expected to do. In this paper, we focus on software development effort prediction for object-oriented (OO) software development, which is used extensively in the industrial projects. Use cases, available relatively early during the software development lifecycle, are used to describe the functional requirements of a software system [5]. Use cases have gained popularity and have been widely used for many years. Consequently, effort prediction (estimation) based on use cases attributes has also gained popularity. Many use cases based methods have, indeed, been proposed in the literature for software size measurement and effort estimation [4, 6, 8-9, 12-14]. These methods use, in fact, different use case attributes as inputs, including actors and use cases ranking, use cases points, normal and exceptional scenarios and various technical and environmental factors. The technical factors are related to non-functional requirements on the target system while the environmental factors characterize the development team and its environment. These factors have been criticized for not improving the precision of the estimate [12].

In this paper, we investigate empirically the potential of use cases attributes to predict source code size. The medium-term objective of this research is to propose a simple and an alternative method, based on objective measures, simplifying source code size prediction from use cases without compromising the accuracy of the prediction. We used in this study four metrics to quantify different attributes related to size and complexity of use cases, and four metrics to quantify different size attributes of corresponding source code. We used three correlation analysis techniques to investigate the relationships between use cases and source code size metrics. We also used a clustering technique (K-means) to classify use cases in three classes: *complex*, *average* and *simple*. The goal here was to observe if the complexity of the use cases, according to the classification we adopted, is well reflected by the distribution of the source code size metrics. The use cases metrics have been compared to the well-known Use Case Points (UCP) method, which is based on a use cases model. We performed an empirical study using data collected from five Java open source projects. Reported results provide evidence that a subset of the use cases metrics suite is better

[*] Software Engineering Research Laboratory, Department of Mathematics and Computer Science. Email: {William.Flageol, Mourad.Badri, Linda.Badri}@uqtr.ca

correlated to the source code size metrics than the Use Case Points values.

The rest of this paper is organized as follows: Section 2 gives a brief survey of major related work. The use cases metrics are presented in Section 3. Section 4 presents summarily the Use Case Points method. Section 5 presents the empirical study we performed in order to investigate the relationships between the use cases and the source code size metrics. Finally, Section 6 concludes the paper and outlines some future work directions.

2 Related Work

Use cases have been used in many studies to estimate the software development effort (cost) [4, 6, 8-9, 12-14]. Karner [6] introduced a method using use cases model as a basis for estimating software development effort. The method was influenced by the Function Points method. Mohagheghi *et al.* [8] introduced some rule based modifications to the UCP (Use Case Points) method in order to estimate the effort in incremental development. Ochodek et *al*. [12] investigated the constructs of the UCP method, the influence of its components on its accuracy, and explored possible simplifications. The aim was, in fact, to search for potential ways to simplify the effect estimation based on use cases, particularly to reduce the impact of the subjectivity of adjustment factors.

The UCP method has been used in many studies to estimate software development (and testing) effort. The UCP method has been proposed to estimate basically software development effort (Person-Hours) in the early stages of the software development lifecycle. The method has been used in many organizations, and several tools to support calculating UCP have been developed. Many studies and experience reports showed the usefulness of the UCP method for early effort and size estimation based on use cases model. The UCP method suffers, however, from some limitations that affect its accuracy. Indeed, some studies pointed out problems concerning the UCP constructs and the way the method assesses the complexity of actors and uses cases [12]. The main drawback of the method is, in fact, the absence of the graduation when classifying the complexity of use cases and actors. For example, if the number of transactions in a given use case is seven the use case is classified as average. However, if the number of transactions is eight, the use case is classified as complex.

3 Use Case Metrics

Use cases are basically used for capturing and describing functional requirements of a system. Informally, a use case is a collection of related success and failure scenarios that describe actors using a system to support a goal [5, 7]. A use cases model defines the functional scope of the system to be developed, and describes how external actors interact with the software system. The interactions between actors and the software system generate events to the software system, known as input system events, which are usually associated with system operations.

A scenario, also called a use case instance, is a specific sequence of actions and interactions between actors and the system. It is one particular story of using the system, or one path through the use case. We present, in what follows, the four metrics we used to characterize use cases size and complexity [1]. We excluded, in fact, the NIM (*Number of Involved Methods*) metric from the original suite of use cases metrics that we proposed in [1], because this metric defines the total number of methods that are involved in the execution of a use case, which is a metric obtained from design models. Moreover, we extended the suite of use cases metrics by introducing the NT metric.

Number of Involved Classes (NIC): This metric defines the total number of analysis classes participating in a use case, which have been identified during the analysis phase.

Number of External Operations (NEO): This metric defines the total number of system operations associated with the system events related to a use case. These operations are easily identifiable from UML system sequence diagrams. The entire set of system operations, identified during system behavior analysis, defines the public system interface.

Number of Scenarios (NS): This metric gives the total number of different scenarios of a use case. It is related to the cyclomatic complexity of the use case. In our approach, we do not make the distinction (as some approaches in the literature) between a normal scenario and an exceptional one. We consider the total amount of scenarios in each use case. By collecting all possible behavioral sequences, based on requests submitted by the actors to the target system, use cases (and corresponding system sequence diagrams) capture the wide range of possible scenarios.

Number of Transactions (NT): This metric gives the total number of transactions of a use case. A transaction is an event (a set of activities in a use case scenario) that occurs between an actor and the target system, the event being performed entirely or not at all. The complexity of a use case is also determined by the number of transactions. The simplest way to count the number of transactions is to count the number of events included in the flow of events described in a use case (and corresponding system sequence diagram).

4 Use Case Points Method

The Use Case Points (UCP) method has been used in several industrial projects to estimate the software development effort in the early stages of the software development process [2-3, 8, 10-12]. The popularity of this method is basically due to its simplicity and its reduced number of steps. The UCP method, introduced by Karner in 1993 [6], allows estimating the effort in Person/Hours using use cases. The UCP method has also been used for predicting maintenance effort [3] and software testability [9]. We present, in what follows, a summary of the UCP method. For more details see [6, 8, 12]. The UCP estimation model consists of six steps:

Unadjusted Actor Weights: The first step of the UCP method is to assign each actor to one of three complexity classes (*simple*: an actor representing a system that communicates with other actors using API, *average*: a system actor who communicates with the system via a protocol (e.g., HTTP, FTP), or a person who interacts with the system through a terminal console, *complex*: human actor that communicates with the system via a GUI). Each type of complexity is assigned a weight: 1 for *simple*, 2 for *average* and 3 for *complex*. Then, the number of each actor type that the target software includes is calculated. After that, each number is multiplied by a weighting factor. The actors' weight is calculated by adding these values together.

Unadjusted Use Case Weights: In the second step of the UCP method each use case is categorized as *simple*, *average* or *complex*. This categorization is based on the number of steps (transactions) in a use case, including alternative paths. A *simple* use case has 3 or fewer transactions, an *average* use case has 4 to 7 transactions, and a *complex* use case has more than 7 transactions. Here also, the number of use case types that the target system includes is calculated and then each number is multiplied by a weighting factor.

Unadjusted Use Case Points: This step is to calculate the unadjusted use case points by adding the total weight for actors to the total weight of use cases.

Assigning values to technical complexity and environmental factors: These factors [8, 12] include technical factors and environmental factors. Technical complexity factors can have an impact on the complexity of the project (e.g., reuse, security, performance, etc.). Environmental factors are the environment in which the project evolves (e.g., team experience in a particular field, stability requirements, etc.). The technical complexity factor is calculated by multiplying the value of each factor by its weight (Table 1). The environmental factor is also calculated by multiplying the value of each factor by its weight (Table 2). The technical complexity factor is calculated as follows: TCF = 0.6+(0.01*TFactor), where TFactor is the sum of all the weighted factors. The environmental factor is calculated as follows: EF = 1.4+(-0.03*EFactor), where here also the EFactor is the sum of all the weighted environmental factors.

Table 1: Technical factors

Factor	Description	Weight
T1	Distributed system	2.0
T2	Response time/performance objectives	1.0
T3	End-user efficiency	1.0
T4	Internal processing complexity	1.0
T5	Code reusability	1.0
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portability to other platforms	2.0
T9	System maintenance	1.0
T10	Concurrent/ parallel processing	1.0
T11	Security features	1.0
T12	Access for third parties	1.0
T13	Enf user training	1.0

Table 2: Environmental factors

Factor	Description	Weight
E1	Familiarity with development process used	1.5
E2	Application experience	0.5
E3	Object-oriented experience of team	1.0
E4	Lead analyst capability	0.5
E5	Motivation of the team	1.0
E6	Stability of requirements	2.0
E7	Part-time staff	-1.0
E8	Difficult programming language	-1.0

Adjusted Use Case Points: UCP is then calculated as follows: UCP = UUCP*TCF*EF.

Estimated Effort: The estimated effort is obtained by multiplying the specific value (man-hours) by the UCP: EE = UCP*Hours/UCP, where Hours/UCP is a value of man-hours per UCP.

5 Empirical Study

5.1 Goal, Selected Projects and Data Collection

The study aims basically at investigating the relationships between use cases metrics (in comparison to the Use Case Points method) and the source code size metrics. We used in our study different metrics to characterize the size of the source code corresponding to a use case (noted UC_i).

Number Of Classes (NOC): This metric gives the number of classes that are involved in the realization of a use case.

Number Of Attributes (NOA): This metrics gives the total number of attributes of the classes involved in the realization of a use case.

Number Of Methods (NOM): This metrics gives the total number of methods of the classes involved in the realization of a use case.

Source Lines of Code (SLOC): This metric defines the cumulative number of lines of source code related to a use case (all of its scenarios). It is used to indicate the total size, in terms of lines of source code, of the parts of software corresponding to a use case. The number of lines of source code has been widely used in many previous empirical software engineering studies.

We used in our experiments each pair (use case metric, size metric) to explore the relationships between the characteristics of a use case, captured by the use cases metrics, and the size of corresponding source code captured by the size metrics. We conducted an experimental study using data collected from five Java open source projects. The selected case studies are from different domains and developed by different teams. The use case models (and corresponding sequence diagrams) have been collected for each project by reverse engineering the source code of the applications. We also used the available projects' documentation (particularly the user guides). We related to each use case the corresponding source code. For each use case, we calculated the values of the use cases metrics and the global value of the UCP method. For the UCP method, we followed the different steps of the method presented in Section 4. The global value of the UCP method, used in this study, corresponds to the *Adjusted Use Case Points* (step five). We also used the size metrics to quantify the source code corresponding to each use case.

The selected projects are: ATM, NextGen, CommonsIO, CommonsEmail and JODA-Time. The first case study ATM[†] is a simulator system allowing performing basic banking operations (withdrawal, deposit, transfer, balance, etc.). We adapted the case study for our purposes. The second case study NextGen is an extension of the application developed by Larman [7]. The original application has been extended for our purposes. We have added features about accounts receivable management, suppliers, and employees. We also added features to support billing and rental payments by debit and credit. The third case study CommonsIO[‡] is a library of utilities to assist with developing IO functionality. We used only a part of this system that is related to reading, writing and files comparison functionalities. The fourth case study Commons Email[§] aims to provide an API for sending email. It is built on top of the Java Mail API, which it aims to simplify. The fifth case study JODA-Time[**] is the de facto standard library for advanced date and time in Java. It provides a quality replacement for the Java date and time classes. The design supports multiple calendar systems, while still providing a simple API.

Because the main data set was heterogeneous, and in order to have a significant sample of data, we decided to perform our study on the whole data set obtained by grouping the use cases of the five case studies. We have then a total of 46 use cases. Table 3 lists the descriptive statistics for the use cases metrics,

Table 3: Descriptive statistics for the use cases metrics (including UCP) and the source code size metrics

Var.	Ob	Min	Max	Mean	SD
NIC	46	1,00	15,000	3,609	3,102
NEO	46	1,00	36,000	2,696	5,456
NS	46	1,00	32,000	3,587	4,717
NT	46	1,00	41,000	4,739	6,112
UCP	46	4,38	15,130	8,243	3,580
NOC	46	1,00	15,000	3,217	3,112
NOA	46	0,00	41,000	5,283	8,334
NO	46	1,00	58,000	5,000	8,859
SLO	46	4,00	399,000	59,174	66,856

[†] http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

[‡] https://commons.apache.org/proper/commons-exec/

[§] https://commons.apache.org/proper/commons-email/

[**] https://commons.apache.org/proper/commons-io/

the global value of UCP and the source code size metrics.

5.2 Correlation Analysis

In order to investigate the relationships between the use cases metrics (noted UCM_m), including the UCP global value, and the source code size metrics (noted S_i) we performed statistical tests using correlation. The null and alternative hypotheses that our study has tested were:

- H0: There is no significant correlation between a use case metric UCM_m (UCP) and the source code size metric S_i.
- H1: There is a significant correlation between a use case metric UCM_m (UCP) and the source code size metric S_i.

In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between a use case metric UCM_m (UCP) and the source code size metric S_i. For the analysis of the collected data, and in order to test the correlation between a use case metric (UCP) and a size metric S_i, we used three correlation analysis techniques: Pearson, Spearman and Kendall. We used these techniques mainly for completeness. The Pearson r correlation is widely used in statistics to measure the degree of the relationship between linear related variables. The variables should be normally distributed. The Spearman rank correlation is a non-parametric test that is used to measure the degree of association between two variables. Spearman rank correlation test does not assume anything about the distribution of the variables. The Kendall rank correlation coefficient is also a statistic used to measure the association between two variables. It is a measure of rank correlation.

Correlation is a bivariate analysis that measures the strengths of association between two variables. In statistics, the value of the correlation coefficient varies between +1 and -1. A positive correlation is one in which the variables increase together. A negative correlation is one in which one variable increases as the other variable decreases. A correlation of +1 or -1 will arise if the relationship between the variables is exactly linear. A correlation close to zero means that there is no linear relationship between the variables.

We used the XLSTAT[††] software tool to measure the three types of correlations. We applied the typical significance threshold ($\alpha = 0.05$) to decide whether the correlations where significant. For each pair $<UCM_m, S_i>$, we analyzed the collected data set by calculating the (Pearson's, Spearman's and Kendall's) correlation coefficients. Tables 4, 5, and 6 summarize the results of the correlation analysis (respectively Pearson's, Spearman's and Kendall's correlation coefficients). The correlation coefficients that are significant ($\alpha = 0.05$) are set in boldface in the three tables. This means that for the corresponding pairs of metrics there exist a correlation at the 95% confidence level.

The first global observation that we can make from the three tables (Tables 4, 5 and 6), is that there is a significant

[††] http://www.xlstat.com/

Table 4: Pearson's correlation values between the use cases metrics (UCP) and the source code size metrics

Var.	NIC	NEO	NS	NT	UCP	NOC	NOA	NOM	SLOC
NIC	**1**	**0,517**	**0,537**	**0,527**	0,254	**0,962**	**0,569**	**0,735**	**0,655**
NEO	**0,517**	**1**	**0,816**	**0,944**	**0,453**	**0,518**	**0,683**	**0,899**	**0,764**
NS	**0,537**	**0,816**	**1**	**0,930**	**0,549**	**0,569**	**0,616**	**0,852**	**0,807**
NT	**0,527**	**0,944**	**0,930**	**1**	**0,648**	**0,543**	**0,690**	**0,894**	**0,833**
UCP	0,254	**0,453**	**0,549**	**0,648**	**1**	0,289	**0,435**	**0,455**	**0,580**

Table 5: Spearman's correlation values between the use cases metrics (UCP) and the source code size metrics

Var.	NIC	NEO	NS	NT	UCP	NOC	NO	NO	SLOC
NIC	**1**	**0,485**	0,149	0,216	0,232	**0,916**	**0,7**	**0,62**	**0,581**
NEO	**0,485**	**1**	0,087	**0,564**	**0,416**	**0,379**	**0,7**	**0,43**	**0,522**
NS	0,149	0,087	**1**	**0,764**	**0,726**	0,243	0,1	0,25	**0,613**
NT	0,216	**0,564**	**0,764**	**1**	**0,903**	0,280	**0,4**	**0,42**	**0,730**
UCP	0,232	**0,416**	**0,726**	**0,903**	**1**	**0,385**	**0,4**	**0,58**	**0,736**

Table 6: Kendall's correlation values between the use cases metrics (UCP) and the source code size metrics

Var.	NIC	NEO	NS	NT	UCP	NOC	NOA	N	SLOC
NIC	**1**	**0,402**	0,122	0,174	0,185	**0,860**	**0,558**	**0**	**0,440**
NEO	**0,402**	**1**	0,074	**0,500**	**0,341**	**0,322**	**0,600**	**0**	**0,414**
NS	0,122	0,074	**1**	**0,725**	**0,629**	0,198	0,160	**0**	**0,460**
NT	0,174	**0,500**	**0,725**	**1**	**0,784**	**0,223**	**0,385**	**0**	**0,561**
UCP	0,185	**0,341**	**0,629**	**0,784**	**1**	**0,297**	**0,343**	**0**	**0,546**

relationship (the correlation values are in bold face) between all the use cases metrics, including the UCP global value, and the source code size metrics (except for few cases, the corresponding correlation values are not in bold face). According to the obtained results, we can therefore reasonably reject the null hypothesis H0. The other global observations that we can make from these tables are:

- According to the Pearson technique: (1) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric SLOC are respectively NT (0.833), and NS (0.807), which are both much better correlated to the source code size metric SLOC than the UCP metric (0.580). (2) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOC are respectively NIC (0.962), and NS (0.569), which are both much better correlated to the source code size metric NOC than the UCP metric (the correlation is not significant). (3) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOA are respectively, NT (0.690), and NEO (0.683), which are both much better correlated to the source code size metric NOA than the UCP metric (0.435). (4) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOM are respectively, NEO (0.899), and NT (0.894), which are here also both much better correlated to the source code size metric NOM than the UCP metric (0.455).
- According to the Spearman technique: (1) the use cases metrics that are the most correlated (compared to the other metrics) to the source code metric SLOC are respectively, UCP (0.736) and NT (0.730), with almost equal correlation values. (2) the use case metric that is the most correlated (compared to the other metrics) to the source code size metric NOC is NIC (0.916), which is much better correlated to the source code size metric NOC than the UCP metric (0.385). (3) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOA are respectively, NEO (0.705), and NIC (0.701), which are both much better correlated to the source code size metric NOA than the UCP metric (0.423). (4) the use case metric that is the most correlated (compared to the other metrics) to the source code size metric NOM is NIC (0.623), which is much better correlated to the source code size metric NOM than the UCP metric (0.580).
- According to the Kendall technique: (1) the use case metric that is the most correlated (compared to the other metrics) to the source code metric SLOC is NT (0.561), which is better correlated to the source code size metric SLOC than the UCP metric (0.546). (2) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOC are respectively, NIC (0.860), and NEO (0.322), which are much better correlated to the source code size metric NOC than the UCP metric (0.297). (3) the use cases metrics that are the most correlated (compared to the other metrics) to the source code size metric NOA are

respectively, NEO (0.600), and NIC (0.558), which are both much better correlated to the source code size metric NOA than the UCP metric (0.343). (4) the use case metric that is the most correlated (compared to the other metrics) to the source code size metric NOM is NIC (0.529), which is much better correlated to the source code size metric NOM than the UCP metric (0.462).

So, results show clearly that, overall, the use case metrics (at least a subset of the use cases metrics suite) are much better correlated to the source code size metrics than UCP. Moreover, as it can be seen from the three tables, the measures have positive correlation. A positive correlation indicates that one variable (use case metric, UCP) increases as the other variable (source code size metric) increases.

5.3 Use Cases Ranking

In order to deepen our analysis and better understand the relationship between use cases attributes and source code size, we wanted to explore the use of clustering techniques for classifying (ranking) use cases into three categories: *simple*, *average* and *complex*. Clustering provides, indeed, a natural way for identifying clusters of related objects (use cases in our study) based on their similarity (use cases metrics in our study). The resulting clusters (three in our study), are to be built so that use cases within each cluster are more closely related to one another than use cases assigned to different clusters. We wanted, in fact, to investigate if the distribution of the source code size metrics will reflect the complexity level of the corresponding use cases.

Let *UC* be a use case and *P* = {UCM_i} be the set of its properties (use cases metrics). In this paper, we used the *K*-means clustering, which is a method of cluster analysis that aims to partition *n* observations (use cases) into *k* clusters (three in our study) in which each observation belongs to the cluster with the nearest mean. We used here also the XLSTAT software tool, which implements many clustering algorithms. We obtain three clusters of use cases. Tables 7 and 8 give, respectively, the descriptive statistics for the source code size metric SLOC and the UCP values according to the K-means use cases ranking. We used in this section only the source code size metric SLOC as a representative size metric.

Table 7: Descriptive statistics for SLOC according to use cases ranking

Clusters	Obs.	Mean	Variance (n)	SD
Simple	26	31,615	1841,467	42,912
Average	15	81,400	507,840	22,535
Complex	5	135,800	17825,360	133,512

Table 8: Descriptive statistics for UCP according to use cases ranking

Clusters	Obs.	Mean	Variance (n)	SD
Simple	26	5,988	3,467	1,862
Average	15	10,545	8,642	2,940
Complex	5	13,058	5,850	2,419

The first cluster, corresponding to *simple* use cases, includes 26 use cases. The mean value of the corresponding SLOC is 31.615 and the mean value of corresponding UCP values is 5.988. The second cluster, corresponding to *average* use cases, includes 15 use cases. The mean value of the corresponding SLOC is 81.400 and the mean value of corresponding UCP values is 10.545. The third cluster, corresponding to *complex* use cases, includes 5 use cases. The mean value of the corresponding SLOC is 135.800 and the mean value of the corresponding UCP values is 13.058.

Results show clearly that the mean value of the SLOC metric of *complex* use cases is higher than the mean value of the same metric of *average* use cases, which is higher than the mean value of the same metric of *simple* use cases. The descriptive statistics of the source code size metric SLOC reflect properly the ranking in terms of complexity of corresponding use cases. This is also well reflected by the curves of Figure 1. This figure gives the distribution of the use cases metrics, the UCP values, the SLOC values, and the other (NOC, NOA and NOM) source code size metrics, respectively. These results seem to suggest that the more use cases are complex, the more effort is required to develop corresponding source code (in terms of source code size metrics). This issue must, however, be more investigated to draw more general conclusions. The same observations can be made for the UCP values.

5.4 Threats to Validity

The study presented in this paper should be replicated using many other OO software systems in order to draw more general conclusions about the relationships between use cases metrics and source code size, as a partial indicator of the software development effort. Indeed, there are a number of limitations that may affect the results of the study or limit their interpretation and generalization. The achieved results are based on the data set we collected from only five Java open source projects. To perform our study, we grouped the use cases of the five case studies to build our data set. Even if the collected data set is statistically significant, we do not claim that our results can be generalized. In addition, the complexity of use cases is determined in part by the number of transactions. The simplest way to count the number of transactions is to count the number of events included in the flow of events. The number of events may be affected by the style adopted in the description of use cases. The findings in this paper should be viewed as exploratory and indicative rather than conclusive. Results show, at least, that the use cases metrics offer a potential way that could be used in early stages of the software development lifecycle to predict the source code size. Moreover, the study has been performed on simple case studies. It is necessary to replicate the study on large software systems.

6 Conclusions and Future Work

We investigated, in this paper, the relationships between use

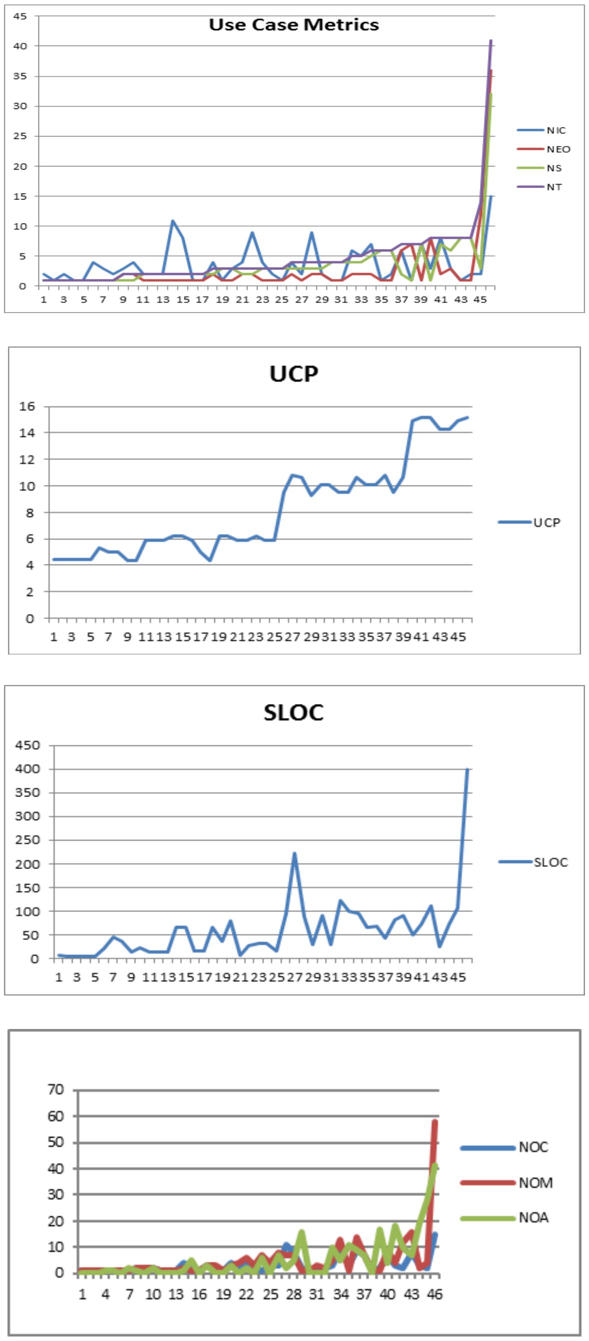


Figure 1: Distribution of the (use cases, UCP, size) metrics

cases metrics and source code size. The use cases metrics are simple and objective measures, which are mathematically valid. The medium-term objective of this research is, in fact, to propose an alternative, a simple and effective method for early predicting of the source code size without compromising the accuracy of the prediction (estimation). The use cases metrics have been compared to the well-known Use Case Points (UCP) method, which is based on the use cases model. We conducted an experimental study using data collected from five Java open source projects. The use case models have been collected for each project. We related to each use case the corresponding source code. We used correlation techniques in order to investigate the relationships between use cases metrics (including UCP), which we proposed in previous work, and the source code size. The goal here was to evaluate the potential of use cases metrics to predict the size of source code. We used three correlation analysis techniques: Pearson, Spearman and Kendall. We observed a significant and in some cases a strong correlation between the use cases metrics (including the UCP) and the final source code size metrics we used in the study.

Overall, results show that a subset of the use cases metrics is better correlated to the source code size metrics than the UCP values, which suggest that these metrics are good measures of size and complexity of use cases, at least in situations where the use cases are structured and detailed at a suitable level, which could be used to predict the size of final source code. This issue is out of the scope of the paper and needs more investigation. We also used a K-means clustering technique in order to classify use cases in three classes: *complex*, *average* and *simple*. We analyzed the distribution of the source code size metrics according to the use cases ranking. We observed that the distribution of the source code size metrics reflects properly the use cases ranking, which seems to suggest that the more use cases are complex, the more effort is required to develop corresponding source code. More investigations, however, are needed to draw more general conclusions. The main limitations of the observations made in this study are basically related to the small size (and number) of the considered set of projects. Moreover, it is important to mention that the proposed use cases metrics are based on only information contained in the use cases model (use cases description, system sequence diagrams, use cases diagram), and there is no need, as in some methods in the literature, to take into account human judgment.

The performed study should be replicated using many other OO software systems in order to draw more general conclusions. There is, in fact, a need for further studies on the relationships between use cases attributes and source code size. The findings in this paper should be viewed as exploratory and indicative rather than conclusive. Results show at least that use cases metrics offer a potential way that can be used in early stages of the software development lifecycle to predict source code size. As future work, we plan to extend the present study by: (1) building development effort prediction models based on use cases attributes, (2) using other methods, such as machine learning methods, in addition to statistical analysis in the construction of the prediction models, (3) comparing our approach to other development effort prediction (estimation) approaches, and finally (4) replicating the study on various OO software systems to be able to give generalized results.

Acknowledgments

This work was partially supported by a NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

References

[1] M. Badri, L. Badri, and W. Flageol, “Predicting the Size of Test Suites from Use Cases: An Empirical Exploration”, H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): *ICTSS 2013*, LNCS 8254:114-132, November 2013.

[2] E. R. Carroll, “Estimating Software Based on Use Case Points”, *OOPSLA’05*, San Diego, California, USA, October 16-20, 2005.

[3] S. Diev, Software Estimation in the Maintenance Context”, *ACM SIGSOFT Software Engineering Notes*, 31(2):1-8, March 2006.

[4] W. Fan, Y. Xiaohu, Z. Xiaochun, and C. Lu, “Extended Use Case Points Method for Software Cost Estimation”, *International Conference on Computational Intelligence and Software Engineering*, 2009.

[5] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1993.

[6] G. Karner, *Resource Estimation for Objectory Projects*, Objective systems, 1993.

[7] C. Larman, *Applying UML and Design Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall, 2004.

[8] P. Mohagheghi, B. Anda, and R. Conradi, “Effort Estimation of Use Cases for Incremental Large-Scale Software Development”, *Proceedings of the International Conference on Software Engineering*, ICSE’05, St. Louis Missouri, USA, pp. 303-311, May 15-21, 2005.

[9] S. Nagheshwaran, “Test Effort Estimation Using Use Case Points”, *Quality Week 2001*, San Francisco, California, USA, 2001.

[10] A. Bou Nassif, L. F. Capretz, and D. Ho, “Software Effort Estimation in the Early Stages of the Software Life Cycle using a Cascade Correlation Neural Network Model”, 13th *ACIS International Conference on Software Engineering, Artificial intelligence, Networking and Parallel/distributed Computing*, IEEE, 2012.

[11] A. Bou Nassif, L. F. Capretz, and D. Ho, “Calibrating Use Case Points”, *ICSE Companion’14*, May 31-June 7, Hyderabad, India - ACM, 2014.

[12] M. Ochodek, J. Nawrocki, and K. Kwarciak, “Simplifying Effort Estimation Based on Use Case Points”, *Information and Software Technology*, 53:200-213, 2011.

[13] G. Robiolo, and R. Orosco, “Employing use Cases to Early Estimate Effort with Simpler Metrics”, *Innovations in Systems and Software Engineering*, 4:31-43, 2008.

[14] G. Robiolo, C. Badano, and R. Orosco, “Transactions and Paths: Two use Case Based Metrics which Improve the Early Effort Estimation”, *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, pp. 422-425, 2009.

William Flageol is a student of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He finished his master in computer science (software engineering) at the University of Quebec at Trois-Rivières. His main areas of interest include object-oriented development, use cases based predictions, software quality attributes, test-driven development, as well as various topics of software engineering.

Mourad Badri is a full professor of computer science (software engineering) at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. His main areas of interest include object, aspect and agent oriented software engineering, software quality attributes, software quality assurance, program analysis, software maintenance and evolution, and aspect mining and refactoring.

Linda Badri is a full professor of computer science (software engineering) at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. Her main areas of interest include object and aspect-oriented software engineering, software quality attributes, web engineering, change impact analysis, regression testing, and software maintenance.

Instructions for Authors

The International Journal of Computers and Their Applications is published multiple times a year with the purpose of providing a forum for state-of-the-art developments and research in the theory and design of computers, as well as current innovative activities in the applications of computers. In contrast to other journals, this journal focuses on emerging computer technologies with emphasis on the applicability to real world problems. Current areas of particular interest include, but are not limited to: architecture, networks, intelligent systems, parallel and distributed computing, software and information engineering, and computer applications (e.g., engineering, medicine, business, education, etc.). All papers are subject to peer review before selection.

A. Procedure for Submission of a Technical Paper for Consideration

1. Email your manuscript to the Editor-in-Chief, Dr. Fred Harris, Jr., Fred.Harris@cse.unr.edu.

2. Illustrations should be high quality (originals unnecessary).

3. Enclose a separate page (or include in the email message) the preferred author and address for correspondence. Also, please include email, telephone, and fax information should further contact be needed.

B. Manuscript Style:

1. The text should be **double-spaced** (12 point or larger), **single column** and **single-sided** on 8.5 X 11 inch pages.
2. An informative abstract of 100-250 words should be provided.
3. At least 5 keywords following the abstract describing the paper topics.
4. References (alphabetized by first author) should appear at the end of the paper, as follows: author(s), first initials followed by last name, title in quotation marks, periodical, volume, inclusive page numbers, month and year.
5. Figures should be captioned and referenced.

C. Submission of Accepted Manuscripts

1. The final complete paper (with abstract, figures, tables, and keywords) satisfying Section B above in **MS Word format** should be submitted to the Editor-in-Chief.
2. The submission may be on a CD/DVD or as an email attachment(s) . **The following electronic files should be included:**

   - Paper text (required).
   - Bios (required for each author). Integrate at the end of the paper.
   - Author Photos (jpeg files are required by the printer, these also can be integrated into your paper).
   - Figures, Tables, Illustrations. These may be integrated into the paper text file or provided separately (jpeg, MS Word, PowerPoint, eps).

3. Specify on the CD/DVD label or in the email the word processor and version used, along with the title of the paper.

4. Authors are asked to sign an ISCA copyright form (http://www.isca-hq.org/j-copyright.htm), indicating that they are transferring the copyright to ISCA or declaring the work to be government-sponsored work in the public domain. Also, letters of permission for inclusion of non-original materials are required.

Publication Charges

After a manuscript has been accepted for publication, the contact author will be invoiced for publication charges of **$50.00 USD** per page (in the final IJCA two-column format) to cover part of the cost of publication. For ISCA members, $100 of publication charges will be waived if requested.

January 2014