Possible Organization for Writing a Thesis including a LaTeXFramework and Examples

by

A Graduate Advisor
B.Sc., University of WhoKnowsWhere, 2053
M.Sc., University of AnotherOne, 2054

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Whichever

© Graduate Advisor, 2008
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Possible Organization for Writing a Thesis including a LaTeXFramework and Examples

by

A Graduate Advisor
B.Sc., University of WhoKnowsWhere, 2053
M.Sc., University of AnotherOne, 2054

Supervisory Committee

Dr. R. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

Supervisory Committee

Dr. R. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

ABSTRACT

This document is a possible Latex framework for a thesis or dissertation at UVic. It should work in the Windows, Mac and Unix environments. The content is based on the experience of one supervisor and graduate advisor. It explains the organization that can help write a thesis, especially in a scientific environment where the research contains experimental results as well. There is no claim that this is the *best* or *only* way to structure such a document. Yet in the majority of cases it serves extremely well as a sound basis which can be customized according to the requirements of the members of the supervisory committee and the topic of research. Additionally some examples on using LaTeXare included as a bonus for beginners.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my cat, Star Trek, and the weather, for supporting me in the low moments.

Supervisor Main, for mentoring, support, encouragement, and patience.

Grant Organization Name, for funding me with a Scholarship.

I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.

Edith Wharton

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

1.1 How to Start an Introduction

1.2 Is a Review of All Previous Work Necessary Here?

Chapter 2

The Problem

While the research problems have been briefly outlined in Chapter 1, this chapter will focus on the underlying studies which motivated the research of this thesis as well as give a more full and rich description of the problem being solved.

In this chapter, two studies will be presented that I conducted which motivated, and gave insights into, the final research goals of this thesis. The first study entitled “Failure Inducing Developer Pairs” (Section 2.1), focuses on the prediction of software failures through identifying indirect conflicts of developers linked by their software modules. This study found that certain pairs of developers when linked through indirect code changes are more prone to software failures than others. The ideas of developer pairs linked in indirect conflicts will be useful for the further development of indirect conflict tools as it shows that a human factor is present and may be used to help resolve such issues.

The second study, “Awareness with Impact” ((Section 2.2)), takes the notion of developer pairs in indirect conflicts learned from Study 1, and adds in source code change detection in order to create an awareness notification system for developers called *Impact*. *Impact* was designed to a developer to any source code changes preformed by another developers when the two are linked in a technical dependency through a developer pair. *Impact* utilized a non-obtrusive RSS style feed for notifications for visualization. While *Impact* showed some promise through its user evaluation, it ultimately suffered the fate of information overload as was seen in other indirect conflict tools [20, 23, 25].

2.1 Study 1: Failure Inducing Developer Pairs

Technical dependencies in a project can be used to predict success or failure in builds or code changes [19, 27]. However, most research in this area is based on identifying central modules inside a large code base which are likely to cause software failures or detecting frequently changed code that can be associated with previous failures [14]. This module-based method also results in predictions at the file or binary level of software development as opposed to a code change level and often lack the ability to provide recommendations for improved coordination other than test focus.

With the power of technical dependencies in predicting software failures, the question I investigated in this study was : "*Is it possible to identify pairs of developers whose technical dependencies in code changes statistically relate to bugs?*"

This study explains the approach used to locate these pairs of developers in developer networks. The process utilizes code changes and the call hierarchies effected to find patterns of developer relationships in successful and failed code changes. As it will be seen, we found 27 statistically significant failure inducing developer pairs. These developer relationships can also be used to promote the idea of leveraging socio-technical congruence, a measure of coordination compared to technical dependencies amongst stakeholders, to provide coordination recommendations.

2.1.1 Technical Approach

Extracting Technical Networks

The basis of this approach is to create a technical network of developers based on method ownership and those methods' call hierarchies effected by code changes. These networks will provide dependency edges between contributors caused by code changes which may be identified as possible failure inducing pairings (Figure 2.1). To achieve this goal, developer owners of methods, method call hierarchies (technical dependencies) and code change effects on these hierarchies must be identified. This approach is described in detail by illustrating its application to mining the data in a Git repository although it can be used with any software repository.

To determine which developers own which methods at a given code change, the Git repository is queried. Git stores developers of a file per line, which was used to extrapolate a percentage of ownership given a method inside a file. If developer A has written 6/10 lines of method foo, then developer A owns 60% of said method.

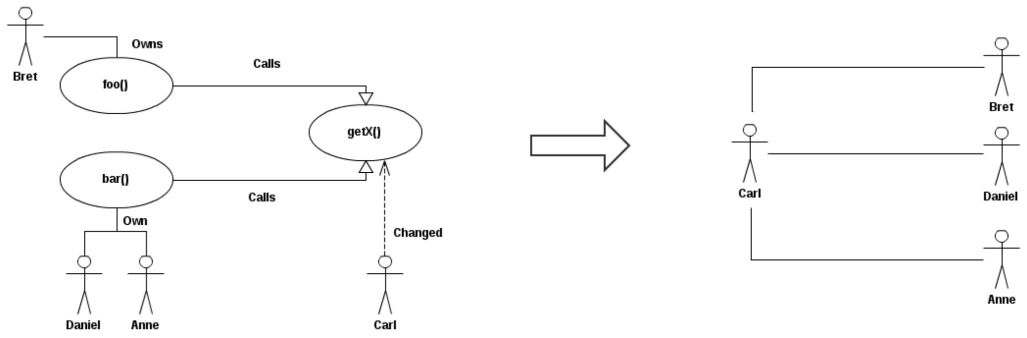


Figure 2.1: A technical network for a code change. Carl has changed method getX() which is being called by Bret's method foo() as well as Daniel and Anne's method bar().

A method call graph is then constructed to extract method call hierarchies in a project at a given code change. Unlike other approaches such as Bodden's et al. [3] of using byte code and whole projects, call graphs are built directly from source code files inside of a code change, which does not have the assumptions of being able to compile or have access to all project files. It is important to not require project compilation at each code change because it is an expensive operation as well as code change effects may cause the project to be unable to compile. Using source files also allowed an update to the call graph with changed files as opposed to completely rebuilding at every code change. This creates a rolling call graph which is used to show method hierarchy at each code change inside a project opposed to a static project view. As some method invocations may only be determined at run time, all possible method invocations are considered for these types of method calls while constructing the call graph.

The code change effect, if any, to the call hierarchy is now found. The Git software repository is used to determine what changes were made to any give file inside a code change. Specifically, methods modified by a code change are searched for. The call graph is then used to determine which methods call those that have been changed, which gives the code change technical dependencies.

These procedures result in a technical network based on contributor method ownership inside a call hierarchy effected by a code change (Figure 2.1 left hand side). The network is then simplified by only using edges between developers, since I am only interested in discovering the failure inducing edges between developers and not the methods themselves (Figure 2.1 right hand side). This is the final technical network.

Identifying Failure Inducing Developer Pairs

To identify failure inducing developer pairs (edges) inside technical networks, edges in relation to discovered code change failures are now analysed. To determine whether a code change was a success or failure (introduce a software failure), the approach of Sliwerski et al. [24] is used. The following steps are then taken:

1. Identify all possible edges from the technical networks.
2. For each edge, count occurrences in technical networks of failed code changes.
3. For each edge, count occurrences in technical networks of successful code changes.
4. Determine if the edge is related to success or failure.

To determine an edge's relation to success or failure, the value FI (failure index) which represents the normalized chance of a code change failure in the presence of the edge, is created.

$$\text{FI} = \frac{\text{edge}_{failed}/\text{total}_{failed}}{\text{edge}_{failed}/\text{total}_{failed} + \text{edge}_{success}/\text{total}_{success}} \tag{2.1}$$

Developer pairs with the highest FI value are said to be failure inducing structures inside a project. These edges are stored in Table 2.1. A Fisher Exact Value test is also preformed on edge appearance in successful and failed code changes, and non-appearance in successful and failed code changes to only consider statistically significant edges (Table 2.1's p-value).

2.1.2 Results

To illustrate the use of the approach, I conducted a case study of the Hibernate-ORM project, an open source Java application hosted on GitHub[1] with issue tracking performed by Jira[2].

This project was chosen because the tool created only handles Java code and it is written in Java for all internal structures and control flow and uses Git for version control. Hibernate-ORM also uses issue tracking software which is needed for determining code change success or failure [24].

[1]https://github.com/
[2]http://www.atlassian.com/software/jira/overview

In Hibernate-ORM, 27 statistically significant failure inducing developer pairs (FI value of 0.5 or higher) were found out of a total of 46 statistically significant pairs that existed over the project's lifetime. The pairings are ranked by their respective FI values (Table 2.1).

Pair	Successful	Failed	FI	P-Value
(Daniel, Anne)	0	14	1.0000	0.0001249
(Carl, Bret)	1	12	0.9190	0.003468
(Emily, Frank)	1	9	0.8948	0.02165

Table 2.1: Top 3 failure inducing developer pairs found.

2.1.3 Conclusions of Study

Technical dependencies are often used to predict software failures in large software system [14, 19, 27]. This study has presented a method for detecting failure inducing pairs of developers inside of technical networks based on code changes. These developer pairs can be used in the prediction of future bugs as well as provide coordination recommendations for developers within a project.

This study however, did not consider the technical dependencies themselves to be the root cause of the software failures. This study focused purely on developer ownership of software methods and the dependencies between developers as the possible root cause of the failures. To study this root cause futher, a study of indirect conflicts and their relationship to developer code ownership will be conducted.

2.2 Study 2: Awareness with Impact

In response to Study 1, a second investigation was conducted. Study 1 revealed that pairs of developers can be used around technical dependencies in order to predict bugs. The natural follow up to these findings was to conduct a study of indirect conflicts surrounding these developer pairs that are involved in source code changes. These indirect conflicts were primarily studies through the notion of task awareness.

Tools have been created to attempt to solve task awareness related issues with some success [2,12,21,26]. These tools have been designed to solve task awareness related issues at the direct conflict level. Examples of direct conflict awareness include knowing when two or more developers are editing same artifact, finding expert knowledge of a particular file, and knowing which developers are working on which files. On the other hand, task awareness related issues at the indirect conflict level have also been studied, with many tools being produced [1,20,23,25]. Examples of indirect conflict awareness include having one's own code effected by another developer's source code change or finding out who might be indirectly effected by one's own code change. Previous interviews and surveys conducted with software developers have shown a pattern that developers of a software project view indirect conflict awareness as a high priority issue in their development [1,5, 9,22].

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis or program slicing [**?**] must be performed in order to find relationships between technical objects which are harmed by changes. While some awareness tools have been created with these indirect conflicts primarily in mind [1,25], most have only created an exploratory environment which is used by developers to solve conflicts which may arise [23]. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [20] has started to work directly on solving indirect conflicts, however, these products are not designed to handle internal structures of technical objects.

In this study, I report on research into supporting developer pairs in indirect conflicts and present the design, implementation, and future evaluation of the tool *Impact*, a web based tool that aims at detecting indirect conflicts among developers and notifying the appropriate members involved in these conflicts. By leveraging technical relationships inherent of software projects with method call graphs [16] as well as detecting changes to these technical relationship through software configuration management (SCM) systems, *Impact* is able to detect indirect conflicts as well as alert developers involved in such conflicts in task awareness while limiting information overload by using design by contract [17] solutions to method design. While this study outlines *Impact's* specific implementation, its design is rather generic and can be implemented in similar indirect conflict awareness tools.

2.2.1 Impact

This section will proceed to give an outlined detail of *Impact* in both its design and implementation. The design of *Impact* was created to be a generic construct which can be applied to other indirect conflict awareness tools while the implementation is specific to the technical goals of *Impact*.

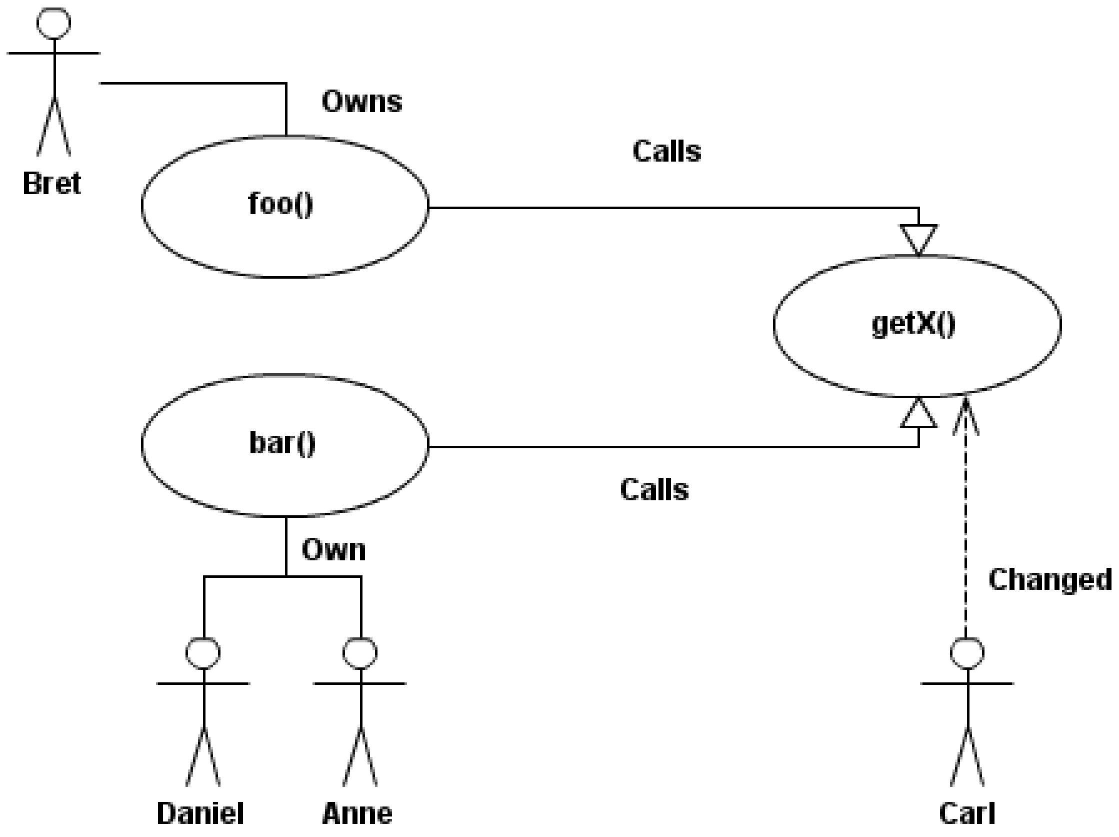


Figure 2.2: Technical object directed graph with ownership

Design

Compared to tool design for direct conflicts, the major concern of indirect conflict tools is to relate technical objects to one another with a "uses" relationship. To say that object 1 uses object 2 is to infer a technical relationship between the two objects which can be used in part to detect indirect conflict that arise from modifying object 2. This kind of relationship is modeled based on directed graphs [11]. Each technical object is represented by node while each "uses" relationship is represented by a directed edge. This representation is used to indicate all indirect relationships within a software project.

While technical object relationships form the basis of indirect conflicts, communication between developers is my ultimate goal of resolving such conflicts (as was seen in Study 1). This being the case, developer ownership must be placed on the identified technical objects. With this ownership, we now infer relationships among developers based on their technical objects "uses" relationship. Developer A, who owns object 1, which uses object 2 owned by developer B, may be notified by a change to object 2's internal workings. Most, if not all, ownership information of technical objects can be extracted from a project's source code repository (CVS, Git, SVN, etc.).

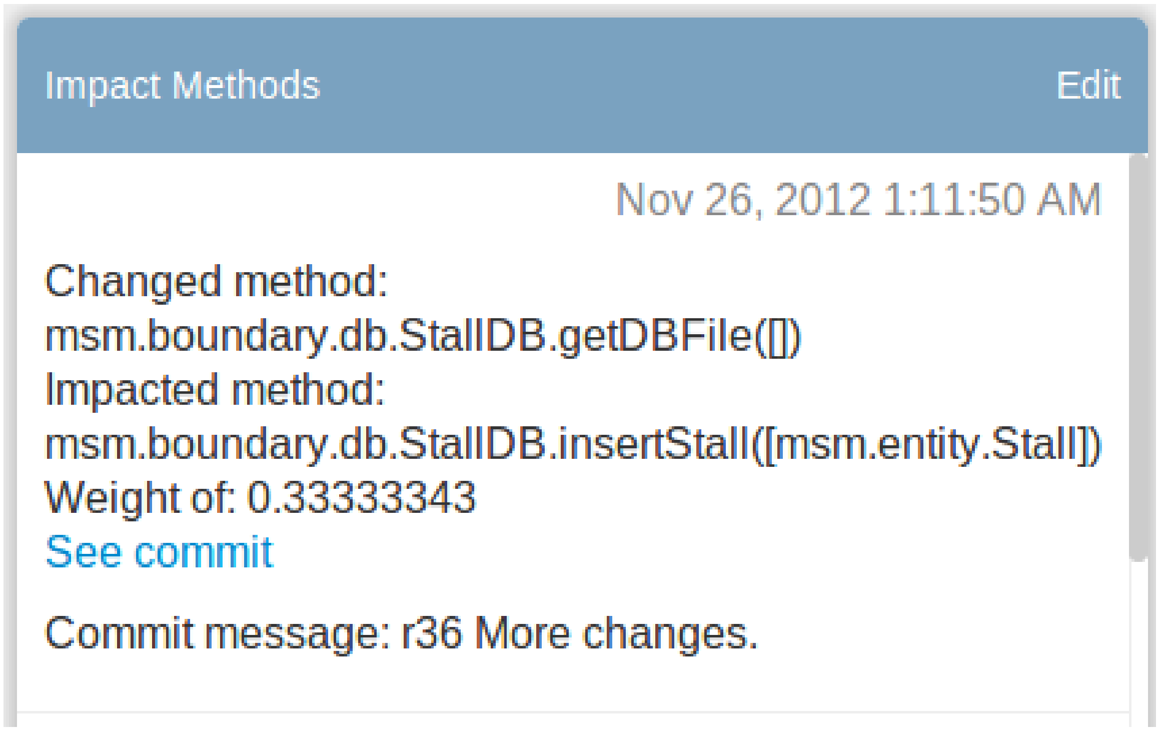
Finally, the indirect conflict tool must be able to detect changes to the technical objects defined above and notify the appropriate owners to the conflict. Two approaches have been proposed for change gathering techniques: real time and commit time [7]. I propose the use of commit time information gathering as it avoids the issue of developers overwriting previous work or deleting modifications which would produce information for changes that no longer exist. However, the trade off is that indirect conflicts must be committed before detected, which results in conflicts being apart of the system before being able to be dealt with as opposed to catching conflicts before they happen. At commit time, the tool must parse changed source code in relation to technical artifacts in the created directed graph detailed above. Where *Impact's* design differs from that of Palantir's is that the object's entire body (method definition and internal body) is parsed, similar to that of CASI [23], at commit time, as opposed to real time, to detect changes anywhere in the technical object. This is a first design step towards avoiding information overload. Once technical objects are found to be changed, appropriate owners of objects which use the changed object should be notified. In Figure 2.2, Carl changes method (technical object) 1, which effects methods 2 and 3 resulting in the alerting of developers Bret, Daniel, and Anne. I have opted to alert the invoking developers rather than the developer making the change to potential solutions as my conflicts are detected at commit time and this supports the idea of a socio-technical congruence [15] from software structure to communication patterns in awareness systems.

With this three step design: (i) creating directed graphs of technical objects, (ii) assigning ownership to those technical objects, and (iii) detecting changes at commit time and the dissemination of conflict information to appropriate owners, I believe a wide variety of indirect conflict awareness tools can be created or extended.

Implementation

For *Impact's* implementation, I decided to focus on methods as my selected technical objects to infer a directed graph from. The "uses" relationship described above for methods is method invocation. Thus, in my constructed dependency graph, methods represent nodes and method invocations represent the directed edges. In order to construct this directed graph, abstract syntax trees (ASTs) are constructed from source files in the project.

Once the directed graph is constructed, I must now assign ownership to the technical objects (methods) as per the design. To do this, I simply query the source code repository.



Nov 26, 2012 1:11:50 AM

Changed method:
msm.boundary.db.StallDB.getDBFile([])
Impacted method:
msm.boundary.db.StallDB.insertStall([msm.entity.Stall])
Weight of: 0.33333343
See commit

Commit message: r36 More changes.

Figure 2.3: *Impact's* RSS type information feed.

In this case I used Git as the source code repository, so the command *git blame* is used for querying ownership information. (Most source code repositories have similar commands and functionality.) This command returns the source code authors per line which can be used to assign ownership to methods.

To detect changes to technical objects (methods), I simply use a commit's *diff* which is a representation of all changes made inside a commit. I can use the lines changed in the *diff* to find methods that have been changed. This gives cause of potential indirect conflicts. I now find all methods in the directed graphs which invoke these changed methods. These are the final indirect conflicts.

Once the indirect conflicts have been found, I use the ownership information of technical objects to send notifications to those developers involved in the indirect conflict. All owners of methods which invoke those that have been changed are alerted to the newly changed method. Impact can been seen in Figure 2.3, the user interface of *Impact*. Here, in an RSS type feed, the changing developer, time of change, changed method, invoking methods, and commit message are all displayed. The weight provided is the percent changed of changed method multiplied by ownership of the invoking method. This allows developers to filter through high and low changes affecting their own source code.

2.2.2 Evaluation

To fully evaluate both the generic design of detecting and resolving indirect conflicts as well as *Impact*, extensive testing and evaluation must be performed. However, I felt that a simple evaluation is first needed to assess the foundation of *Impact*'s design and claims about indirect conflicts at the method level.

I performed a user case study where I gave *Impact* to two small development teams composed of three developers. Each team was free to use *Impact* at their leisure during their development process, after which interviews were conducted with lead developers from each development team. The interviews were conducted after each team had used *Impact* for three weeks.

I asked lead developers to address two main concerns: do indirect conflicts pose a threat at the method level (e.g. method 1 has a bug because it invokes method 2 which has had its implementation changed recently), and did *Impact* help raise awareness and promote quicker conflict resolution for indirect conflicts. The two interviews largely supported the expectation of indirect conflicts posing a serious threat to developers, especially in medium to large teams or projects as opposed to the small teams which they were apart of. It was also pointed out that method use can be a particularly large area for indirect conflicts to arise. However, it was noted that any technical object which is used as an interface to some data construct or methodology, database access for instance, can be a large potential issue for indirect conflicts. Interview responses to *Impact* were optimistically positive, as interviewees stated that *Impact* had potential to raise awareness among their teams with what other developers are doing as well as the influence it has on their own work. However, *Impact* was shown to have a major problem of information overload. It was suggested that while all method changes were being detected, not all are notification worthy. One developer suggested to only notify developers to indirect conflicts if the internal structure of a method changes due to modification to input parameters or output parameters. In other words, the boundaries of the technical objects (changing how a parameter is used inside the method, modifying the return result inside the method) seem to be more of interest than other internal workings. More complex inner workings of methods were also noted to be of interest to developers such as cyclomatic complexity, or time and space requirements.

These two studies have shown that my design and approach to detecting and alerting developers to indirect conflicts appear to be on the correct path. However, *Impact* has clearly shown the achilles heel of indirect conflict tools, which is information overload because of an inability to detect "notification worthy" changes.

2.2.3 Conclusions of Study

In this study, I have proposed a generic design for the development of awareness tools in regards to handling indirect conflicts. I have presented a prototype awareness tool, *Impact*, which was designed around the generic technical object approach. However, *Impact* suffered from information overload, in that it had too many notifications sent to developers.

A potential solution to information overload comes from the ideas of Meyer [17] on "design by contract". In this methodology, changes to method preconditions and postconditions are considered to be the most harmful. This includes adding conditions that must be met by both input and output parameters such as limitations to input and expected output. To achieve this level of source code analysis, the ideas of Fluri et al. [8] can be used on the previously generated ASTs for high granularity source code change extraction when determining if preconditions or postconditions have changed.

Aside from better static analysis tools for examining source code changes, the results of this study potentially imply a lack of understanding into the root causes of indirect conflicts. A theme of information overload to developers continues to crop up in indirect conflicts, of which the root cause should be examined in future studies.

2.3 Problem Description

As Software Configuration Management (SCM) has grown over the years, the maturity and norm of parallel development has become the standard development process instead of the exception. With this parallel development comes the need for larger awareness among developers to have "an understanding of the activities of others which provides a context for one's own activities" [6]. This added awareness mitigates some downsides of parallel development which include the cost of conflict prevention and resolution. However, empirical evidence shows that these mitigated losses continue to appear quite frequently and can prove to be a significant and time-consuming chore for developers [18].

Through the two previous studies, I have shown that developers linked indirectly in source code changes can statistically related to software failures. In the attempts of mitigating these loses through added awareness, I implemented an indirect conflict tool called *Impact*. However, *Impact* ultimately suffered from information overload as seen in its evaluation which was caused by false positives and scalability of the tool.

While other indirect conflict tools have shown potential from developer studies, some of the same problems continue to arise throughout most, if not all tools. The most prevalent issue is that of information overload and false positives. Through case studies, developers have noted that current indirect conflict tools provide too many false positive results leading to information overload and the tool eventually being ignored [20, 23]. A second primary issue is that of dependency identification and tracking. Many different dependencies have been proposed and used in indirect conflict tools such as method invocation [25], and class signatures [20] with varying success, but the identification of failure inducing changes, other than those which are already identifiable by other means such as compilers, and unit tests, to these dependencies still remains an issue. Dependency tracking issues are also compounded by the scale of many software development projects leading to further information overload.

Social factors such as Cataldo et al. [4] notion of socio-technical congruence, have also been leveraged in indirect conflict tools [**?**, 1, 15]. However, issues again of information overload, false positives, dependencies (in developer organizational structure), and scalability come up.

While these issues of information overload, false positives, dependencies, and scalability continue to come up in most indirect conflict tools, only a handful of attempts have been made at rectifying these issues or finding the root causes [10, 13]. In order to fully understand the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts, I will proceed by taking a step back and determine what events occur to cause indirect conflicts, when they occur, and if conditions exist to provoke more of these events. By determining the root causes of source code changes in indirect conflicts, we may be able to create indirect conflict tools which have filtered monitoring in order to only detect those changes with a high likelihood of causing indirect conflicts. I then set out to understand what mitigation strategies developers currently use as opposed to those created by academia. Since developers have identified indirect conflicts as a major concern for themselves, but at the same time are not using the tools put forth by academia, I wish to find what they use to mitigate indirect conflicts. Through these findings, we can create tools which are similar to those already in use by developers in the hopes of a higher adoption rate of tools produced by academia. Finally, I look to find what can be accomplished moving forward with indirect conflicts in both research and industry.

I restate the research goals of this thesis for ease of the reader:

RQ1 *What events or conditions lead to indirect conflicts?*

RQ2 *What mitigation techniques are used by developers in regards to indirect conflicts?*

RQ3 *What do developers want from future indirect conflict tools?*

Chapter 3

The New Approach and Solution

3.1 Some LaTeXExamples

Chapter 4

Experiments

Chapter 5

Evaluation, Analysis and Comparisons

Chapter 6

Conclusions

Appendix A

Additional Information

Bibliography

[1] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[2] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[3] Eric Bodden. A high-level view of java applications. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 384–385, New York, NY, USA, 2003. ACM.

[4] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[5] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 81 –90, aug. 2007.

[6] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[7] Geraldine Fitzpatrick, Simon Kaplan, Tim Mansfield, Arnold David, and Bill Segall. Supporting public availability and accessibility with elvin: Experiences and reflections. *Comput. Supported Coop. Work*, 11(3):447–474, November 2002.

[8] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[9] Christine A. Halverson, Jason B. Ellis, Catalina Danis, and Wendy A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 39–48, New York, NY, USA, 2006. ACM.

[10] Reid Holmes and Robert J. Walker. Customized awareness: recommending relevant external change events. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM.

[11] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[12] Himanshu Khurana, Jim Basney, Mehedi Bakht, Mike Freemon, Von Welch, and Randy Butler. Palantir: a framework for collaborative incident response and investigation. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, IDtrust '09, pages 38–51, New York, NY, USA, 2009. ACM.

[13] Miryung Kim. An exploratory study of awareness interests about software modifications. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 80–83, New York, NY, USA, 2011. ACM.

[14] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Irwin Kwan and Daniela Damian. Extending socio-technical congruence with awareness relationships. In *Proceedings of the 4th international workshop on Social software engineering*, SSE '11, pages 23–30, New York, NY, USA, 2011. ACM.

[16] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 273–284, New York, NY, USA, 1993. ACM.

[17] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[18] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[19] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[20] Anita Sarma, Gerald Bortis, and Andre van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 94–103, New York, NY, USA, 2007. ACM.

[21] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 23–33, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Adrian Schröter, Jorge Aranda, Daniela Damian, and Irwin Kwan. To talk or not to talk: factors that influence communication around changesets. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1317–1326, New York, NY, USA, 2012. ACM.

[23] Francisco Servant, James A. Jones, and André van der Hoek. Casi: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 39–46, New York, NY, USA, 2010. ACM.

[24] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[25] Erik Trainer, Stephen Quirk, Cleidson de Souza, and David Redmiles. Bridging the gap between technical and social dependencies with ariadne. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 26–30, New York, NY, USA, 2005. ACM.

[26] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 2:1–2:1, New York, NY, USA, 2008. ACM.

[27] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.