A Hybrid Software Change Impact Analysis for Large-scale Enterprise Systems

A HYBRID SOFTWARE CHANGE IMPACT ANALYSIS FOR LARGE-SCALE ENTERPRISE SYSTEMS

BY

WEN CHEN, B.Sc, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY OF COMPUTER SCIENCE

© Copyright by Wen Chen, September 2015

All Rights Reserved

Doctor of Philosophy of Computer Science (2015) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	A Hybrid Software Change Impact Analysis for Large-scale Enterprise Systems
AUTHOR:	Wen Chen M.Sc., (Computer Science) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Alan Wassyng and Dr. Tom Maibaum
NUMBER OF PAGES:	xiv, 200

To my daughter Weining Chen

Abstract

This work is concerned with analysing the potential impact of direct changes to large-scale enterprise systems, and, in particular, how to minimise testing efforts on such changes. A typical enterprise system may consist of hundreds of thousands of classes and millions of methods. Thus, it is extremely costly and difficult to apply conventional testing techniques to such a system. Retesting everything after a change is very expensive, and in practice generally not necessary. Selective testing can be more effective. However, it requires a deep understanding of the target system and a lack of that understanding can lead to insufficient test coverage. *Change Impact Analysis* can be used to estimate the impacts of the changes to be applied, providing developers/testers with confidence in selecting necessary tests and identifying untested entities. Conventional change impact analysis approaches include static analysis, dynamic analysis or a hybrid of the two analyses. They have proved to be useful on small or medium size programs, providing users an inside view of the system within an acceptable running time. However, when it comes to large-scale enterprise systems, the sizes of the programs are orders of magnitude larger. Conventional approaches often run into resource problems such as insufficient memory and/or unacceptable running time (up to weeks). More critically, a large number of false-negatives and false-positives can be generated from those approaches.

In this work, a conservative static analysis with the capability of dealing with inheritance was conducted on an enterprise system and associated changes to obtain all the potential impacts. Later an aspect-based dynamic analysis was used to instrument the system and collect a set of dynamic impacts at run-time. We are careful not to discard impacts unless we can show that they are definitely not impacted by the change. Reachability analysis examines the program to see "Whether a given path in a program representation corresponds to a possible execution path". In other words, we employ reachability analysis to eliminate *infeasible* paths (*i.e.,* miss-matched calls and returns) that are identified in the control-flow of the program. Furthermore, in the phase of alias analysis, we aim at identifying paths that are *feasible* but cannot be affected by the direct changes to the system, by searching a set of possible pairs of accesses that may be aliased at each program point of interest.

Our contributions are, we designed a hybrid approach that combines static analysis and dynamic analysis with reachability analysis and alias/pointer analysis, it can be used to (1) solve the scalability problem on large-scale systems, (2) reduce false-positives and not introduce false-negatives, (3) extract both direct and indirect changes, and (4) identify impacts even before making the changes. Using our approach, organizations can focus on a much smaller, relevant subset of the overall test suite instead of blindly doing their entire suite of tests. Also it enables testers to augment the test suite with tests applying to uncovered impacts. We include an empirical study that illustrates the savings that can be attained.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my co-supervisors Dr. Alan Wassyng and Dr. Tom Maibaum for the continuous support of my Ph.D study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better supervisors for my Ph.D study.

The author is also thankful to Ron Mison, Sundaram Viswanathan, and Vinayak Viswanathan of Legacy Systems International, for introducing the problem, and also for working together to ensure the research produces practical methods and tools.

My sincere thanks also goes to Chris George, Dr. Wolfram Kahl, Dr. Mark Lawford, Dr. Ryszard Janicki, for their technical advice either in the beginning or throughout the study. I also thank my colleagues Asif Iqbal, Akbar Abdrakhmanov, Jay Parlar, for their great support in working together.

Last but not least, I would like to thank my family: my parents, wife and daughter, for supporting me spiritually throughout my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

This thesis explores *change impact analysis* for *large-scale enterprise systems*. **Large-scale enterprise systems** are commercial software packages that provide organizations with the ability to integrate varieties of applications cross-functionally, replace hard-to-maintain interfaces, and eliminate redundant data entries to accommodate business growth. However, the use of enterprise systems may lead to high costs of software maintenance and testing since changes are made often to the software. One type of software changes, *patches*, have to be applied often as they are required to upgrade the system, fix defects, and introduce new features. Hence, impact analysis is needed to comprehend the changes, and identify potential impacts so that testers can conduct target testing with appropriate test coverage.

1.1.1 Enterprise Systems

Enterprise systems are commercial software packages that enable the integration of transaction-oriented data and business processes throughout an organization (and perhaps eventually throughout the entire interorganizational supply chain)[MT00]. Enterprise systems include Enterprise Resource Planning (ERP) software and such related packages as advanced planning and scheduling, sales force automation, customer relationship management, financial planning and reporting, and product configuration. Organizations that adopt enterprise systems have a wide range of options for implementation and ongoing operations, from do it yourself, through selective external assistance, to total outsourcing. Enterprise systems are gaining popularity in organizations all over the world. By 1998 approximately 40% of companies with annual revenues of more than $1 billion had implemented an ERP system [CSB98]. One of the largest enterprise vendors SAP, had 2012 revenue of 16.22 billion Euros [AG12]. Among SAP product lines, SAP Business One operation, financials and human resources has over 40,000 customers. Enterprise systems are clearly a staple of the modern IT marketplace. Given the richness of enterprise systems in terms of functionality and potential benefits, it should not be surprising that companies are adopting these systems for many different reasons. The reasons can be technical and/or business related as shown in Table 1.1 [MT00]:

Technical Reasons	• Integrate applications cross-functionally • Replace hard-to-maintain interfaces • Reduce software maintenance burden through outsourcing • Eliminate redundant data entry, concomitant, errors and difficulty analyzing data • Consolidate multiple different systems of the same type
Business Reasons	• Accommodate business growth • Acquire multilanguage and multicurrency IT support • Improve informal and/or inefficient business process • Provide integrated IT support • Standardize procedures across different locations • Present a single face to the customer

Table 1.1: Reasons for Adopting Enterprise Systems

Enterprise systems represent a nearly complete re-architecting of an organization's portfolio of transactions-processing application systems to achieve integration of business processes, systems, and information – along with corresponding changes in the supporting computing platform (hardware, software, databases, telecommunications). They are typically large, complicated, and may also be inadequately documented and date back a number of decades.

Some crucial characteristics of enterprise systems are:

1. Scale.

   Enterprise systems are large. For instance, Oracle Corporation's *E-Business Suite* consists of a collection of ERP, CRM, SCM computer applications either developed or acquired by Oracle. Significant technologies were incorporated into the application including the Oracle database technologies, (engines for RDBMS, PL/SQL, Java, .Net, HTML and XML), and the "technology stack"

(Oracle Forms Server, Oracle Reports Server, Apache Web Server, Oracle Discover, Jinitiator and Sun's Java)[Ora10]. The total number of classes in release 11.5 is over 230 thousand, and the total number of methods is over 4.6 million. This does not include a users' customized code that is built on top of E-Business Suite.

2. Complex.

   Enterprise systems embody an incredibly rich functionality and so it is not trivial to fully understand how the components within the system communicate. We again take Oracle E-Business Suite version 11.5 as an example, and there are over 18 million dependencies between entities in this suite. While in the SAP ERP, there are over 240 individual modules [SAP14].

3. Critical.

   Enterprise systems play a critical role in organizations. They reflect the actual business processes, information flows, reporting, data analytics etc., in an organization, and it is thus critical that the functionality is implemented correctly, and maintained in a safe and efficient way.

4. Costly.

   It is estimated that "For a Fortune 500 company, software, hardware, and consulting costs can easily exceed $100 million (around $50 million to $500 million) yearly. Large companies can also spend $50 million to $100 million on upgrades yearly. Full implementation of all modules can take years." which also adds to the end price. Mid-sized companies (fewer than 1,000 employees) are more likely to spend around $10 million to $20 million at most, and small companies are not likely to have the need for a fully integrated SAP ERP system unless they have the likelihood of becoming mid-sized and then the same data applies as would a mid-sized company. [MW08]

As a consequence of these characteristics, these systems can also often be classified as *legacy systems* that are poorly understood and difficult to maintain. To make matters worse, they are often mission critical, being found in critical roles as strategic systems in large companies. So they are typically seen as both critical and fragile. The Free On-Line Dictionary of Computing (FOLDOC) defines a legacy system [How98] as, "A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and often despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify."

1.1.2 Software Changes

Software change is a fundamental ingredient of software maintenance, and most of the time is *inevitable.* A software change can be produced by any operation (add, modify, delete etc.,) on a set of software entities (functions, fields, database objects, modules etc.,) in a program.

Typical required changes are:

- *Hardware/software upgrades.* In response to increasing system performance requirements, organizations need to upgrade the hardware and/or software environment. Typically a hardware change does not change software functional behaviours, however a minor change in the software environment can lead to unintended and unwanted behaviour.

- *User requirement changes.* It is widely known that user requirements change often during the software development life cycle. This is especially true during the maintenance phase. Each time when a change is requested and then made at the users' end, corresponding code change(s) have to be made in the software to reflect the changed requirement.
- *System upgrading.* Patches are supplied by enterprise system vendors to underlying middleware, and for a number of reasons such as bug fixing, and error correction they may need to be applied. For a Fortune 500 company, the upgrading cost can easily exceed 50 million US dollars [MW08].
- *Customization.* Organizations that implement and benefit from enterprise systems may need to adjust how they use the system. For instance, an upgrading of the system may cause old APIs to be unavailable to customized code, hence changes need also be made to the customized code.

The 2011 IT Key Metrics Data from Gartner [gar11] report that some 16% of application support activity was devoted to technical upgrades, rising to 24% in the banking and financial services sector. A perpetual problem for the organization is how to manage such changes with minimum risk and cost.

However, changes can be *unintended*, which may lead to a decrease in software reliability, and may even cause software defects and failures. Risk of unintended change is typically addressed by regression testing. As already noted, one problem is that regression testing is expensive and time-consuming for large systems with interactive interfaces. Organizations can spend millions of dollars per annum on it. The actual effect of a middleware patch or an application software change may in fact be minimal, so a small fraction of the regression tests may be sufficient; but, with an enterprise system, it is very risky to make a judgement about what should be tested and what can be assumed to be unaffected by the change.

Moreover, there is some chance the changes are not fully covered by the test suite. So, the addition, modification, and deletion of code entities in the program may impose new application logic that needs to be covered by testing. However, before applying the actual change, it is very risky to pick up entities to be tested by one's domain knowledge only. For the sake of safety and effectiveness, we need a way to identify all the impacts after or even before making a change. What business rule might be affected by a patch to the enterprise system, or by a planned change to the customization code, or what data is stored in the database? If organizations know the possible impact of a change they can select only the relevant tests, confident that the others do not need to be run in the old test suite; in addition to that, knowing what was not but should be covered can be used to augment the original test suite.

1.1.3 Impact Analysis

Change impact analysis is the key in analyzing software changes or potential changes and in identifying the software objects the changes might affect [Boh96]. Organizations need a change impact analysis tool to identify the impact of a change after or even before making a change. If such impacts can be obtained even before applying the changes, it enables the organization to make test plans or run tests in advance, saving the lag between system deployment and release.

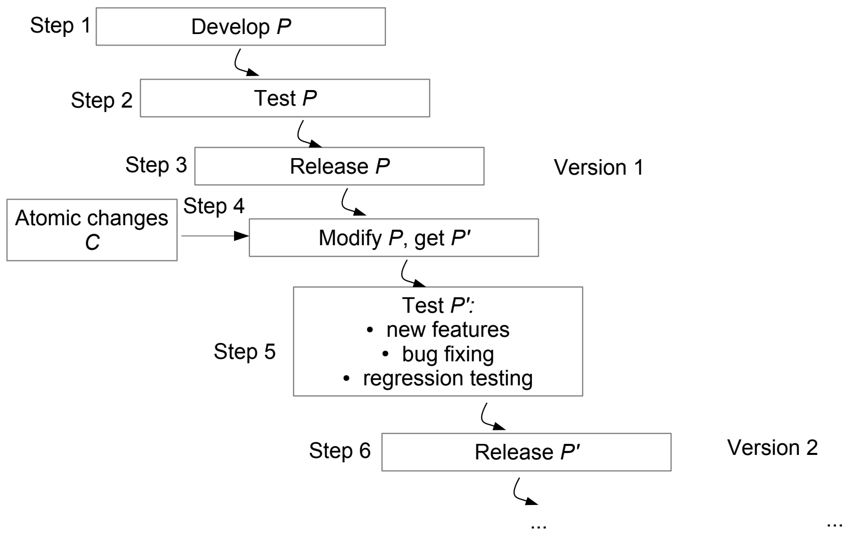


Figure 1.1: Program Release Process

By using the identified impacts, organizations can

1. know what to test instead of testing blindly.

   Suppose we have a program P to be tested, and the process for developing, testing and releasing the program is depicted in Figure 1.1. We note that if we know the changes we want to make to P to create a new version, P', an impact analysis could be conducted between Step 3 and Step 4, so that the effects of these changes may be understood after the release of the old version and before making any modifications to the actual code. This is an important component of risk management, since small changes in C may have subtle undesired effects in other seemingly unrelated parts of the program.

   Through the use of impact analysis we could obtain a set of impacts I that depicts what other parts of the program can be affected, and verify whether the changes introduce new bugs. This enables organizations to revalidate the functionalities inherited from the old release. To do this, organizations select a subset T_{sub} (relevant tests) of the original test suite T such that successful execution of the modified code P' against T_{sub} implies that all the functionality carried over from the original code to P' is still intact by comparing the test results to the previously recorded baseline of testing P against T_{sub}. Without effective impact analysis, regression testing can be very dangerous and risky.

2. augment the test suite to cover software entities that are affected but not covered in the original test suite.

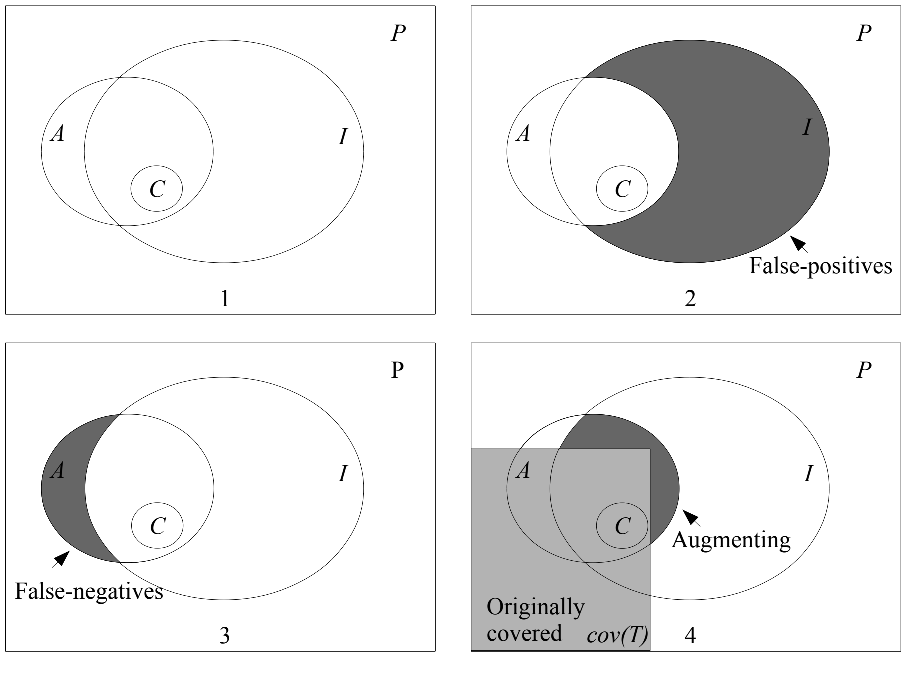


Figure 1.2: False-positives And False-negatives

The impact set I we can develop should (but may not) contain all affected entities, not only the ones covered by the original test suite T, but also entities that may not have been covered by T. To better illustrate how impact analysis can help with completing the test coverage, we introduce a metric for evaluating impact analysis tools. In Figure 1.2(1), a program P is under analysis with atomic changes C, estimated set of impacts I and real set of impacts A. We can evaluate the analysis by a pair of trade-offs: *False-positive* and *False-negative* [MBdFG10]. If the analysis results contain a specific entity that was not truly affected, this error in the analysis is called a *false-positive.* Conversely, when the analysis results do not contain a truly affected entity, an error occurs that is called a *false-negative.* From the same source we also restate two metrics from the information retrieval domain: *Precision* and *Recall*, which are respectively associated with false-positives and false-negatives. Precision is the ratio between correctly estimated entities and the total estimated entities:

$$Precision = \frac{|A \cap I|}{|I|} \tag{1.1}$$

On the other hand, recall is the ratio between correctly estimated entities and the total number of truly affected entities:

$$Recall = \frac{|A \cap I|}{|A|} \tag{1.2}$$

In Figure 1.2(2) and Figure 1.2(3), the areas in shadow are false-positives and false-negatives, respectively. And in Figure 1.2(4), the area in light shadow is part of the program that has test coverage in the original test suite T. Note that one may not have made tests that fully cover all the actual impacts A, since conventional regression testing has a high risk of missing entities that need to be tested. Hence we can employ impact analysis to augment T to $T^{'}$ to cover the heavily shadowed area. Assuming $cov(T)$ is the set of entities covered by T, we can express this augmenting area A' and uncovered impacts $uncov(T)$ after the analysis by:

$$A' = A - (A - cov(T) - I) - (A \cap cov(T)) \tag{1.3}$$

and

$$uncov(T) = A - cov(T) - I^1 \tag{1.4}$$

Then $cov(T)$ could be augmented to $cov(T)'$ as

$$cov(T)' = cov(T) \cup A' \tag{1.5}$$

However, organizations tend to conduct full regression testing or select tests in a conservative way, using testers' domain knowledge, rather than utilizing any impact analysis tool. The reasons may vary, but major ones are likely to be:

- current impact analysis techniques are not reliable, in terms of completeness and precision.

  It does not appear to the author that there exists any impact analysis that is

[1]Depending on how good the *recall* is, after analysis there still might be a portion of the actual impacts not covered. In Figure 1.2(4), this corresponds to the little white triangle area on the top-left.

precise and safe (details will be discussed in Chapter 2), i.e., with few false-positives but having all the potential impacts taken into account. For enterprise systems, organizations consider false-negatives way more risky than false-positives, while most of the current tools seek a balance between false-positives and false-negatives with no priorities.

- the size of the system puts it beyond any existing tool's ability to adequately comprehend dependencies between entities in the system.

  Enterprise systems are orders of magnitude larger than the size of programs other existing tools that the author is aware of can deal with. Hence most of the existing tools typically run out of memory and/or the execution time is too long to be useful.

1.2 Research Motivation

The characteristics of complexity and scalability of large-scale enterprise systems, put it beyond a tester's ability to adequately comprehend the impact of changes that are made to the system, and this often results in high costs related to testing, as well as the risk of missing dependencies that later result in software defects in the system. The majority of testing performed on these systems is dependent on testers' domain knowledge, which makes the testing results very risky to rely on as such knowledge has its limits, especially in the context of complexity. The author's research motivation and objectives are:

1. Deal with the scale. Scalability remains one of the biggest issues in enterprise systems. Existing impact analysis tools can work perfectly for small programs

but fail on large ones, since researchers do not pay enough attention to the challenges imposed by the size of enterprise systems. This is also one of the reasons that organizations prefer to rerun all the tests to validate the system after a change since existing tools simply do not work on these very large systems.

2. Determine both direct changes and indirect changes. Most of the time, it is not straightforward to determine precisely what entities in the system are to be changed directly. For example, if the system is implemented in Java, the changed entities can be compiled into bytecode that then needs to be decompiled in order to determine exactly which entities have been changed. In the case of making changes to the database, changes may be encapsulated in database scripts. This is important since not only code changes can impact the system, but data changes also may affect system behaviours. We want to identify both direct changes and indirect changes, and indirect changes are not straightforward either. For instance, an efficient *string analysis* had to be used to collect indirect changes resulting from a direct change made to a database table in a program written in Java.

3. Preserve both safety and precision[2]. Since enterprise systems play a critical role in organizations, any mis-identified impacts may cause financial losses, and so we want our approach to impact analysis to be both safe and precise. By safe we mean that the approach is conservative and will find all the actual impacts. This translates into not allowing any false-negatives! In terms of false-negatives and corresponding recall[3], the recall must be equal to 1. In

[2]With regard to the definition of *precision* in equation 1.1.
[3]With regard to the definition of *recall* in equation 1.2.

the meantime, the approach should be precise by removing false-positives, such that the impact set more accurately reflects the actual impacts that need to be tested. In other words, we want to eliminate over-estimation as much as we can. Ideally, precision should also be close to 1, but precision has much lower priority than recall, and one does not have unlimited time to instrument the program but is only able to instrument a subset of the entire program. To achieve this, either static or dynamic analysis by itself may not be sufficient. We need a refined approach that takes advantages of conventional approaches but also has the ability to eliminate disadvantages. This may require combining static analysis (which can be made safe but with poor precision) and dynamic analysis to remove false-positives, thereby introducing a new technique that can subtract over-estimated entities as much as possible in a safe way.

4. Identify impacts even before making a change. Applying a single change/patch takes time, since one has to install also the preliminaries of the change. More importantly, without a full understanding of the potential impacts, it is very risky to apply the changes. Restoring an entire enterprise system to a previous working state is extremely costly and sometimes even impossible. Any faults, failures or bugs have to be fixed after the modification. Therefore, we want an impact analysis that can provide impacts before making any change. Even though one may still need to conduct testing after the deployment, this ability could largely reduce the risks of unintended changes. To achieve this, the analysis may need to establish the linkage between the changes and code entities in the old version.

1.3 Structure of the Thesis

Chapter 1 described the problem domain and motivation for the work. Chapter 2 discusses related work from different aspects: general impact analysis, static/dynamic approaches, instrumentation etc. Static and dynamic impact analysis are discussed in Chapter 3: definitions, pros and cons, two example approaches, and a combination of static and dynamic analysis is introduced in the last section. Chapter 4 discusses how to extract both direct changes and indirect changes from the set of atomic changes, with respect to both the application library and the database. To further elaborate the approach, reachability analysis and alias analysis are introduced in Chapter 5 and Chapter 6, respectively. Then a complete approach is proposed in Chapter 7, in which varieties of analysis that constitute the approach are combined into the hybrid approach. Chapter 8 is the empirical study of the hybrid approach, in which evaluation criteria, implementation details, experiment design, empirical results etc. are presented. Chapter 9 sums up the thesis and discusses future work.

Chapter 2

Related Work

Research on software change impact analysis can be traced back to the 1970s. Reasons for doing change impact analysis are well known and understood: “As software components and middleware occupy more and more of the software engineering landscape, interoperability relationships point to increasingly relevant software change impacts [Boh96]”. Moreover, due to the increasing use of techniques such as inheritance and dynamic dispatching/binding, which come from widely used object-oriented languages, small changes can have major and non-local effects [RT01]. To make matters worse, those major and non-local effects might not be easily identified, especially when the size of the software puts it beyond any maintainer’s ability to adequately comprehend.

There is considerable research related to this field, but it seems that there are limited known ways of performing change impact analysis. Bohner and Arnold [Boh96] identify three types of impact analysis (IA): *traceability*, *dependency* and *experimental*. In traceability IA, links between requirements, specifications, design elements, and tests are captured, and these relationships can be analyzed to determine the scope of an initiating change. In dependency IA, linkages between parts, variables, logic, modules etc., are assessed to determine the consequences of an initiating change. Dependency IA occurs at a more detailed level than traceability IA. Within software design, static and dynamic algorithms can be run on code to perform dependency IA. Static methods focus on the program structure, while dynamic algorithms gather information about program behaviour at run-time. Experiential IA, is when the impact of changes is determined using expert design knowledge. Review meeting protocols, informal team discussions, and individual engineering judgement can all be used to determine the consequences of a modification.

This thesis falls into the second category: dependency IA. Our target system, the enterprise systems, often date back many years, hence requirements and other specifications are either incomplete or even incorrect. Traceability IA would not be a good option for us. Also, since we are aiming at saving human effort and financial costs via testing guided by IA, experimental IA is not of interest to us at this time as it requires a tremendous amount of activities such as team discussions.

Dependency impact analysis can be either static, or dynamic, or a hybrid of the two. We discuss some of the work from those techniques in the discussion that follows. *Static impact analysis* [Boh96, LMS97, PA06, TM94, RST$^+$04] identifies the impact set — the subset of elements in the program that may be affected by the changes made to the system, by analyzing relevant source code. For instance, Chianti [RST$^+$04] is a static change impact analysis tool for Java that is implemented in the context of the `Eclipse` environment, and it (Chianti) analyzes two versions of an application and decomposes their differences into a set of atomic changes, i.e., it tells which tests in the test suite can be affected. The change impact is then reported in terms of affected tests. It consists of four steps as follows:

- Step 1. A source code edit is analyzed to obtain a set of interdependent atomic changes A, whose granularity is (roughly) at the method level. These atomic changes include all possible effects of the edit on dynamic dispatch.
- Step 2. A call graph is constructed for each test in the original test suite T.
- Step 3. For a given set T of unit or regression tests, the analysis determines a subset T_{sub} of T that is potentially affected by the changes in A, by correlating the changes in A against the call graphs for the tests in T in the original version of the program.
- Step 4. Finally, for a given test t_i in T, the analysis can determine a subset A' of A that contains all the changes that may have affected the behaviour of t_i. This is accomplished by constructing a call graph for t_i in the edited version of the program, and correlating that call graph with the changes in A.

Note that this tool requires source code of the program being analyzed while many organizations only have a running (compiled) version of the program. Also it relies on the user's test cases, which means that, if the test suite created by the user is not well defined (bad test coverage), impacts of the change might not be fully identified. It constructs call graphs for each test in the user's regression test suite, and then determines a subset of that suite.

Apiwattanapong *et al.* [Api05] pointed out that static impact analysis algorithms often come up with too large impact sets due to their over conservative assumption and might turn out to be effectively useless. For example, regression testing techniques that use impact analysis to identify which parts of the program to retest after a change may have to retest most of the program. They also point out a two-fold problem with sound static impact analysis. First, it considers all possible behaviours of the software, whereas, in practice, only a subset of such behaviours may be exercised by the users. Second, and more importantly, it also considers some impossible behaviours, due to the imprecision of the analysis. Therefore, recently, researchers have investigated and defined impact analysis techniques that rely on *dynamic*, rather than static, information about program behaviour [LR03a, LR03b, OAH03, BDSP04].

The dynamic information consists of execution data for a specific set of program executions, such as executions in the field, executions based on an operational profile, or executions of test suites. Apiwattanapong *et al.* [Api05] defines the dynamic impact set to be the subset of program entities that are affected by the changes during at least one of the considered program executions. `CoverageImpact` [OAH03] and `PathImpact` [LR03a, LR03b] are two well known dynamic impact analysis techniques that use dynamic impact sets. `PathImpact` works at the method level and uses compressed execution traces to compute impact sets. `CoverageImpact` also works at the method level but it uses coverage, rather than trace information to compute impact sets. Though the dynamic approach can make the analysis more efficient, it does not guarantee that all actual system behaviours can be captured. Thus it might result in a good number of *false-negatives*, and this will usually cause bugs that then need to be fixed [MBdFG10]. Consequently, false-negatives may cause tremendous financial losses for the organization.

A crucial task in dynamic approaches is to instrument the program such that dynamic information can be collected to compute the impacts. The instrumentation of applications to generate run-time information and statistics is an important enabling technology for the development of tools that support the fast and accurate simulation of computer architectures [PSM95]. There are two types of collection schemes: *hardware-assisted* and *software-only.* Most of the existing research focuses on software-only instrumentation and collection, since it is more portable and relatively inexpensive. We can divide software-only instrumentation into two approaches: (1) those that simulate, emulate, or translate the application code; and (2) those that instrument the application code. `SPIM` [PH08], `Shade` [CK95] and more recently `Pin` [LCM$^+$05], `Soot` [LBL$^+$10] fall into the first type in which the original code was simulated or transformed to some **intermediate representation** to be processed. On the other hand, tools like `ATUM` [ASH86] and more recently `BCEL` [Sos04], `AspectJ` [Asp14] and `InsECTJ` [SO05] fall into the other type, where the application code is executed and runtime information is collected.

Instrumentation tools that simulate, emulate or translate the application code require extra computing time to accomplish the simulation, emulation or translation. For instance Soot provides four intermediate representations for code: Baf, Jimple, Shimple and Grimp [EN08]. The representations provide different levels of abstraction of the represented code and are targeted at different uses e.g., Baf is a bytecode representation resembling the Java bytecode and Jimple is a stackless, typed 3-address code suitable for most analyses. Jimple representations can be created directly in Soot or based on Java source code (up to and including Java 1.4) and Java bytecode/Java class files (up to and including Java 5). The translation from bytecode to Jimple is performed using a naive translation from bytecode to untyped Jimple, by introducing new local variables for implicit stack locations and using subroutine elimination to remove `jsr` instructions. Types are inferred for the local variables in the untyped Jimple and added. The Jimple code is cleaned of redundant code like unused variables or assignments. This tool works perfectly for small or medium programs, but when it comes to systems of the size of enterprise system, it runs out of memory since an intermediate representation is required for each class. Instrumentation tools in the first category are not the first choice in solving our problem.

Among the instrumentation tools, aspect-oriented programming and `AspectJ` (one of its implementations to Java) are gaining more popularity. AspectJ [KHH+01] is a simple and practical aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support for the modular implementation of a range of crosscutting concerns. Join points are principled points in the execution of the program; pointcuts are collections of join points; advice is a special method-like construct that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprised of pointcuts, advice, and ordinary Java member declarations. AspectJ code is compiled into standard Java bytecode. Simple extensions to existing Java development environments make it possible to browse the crosscutting structure of aspects in the same kind of way as one browses the inheritance structure of classes. There are many examples [SB06b] [CC07] [BHRG09] showing that AspectJ is powerful in instrumenting the code, and that programs written using it are easy to understand.

Research conducted by Dean and Spencer [JR00] presents a *program understanding* technique that combines static and dynamic analyses to extract components and connectors. The technique was implemented in a tool called *Interaction Scenario*

Visualizer (ISVis). Component identification is supported by traditional static analysis, which is similar to our approach. However, connectors consist of component interaction patterns as recognized from actual execution traces, a form of dynamic analysis. Dynamic analysis of interactions between components might work perfectly in *architectural localization*, but as we have pointed out, such analysis is inherently not complete.

Recently, a hybrid of static and dynamic analysis was investigated. Mirna *et al.* [MBdFG10] proposed a hybrid technique for object-oriented software change impact analysis. The technique consists of three steps: static analysis to identify structural dependencies between code entities, dynamic analysis to identify dependencies based on a succession relation derived from execution traces, and a ranking of results from both analyses that takes into account the relevance of dynamic dependencies. The evaluation of this work showed the hybrid approach produced fewer false-negatives and false positives than a precise and efficient dynamic tool `CollectEA` [Api05]. At the same time, it produced more false positives but fewer false-negatives than a static tool `Impala` [HGF$^+$08].

Tom Reps *et al.* [Rep98] showed how a number of program analysis problems can be solved by transforming them to a *graph-reachability* problem. The purpose of program analysis is to ascertain information about a program without actually running the program. In Reps's work, program-analysis problems can be transformed to *context-free-language reachability problems* ("CFL-reachability problems"). In graph theory, reachability refers to the ability to get from one vertex to another within a graph. We say a vertex s can reach a vertex t (or that t is reachable from s) if there exists a sequence of adjacent vertices (i.e. a path) which starts with s and ends with t. Algorithms for determining reachability fall into two classes: Those that require preprocessing and those that do not [Ger06]:

- If you have only one (or a few) queries to make, it may be more efficient to forgo the use of more complex data structures and compute the reachability of the desired pair directly. This can be accomplished in linear time using algorithms such as *breadth-first search* or iterative deepening *depth-first search*.
- If you will be making many queries, then a more sophisticated method may be used; the exact choice of method depends on the nature of the graph being analyzed. In exchange for preprocessing time and some extra storage space, we can create a data structure which can then answer reachability queries on any pair of vertices in as low as O(1) time. Algorithms such as the *Floyd-Warshall Algorithm*, *Thorup's Algorithm*, *Kameda's Algorithm* are in this category.

A CFL-reachability problem [Rep98] is not an ordinary reachability problem (e.g. transitive closure), but one in which a path is considered to connect two nodes only if the concatenation of the labels on the edges of the path is a word in a particular context-free language: Let L be a context-free language over alphabet S, and let G be a graph whose edges are labeled with members of S. Each path in G defines a word over S, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is a member of L. By using CFL-reachability analysis, it can answer our question "Does a given path in a program representation (e.g. graphs) correspond to a possible execution path?". The answer to this question can help us reduce false-positives from the impact set.

Many compiler analyses and optimizations require information about the behaviour of pointers in order to be effective. A *pointer analysis* is a technique for computing information that attempts to statically determine the possible runtime values of a pointer [Hin01]. Aliasing occurs when two distinct names (data access paths) denote the same run-time location. It is introduced by reference parameters and pointers. This analysis has been studied extensively over the last decade. A *pointer alias analysis* attempts to determine when two pointer expressions refer to the same storage location. A *points-to analysis* [EGH94, And94], or similarly, an analysis based on a "compact representation" [CBC93, BCCH95, HBCC99], attempts to determine what storage locations a pointer can point to. This information can then be used to determine the aliases in the program. Alias information is central to determining what memory locations are modified or referenced.

Lhoták introduced a flexible framework `SPARK` for experimenting with points-to analyses for Java [Lho02]. `SPARK` is intended to be a universal framework within which different points-to analyses can be easily implemented and compared in a common context. And two client analyses that use the points-to information are described in [Lho02], call graph construction and side-effect analysis. Pointer analysis also has a trade-off between the efficiency of the analysis and the precision of the computed solution. Hence, according to where the analysis is used, there are different dimensions that affect the cost/precision trade-offs:

- *Flow-sensitivity.* Flow-sensitive pointer analysis uses the control-flow information of a procedure during the analysis to compute a solution for each program point. Conversely, flow-insensitive analysis computes one solution either for the whole program [And94, Ste96] or for each method [HBCC99, LH99].

- *Context-sensitivity.* Is calling context considered when analyzing a function or can values flow from one call through the function and return to another caller?

- *Aggregate modelling.* Are elements of aggregates distinguished or collapsed into one object?
- *Whole program.* Does an analysis require the whole program or can a sound solution be obtained by analyzing only components of a program?
- *Alias representation.* Is an explicit alias representation [LR04, O'C01] or a points-to/compact representation used?

The aim of aliasing/pointer analysis is to determine for each program point L an upper approximation of the exact set of possible pairs of accesses that may be aliased when L is reached [Deu94]. It does not appear to the author that there exists any work that uses aliasing analysis to assist impact analysis, but we may utilize it to identify aliased objects to the direct and indirect changes and try to reduce false-positives from the impact set.

To sum up, there is great potential for combining static and dynamic approaches in analyzing change impacts on software programs. Static analysis operates by building a model of the state of the program, then determining how the program reacts to this state [Ern03]. It considers all possible software behaviours which may result in imprecision, but it provides a conservative way to assess the impacts that leads to soundness and safety. Also, as software impact analysis often works for *Regression Testing* and *Test Selection*, it can be used to jointly determine whether changes made on the system have been fully covered by a user's test suite. It is important that no necessary impacts are omitted. Starting with an abstract model of the state of the program (e.g., *Dependency Graph, Control Flow Graph*), dynamic analysis can be used to make the analysis more precise. Dynamic analysis is precise because no approximation or abstraction needs to be done: the analysis can examine the actual run-time behaviour of the program [Ern03], or more exactly, some paths through the program. There is little or no uncertainty in what control paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest. Dynamic analysis can be as fast as program execution. Thus, if static and dynamic analysis can be combined in an effective way, both safety and precision may be accomplished.

Additionally, to achieve better precision, extra analysis can be undertaken in the impact analysis. It appears to the author both reachability analysis and aliasing/ pointer analysis can be employed to help reduce the false-positives. As far as the author can ascertain, there does not exist any other work that has added reachability or aliasing analysis to impact analysis. Reachability analysis searches the graph representation and determines if a path is feasible. Due to the over-estimation in static analysis, there is a chance that infeasible paths can be left in the results. Aliasing analysis has been extensively researched in the last decade; it examines pairs of aliased references or pointers. Hence, using this analysis we may further reduce false-positives by identifying entities aliased to the change.

Chapter 3

Static and Dynamic Analysis[1]

Conventional impact analysis approaches include ones based on static analysis and others based on dynamic analysis. There has been extensive research work on conventional impact analysis approaches, but they do not deal adequately with the scale of enterprise systems or provide a good solution in terms of both safety and precision. In this chapter, we will discuss how exactly static or dynamic analysis is used in IA, what representations are abstracted to support the analyses, how dependency data is built, how to cope with problems caused by object-oriented features, how to instrument the system, what information is to be collected, and what the pros and cons of the various approaches are. Also, we discuss how to combine static and dynamic analysis to achieve a better solution with respect to the target enterprise systems.

[1]This chapter is partially based on previous work by the author and colleagues: Wen Chen, Asif Iqbal, Akbar Abdrakhmanov, Jay Parlar, Chris George, Mark Lawford, Tom Maibaum, and Alan Wassyng. "Large-Scale Enterprise Systems: Changes and Impacts." In *Enterprise Information Systems*, pp. 274-290. Springer Berlin Heidelberg, 2013. CIA$^+$13

3.1 Static Analysis[2]

Static analysis [Ern03] examines program code and reasons over all possible behaviours that might arise at run-time. Typically, static analysis is conservative and sound. Soundness guarantees that analysis results are an accurate description of the program's behaviour, no matter on what inputs or in what environment the program is run. Conservatism means reporting weaker properties than may actually be true; the weak properties are guaranteed to be true, preserving soundness, but may not be strong enough to be useful. For instance, given a function f, the statement "f returns a non-negative value" is weaker (but easier to establish) than the statement "f returns the absolute value of its argument." A conservative analysis might report the former, or the even weaker property that f returns a number. In our case, conservative and sound means that no impacts are missed. However, it also results in impacts being reported that are not actually impacts at all.

Static analysis operates by building a model of the state of the program, then determining how the program reacts to this state. Because there are many possible executions, the analysis must keep track of multiple different possible states. It is usually not reasonable to consider every possible run-time state of the program; for example, there may be arbitrarily many different user inputs or states of the runtime heap. Therefore, static analyses usually use an abstracted model of program state that loses some information, but which is more compact and easier to manipulate than a higher-fidelity model would be. In order to maintain soundness, the analysis must produce a result that would be true no matter the value of the abstracted-away state components. As a result, the analysis output may be less precise (more approximate,

[2]This section is joint work with Asif Iqbal [Iqb11].

more conservative).

Static analysis in IA [Boh96, LMS97, PA06, TM94, RST$^+$04] identifies the impact set - a subset of elements in the program that may be affected by the changes made to the system. To obtain this subset, a representation of the program has to be abstracted. Among the abstract representations, *graphs* have been used extensively to model many problems. We need a graph consisting of nodes and edges that incorporates the dependency relationship between the entities (methods, fields). The graph structure and dependency relation it represents can be used as a knowledge base during the impact analysis.

3.1.1 Building Graphs

Graphs have been used extensively to model many problems that arise in the fields of computer science and software engineering. Especially in software engineering, *call graphs*, *control-flow graphs*, *data-flow graphs*, *component graphs* etc., provide an effective analytical approach to understand and characterize software architecture, static and dynamic structure and meaning of programs. A diagrammatic view (by graphs) of the structure of the code is almost always an excellent way to present issues related to software engineering analysis. That is why graphs are a preferred tool used by software engineers and researchers to understand, re-engineer and analyze codes.

A number of graph analysis techniques are available for software engineering applications. Control-flow analysis, data-flow analysis, call graph analysis, and analysis using component graphs are some of them. In control-flow analysis, a control-flow graph is used to analyze and understand how the control of the program is transferred from one point to another. Similarly data-flow analysis uses data-flow graphs to show and analyze data dependencies among the instructions of the program. Component graphs identify the components of a program, and showing the "uses' relation among those components is very useful in software architecture identification and recovery. Call graph analysis uses call graphs to detect calling dependency relations among entities of the environment. A call graph example and definition is given below:

A call graph is a directed graph $G = (N, E)$ with a set of nodes N and a set of edges $E \subseteq N \times N$. A node $u \in N$ represents a program procedure and an edge $(u, v) \in E$ indicates that procedure u calls procedure v. Consider the call graph in Figure 3.1. It has a set of nodes $N = \{a, b, c, d, e\}$ and a set of edges $\{(a, b), (a, c), (b, d), (c, d), (c, e)\}$.

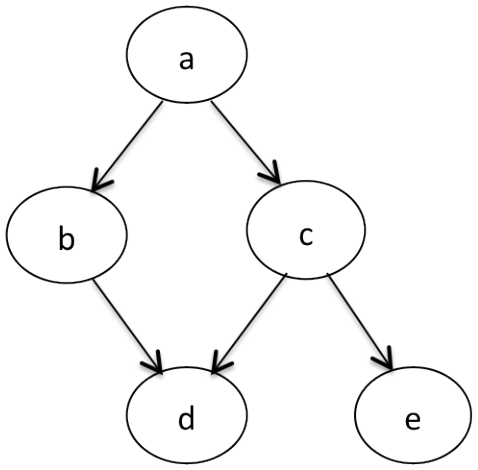


Figure 3.1: Call Graph

There are a few factors that can affect building the call graphs, especially in an *object-oriented language* context like Java. According to the Java Virtual Machine Specification [LY99], a method can be called in 4 ways: `invokeinterface`, `invokespecial`, `invokestatic`, and `invokevirtual`. `invokeinterface` is used to invoke a method declared within a Java interface. `invokespecial` is used in certain special cases to invoke a method. Specifically, it is used to invoke:

- the instance initialization method, $< init >$
- a private method of the calling class itself
- a method in a superclass of the calling class.

`invokestatic` is used to invoke static methods. `invokevirtual` dispatches a Java method. It is used in Java to invoke all methods except interface methods (which use `invokeinterface`), static methods (which use `invokestatic`), and the few special cases handled by `invokespecial`. The actual method that is run depends on the runtime type of the object `invokevirtual` is used with.

To handle these four kinds of method call we have to include in the call graph edges that represent calls to methods, whenever we encounter any of these four kinds of invocations in the bytecode. The simple notion of call graph described works well in traditional non-object oriented languages like C. However, in an object oriented language like Java, where methods (procedures) are encapsulated inside classes and those classes can have fields in addition to methods, the notion of a call graph is far more complicated. Also, features like *inheritance*, *dynamic binding* etc, can introduce implicit dependencies on methods or fields which are not explicitly present in the source code or even in the compiled bytecode. For example, consider class B which extends class A and overrides A's method $m()$. Now there is an explicit call from class C's method $c()$ to class A's method $m()$, and due to dynamic binding this call might actually result in a call to class B's method instead of class A's method $m()$.

Soot[LBL$^+$10] is quite widely used in Java optimization frameworks for optimizing bytecode and carrying out several kinds of control-flow analysis, data-flow analysis, extracting call-graphs, etc. We attempted to generate call graphs with it but experienced a number of performance related issues as described a little later in this section.

Vijay *et. al* discussed 3 kinds of analyses to generate call graphs which have been incorporated in Soot: *Class Hierarchy Analysis (CHA)*, *Rapid Type Analysis (RTA)*, *Variable-type Analysis and Declaration-type Analysis.*

Class hierarchy analysis is a standard method for conservatively estimating the run-time types of receivers[3] [BS96]. Given a receiver o with a declared type d, $hierarchy\_types(d)$ for Java are defined as follows:

- If receiver o has a declared class type C, the possible run-time types of o, $hierarchy\_types(C)$, includes C plus all subclasses of C.
- If receiver o has a declared interface type I, the possible run-time types of o, $hierarchy\_types(I)$, includes: (1) the set of all classes that implement I or implement a subinterface of I, which they call $implements(I)$, plus (2) all subclasses of $implements(I)$.

It is worth noting that this analysis takes into account only the subclasses and subinterfaces, but not the superclasses. This analysis results in the call graph with the maximum number of edges.

Rapid type analysis [SHR+00, BS96] is a very simple way of improving the estimate of the types of receivers. The observation is that a receiver can only have a type of an object that has been instantiated via a *new* operation. Thus, one can collect the set of object types instantiated in the program P, call this $instantiated\_types(P)$. Given a receiver o with declared type C with respect to program P, they use $rapid\_types(C, P) = hierarchy\_types(C) \cap instantiated\_types(P)$ as a better estimate of the runtime types for o. This particular version of rapid type analysis is called pessimistic rapid type analysis [SHR+00] since it starts with the complete conservative call graph built by

[3]A receiver is the object on which the method is invoked.

CHA and looks for all instantiations in methods in that call graph. The original approach suggested by Bacon and Sweeney [BS96] is optimistic rapid type analysis. In the optimistic approach the call graph is iteratively created, and only instantiations in methods already in the call graph are considered as a possible set for computing $instantiated\_types(P)$.

According to [SHR+00], Rapid type analysis can be considered to be a coarse grain mechanism for approximating which types reach a receiver of a method invocation. In effect, rapid type analysis says that a type A reaches a receiver o if there is an instantiation of an object of type A ($i.e.$ an expression $newA()$) anywhere in the program, and A is a plausible type for o using class hierarchy analysis.

For a better approach, they point out that for a type A to reach a receiver o there must be some execution path through the program which starts with a call of a constructor of the form $v = newA()$ followed by some chain of assignments of the form $x_1 = v; x_2 = x_1; \ldots ; xn = x_{n-1}; o = x_n$. The individual assignments may be regular assignment statements, or the implicit assignments performed at method invocations and method returns. They propose two flow-insensitive approximations of this reaching types property. Both analyses proceed by: (1) building a type propagation graph where nodes represent variables, and each edge $a \to b$ represents an assignment of the form $b = a$, (2) initializing reaching type information generated by assignments of the form $b = newA()$ ($i.e.$ the node associated with b is initialized with the type A) and, (3) propagating type information along directed edges corresponding to chains of assignments. These two are Variable-type Analysis and Declaration-type Analysis. The details of these techniques have been discussed in [SHR+00].

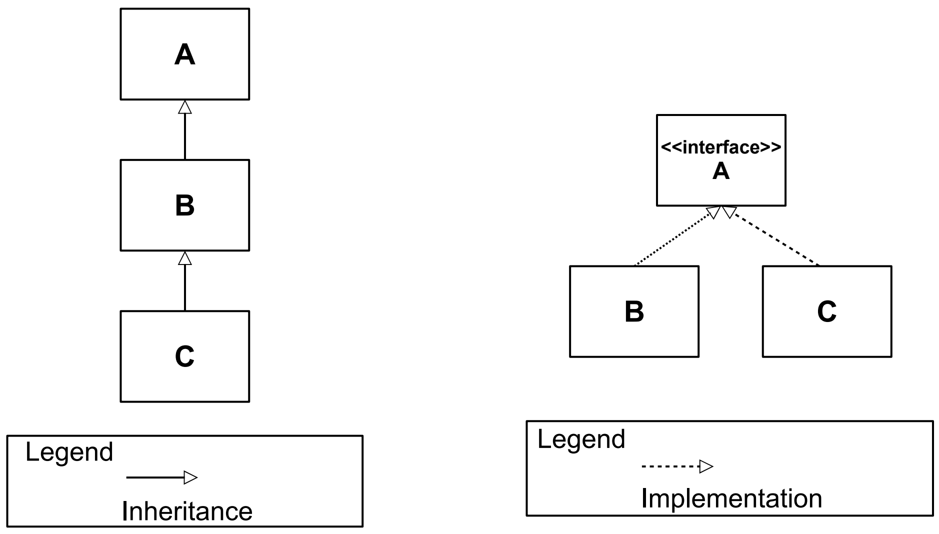
With Soot, call graphs can be generated using these different techniques, the computational complexity being the least in the case of class hierarchy analysis and the most in the case of variable type analysis. We tried the class hierarchy analysis to extract the call graph from a single sample *class* file on our local machines and it took almost 50 seconds to generate the call graph, the maximum heap size being specified as 800 MB. Without specifying the heap size, it ran out of memory. Then we went on and tried to generate the call graph for our sample small size projects in whole program mode and even with 2 GB of maximum heap size specified, it ran out of memory. Considering this performance, the call graph generation process of Soot did not seem feasible in the case of large-scale enterprise systems at all.

The amount of time needed by Soot is due to the fact that when it begins an analysis from a particular class, it loads that class into memory and then subsequently loads all the classes that are directly or transitively referenced by that class, in addition to carrying out all the computations. And thus, when executed, it also needs all those classes to be present in its classpath. As a result, despite being an excellent tool and incorporating excellent techniques like class hierarchy analysis, rapid type analysis, variable type analysis etc., Soot was empirically unable to run successfully in our specific problem domain.

Fortunately, by exploring other existing tools, we came across the `Dependency Finder` toolset [Tes10a] which is actually a suite of tools for analyzing Java bytecode. Among its tool suite, two were of special interest to us – `Dependency Extractor` and `ClassReader`. `Dependency Extractor` generates XML containing information specifically pertaining to dependencies, but lacks some useful information like inheritance, invocation type (`interfaceinvoke, virtualinvoke, specialinvoke, staticinvoke`), field access type (`getfield, putfield, getstatic, putstatic`) etc. So we decided to use `ClassReader` which generates an equivalent one-to-one representation of the bytecode containing all the information we need.

3.1.2 The Dynamic Binding Problem

The **dynamic binding** problem (also known as the dynamic dispatch or virtual method problem), is the process of mapping a message to a specific sequence of code (method) at runtime. This is done to support the cases where the appropriate method cannot be determined at compile-time (*i.e.* statically) [wik11]. Dynamic dispatch is needed when multiple classes contain different implementations of the same method. This can happen because of *class inheritance* and *interface implementation*. Figure 3.2 shows the two scenarios.



(a) (b)

Figure 3.2: (a) Class Inheritance (b) Interface Implementation.

Method invocations (*e.g.* invokeinterface, invokespecial, invokestatic, and invoke-virtual in the Java context) are candidates for dynamic binding, meaning that compile time calling of a method might cause calling of another method at runtime, due to class inheritance, interface implementation and method overriding. Consider the situation in Figure 3.3 where classes B and C override class A's method $m()$. Although statically all three calls are to $A.m()$, dynamically they redirect to $A.m()$, $B.m()$ and $C.m()$, respectively.

```
...
A a = new A();
a.m(); //call to A.m()
a = new B();
a.m(); //call to B.m()
a = new C();
a.m(); //call to C.m()
...
```

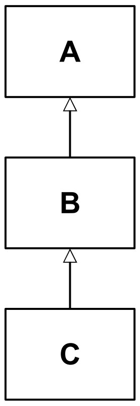


Figure 3.3: Dynamic Binding Example 1

As a second example, consider Figure 3.4 where classes B and C do not override class A's $m()$ method. Here, the compile time call to $C.m()$ redirects to $A.m()$ at runtime.

```
...
C c = new C();
c.m(); //call to A.m()
...
```

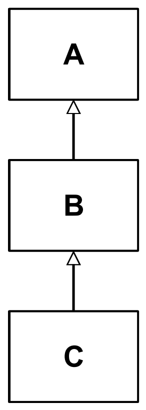
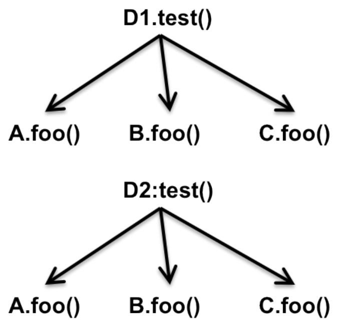


Figure 3.4: Dynamic Binding Example 2

One way to handle dynamic binding statically is to include all classes from the inheritance hierarchy, as in `Class Hierarchy Analysis` (CHA) [BS96]. The drawback of this approach is the huge number of redundant call edges that might result: it creates an edge from each caller of a method m to every possible instance of m. Consider Figure 3.5 where the edges to $C.foo()$ are redundant because only $A.foo()$ and $B.foo()$ have real bodies defined.

```
Class A{
        public void foo(){
        ...
        }
}
Class B extends A{
        public void foo(){
        ...
        }
}
Class C extends B{
        //does not override foo()
}
Class D1{
        public void test(){
                A a = new A();
                a.foo();
        }
}
Class D2{
        public void test(){
                A a = new B();
                a.foo();
        }
}
```

(a) Sample code segment



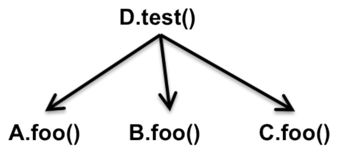
(b) Graph generated

Figure 3.5: Conservative Analysis Example 1

Similarly, in Figure 3.6 the edge to $B.foo()$ is redundant because A is actually the closest transitive superclass of C that has a body of method $foo()$ defined.

```
Class A{
        public void foo(){
        ...
        }
}
Class B extends A{
        //does not override foo()
}
Class C extends B{
        //does not override foo()
}
Class D{
        public void test(){
                C c = new C();
                c.foo();
        }
}
```

(a) Sample code segment



(b) Graph generated

Figure 3.6: Conservative Analysis Example 2

Table 3.1 shows the results of this approach on a Java library. In this library there was an interface with more than 50,000 transitive subclasses. If there were, say, 100 callers of a method of this interface, 5 million edges would be generated. In practice we found that only a few dozen of the transitive subclasses would define a particular method, and a more precise analysis could eliminate perhaps 99% of these edges.

Class or Interface Name	**Transitive Subclasses**
oracle.jbo.XMLInterface	53,934
oracle.jbo.server.TransactionListener	40,786
oracle.jbo.Properties	36,487
oracle.jbo.VariableManagerOwner	36,458
oracle.jbo.ComponentObject	36,449
oracle.jbo.common.NamedObjectImpl	36,370
oracle.jbo.server.NamedObjectImpl	36,105
oracle.jbo.server.ComponentObjectImpl	36,104
oracle.jbo.server.TransactionPostListener	33,181
oracle.apps.fnd.framework.OAFwkConstants	30,979

Table 3.1: Top 10 classes/interfaces with highest number of transitive subclasses/interfaces

Some techniques like `Rapid Type Analysis` (RTA) and `Variable Type Analysis` [BS96] do exist to tackle this problem and we tried these approaches using the tool Soot [LBL$^+$10], but had to abandon it due to excessive memory consumption and memory overflow problems. These approaches turned out to be unsuitable for our huge domain. This is one of the reasons we resorted to a new technique which we called *access dependency analysis.*

Consider Figure 3.7. The graph shown is the dependency graph resulting from the access dependency analysis of the code shown in Figure 3.5a. Note that since $C.foo()$ has no real body of its own, it is not in the graph. We only consider the overridden versions of methods during the addition of extra edges for handling dynamic binding, which reduces the number of edges. Also instead of adding call edges from $D1.test()$

and $D2.test()$ to $B.foo()$, we add an edge from $A.foo()$ to $B.foo()$. What this edge implies is that a compile time call to $A.foo()$ *might* result in a runtime call to $B.foo()$. This kind of edge reduces the number of edges even further because each additional caller only increases the number by one (like the edge from $D2.test()$ to $A.foo()$).

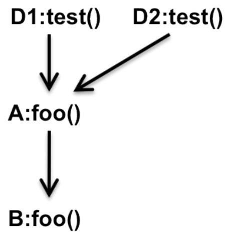


Figure 3.7: Access Dependency Analysis Example 1

As for the second example, consider Figure 3.8 where the graph shown is the dependency graph resulting from the access dependency analysis of the code shown in Figure 3.6a. Here, since A is the closest transitive superclass of C for the function $foo()$, a compile time call to $C.foo()$ redirects to $A.foo()$, and we do not include $B.foo()$ in the graph. The result is, once again, a reduced number of edges.

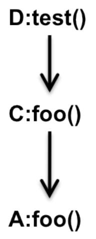


Figure 3.8: Access Dependency Analysis Example 2

We see that we get an efficient dependency graph because (i) links for each overridden method are only included for the actual overrides and (ii) the size of the graph grows linearly with the number of callers.

As mentioned earlier, a class with over 50,000 transitive subclasses was found to have only a few dozen of them overriding a particular method. Using our access dependency analysis, we only get a few hundred edges (rather than almost 5 million edges generated by a more conservative analysis). Since the number of edges are reduced, we also get rid of the memory overflow problem we faced in applying other existing approaches.

3.1.3 Access Dependency Graph

Considering everything we have discussed so far, we formally describe our concepts of the **access dependency graph** in this section. Below are the criteria we take into account while building our dependency graph using access dependency analysis:

1. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using any of `invokeinterface, invokestatic, invokespecial` and `invokevirtual`, then we add the following edge to the dependency graph:

   $A : a() \to B : b()$

2. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using either of *invokeinterface* or *invokevirtual*, and B has transitive subclasses or implementations $B_1, B_2 \ldots B_n$ explicitly implementing or overriding $b()$ as its own version, then we add the following edges to the dependency graph in addition to the edge described in criterion 1:

$B : b() \rightarrow B_1 : b()$

$B : b() \rightarrow B_2 : b()$

$\vdots$

$B : b() \rightarrow B_n : b()$

In addition, if B is a class that inherited method $b()$ from some other class but does not override $b()$ itself, then we add the following edge to the dependency graph:

$B : b() \rightarrow S : b()$

where S is the closest transitive superclass of B up the inheritance hierarchy.

3. For any two classes A and B (where A and B could possibly be the same class), if A's method $a()$ accesses B's field b using any of *putfield, getfield, putstatic* and *getstatic*, then we add the following edge to the dependency graph:

   $A : a() \rightarrow B : b$

   In addition, if b is a static field, we also add the following edge to the dependency graph:

   $A : a() \rightarrow B :< clinit > ()$

   where $< clinit >$ is the bytecode method representing the static initializers of the class.

Notice that every sensible path that could have been traversed by the conservative analysis is also traversable by the access dependency analysis, but possibly with several orders of magnitude fewer number of edges. By sensible, we mean a path that is worth traversing (for example, if a class does not override a certain method, it is not sensible to traverse a path to that method of that class). This means that the transitive closure of the sensible accessor-accessee relation pairs are the same for the conservative analysis and the access dependency analysis. So we have the same sensible reachability in the access dependency analysis as we would have had in the more conservative analysis.

3.2 Dynamic Analysis

Dynamic analysis operates by executing a program and observing the executions. The dynamic information consists of execution data for a specific set of program executions, such as executions in the *field*, executions based on an *operational profile*, or executions of *test cases*. Apiwattanapong *et al.,* [Api05] defines the dynamic impact set to be the subset of program entities that are affected by the changes during at least one of the considered program executions. Testing and profiling are standard dynamic analyses. Dynamic analysis is precise because no approximation or abstraction need be done: the analysis can examine the actual, exact run-time behaviour of the program. There is little or no uncertainty in what control-flow paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest. Dynamic analysis can be as fast as program execution. Some static analyses execute quite quickly, but in general, obtaining accurate results entails a great deal of computation and long waits, especially when analyzing large programs.

Furthermore, certain problems, such as pointer or alias analysis, remain beyond the state of the art; even exponential-time algorithms do not always produce sufficiently precise results. By contrast, determining at run time whether two pointers are aliased requires a single machine cycle to compare the two pointers (somewhat more, if relations among multiple pointers are desired) [Ern03].

Dynamic analysis, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system because it exposes the system's actual behaviour [CZvD+09]. This picture can range from class-level details up to high-level architectural views [RD99] [SAG+06] [WMFB+98]. Among the benefits over static analysis are the availability of runtime information and, in the context of object-oriented software, the exposure of object identities and the actual resolution of late binding. A drawback is that dynamic analysis can only provide a partial picture of the system, i.e., the results obtained are valid only for the scenarios that were exercised during the analysis.

Dynamic analysis is typically comprised of the analysis of a system's execution through interpretation (e.g., using the Virtual Machine in Java) or instrumentation, after which the resulting data are used for such purposes as reverse engineering and debugging. Program comprehension constitutes one such purpose and, over the years, numerous dynamic analysis approaches have been proposed in this context, with a broad spectrum of different techniques and tools as a result. Dynamic analyses are widely used in *program analysis* and *program comprehension*, which can be traced back to as early as 1972. It can be used in program visualization, feature location, frequency spectrum analysis, trace analysis, and in this work we are interested in applying dynamic analysis in the field of impact analysis [Api05] [LR03a] [LR03b] [BDSP04] [HS07].

Dynamic analysis consists of two major steps:

- instrumentation. One can simulate, emulate, translate the program code, or execute the program code directly.

- dynamic information collection. Corresponding to the way of instrumenting the code, dynamic information to be collected can be executed event traces, executed tests in the test suite, executions in the field, executions based on an operations profile. Then the collected dynamic information is used to compute the impact set.

Let P be the program that we want to preform impact analysis on, M the set of all functions within the program, C the set of changes that are made. We want to compute I - a subset of total number of functions M that may be affected by C. A technique is needed to constrain both the instrumentation required to collect the data and the data collected for each execution.

One can add instrumentation at different levels and to different extents based on the context in which P is used. The way of conducting instrumentation affects both the "cost-effective" and "safety-precision" trade-offs. In the following subsections, we discuss the feasibility of applying different types of instrumentation to enterprise systems.

3.2.1 Coverage Execution

The reason we instrument the code is to gather information about the *coverage* of either basic blocks[4] or methods[OAH03]. The instrumentation can be accomplished by collecting **field data**, **in-house/synthetic data** or executing a **test suite**. Assume one collects a set of execution data E for each use of the program, then the data for each execution contain the coverage information related to that execution (the set of entities traversed in the execution as expressed by e). At the conclusion of an execution, for each entity change c in C, we expect to identify a dynamic slice based on the execution data that traverse c. Then the impact set I is the union of the slices computed for each c in C.

In Figure 3.9, assume we can collect **coverage** information from deployed instances of the software for use in creating user profiles, determining classes of users

[4]A *basic block* is a sequence of statements in which control enters at the first statement and exits at the last statement without halting or branching, except at the end.

of the software, and assessing the costs and identifying the issues associated with collecting field data (*Gamma* approach [AH02] [AB93]). To use field data, a technique must constrain both the instrumentation required to collect the data and the data collected for each execution.

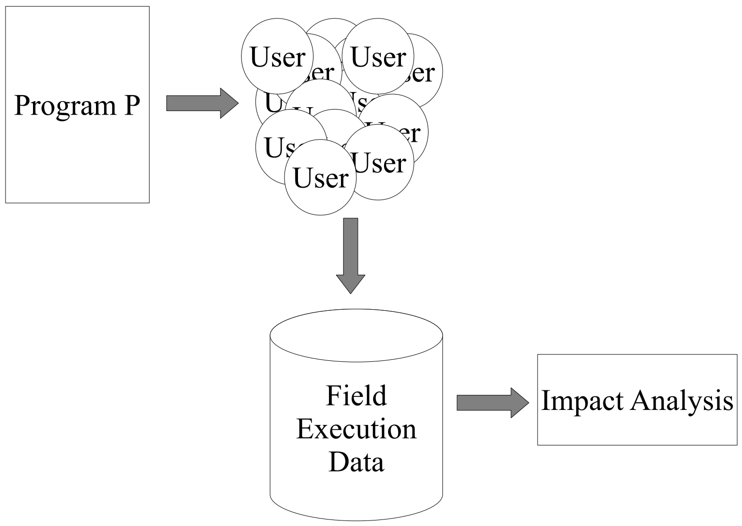


Figure 3.9: Field Data Execution Overall

The field execution data in Figure 3.9 is the executed coverage data E at the method level. An illustrative table of this information is drawn in Table 3.2.

	M_1	M_2	M_3	M_4	M_5	M_6	...
User A_1	X			X			
User A_2		X	X	X	X		
User B_1			X			X	
User B_2		X					
User C_1			X			X	
User...							

Table 3.2: Field Execution Example

Table 3.2 keeps records of users' multiple executions of program methods in M over time. If a method was traversed in one execution, a "cross" is marked in the coverage matrix. The steps to compute the impact set are:

- Step 1. Identify user executions through methods in the set of changes C, and fill the coverage matrix with methods covered by such executions. A set of covered methods CM is obtained.
- Step 2. *Static forward slice* from C to obtain the slice of the program SL that is associated with C.
- Step 3. Intersect the covered methods CM and the forward slice SL to obtain the impact set I.

In Step 2, forward slicing determines, for a point p in program P and a set of variables V, those statements in P that may be affected by the values of variables in V at p [Tip95].

On the other hand, if a test suite is accessible, we could augment the coverage matrix as in Table 3.3.

	M_1	M_2	M_3	M_4	M_5	M_6	...
Test case #1	X			X			
Test case #2		X	X	X	X		
Test case #3			X			X	
Test case #4		X					
Test case #5			X			X	
Test case ...							

Table 3.3: Test Execution Example

The test coverage approach does not require a record of users' executions of the program, but each test case in the test suite has to be executed to collect the coverage information. Other than that, the two approaches are exactly the same.

Coverage-based instrumentation that collects dynamic data from either field execution or test case execution, is not suitable because:

- Field execution requires the recording of users' executions through methods in the set of changes C. Since methods can be newly added to the set of changes to introduce new functionalities, after changes have been applied to the system, there may be no way to collect the information concerning executions of the new methods. In addition, the coverage information in the collection will be inaccurate. Even for modified methods, it is still uncertain that users' executions can cover all the changes. Thus, the uncertainty of covering all the changes may greatly increase the risk of missed impacts. The success of collecting coverage information from executing real test cases is very much dependent on the comprehensiveness of the test suite. Many software failures have been caused by *low probability-high impact* conditions within the code, and the test suite may not inclusively capture all these conditions, not to mention that usually the test suite reflects only a limited part of the enterprise systems in an organization.

- Due to the size of enterprise systems, it is extremely unrealistic to collect users' executions in a continuous way. Orso *et al.,* [OAH03], with the aid of 12 users, spent 12 weeks to collect 1,100 executions on a program `JABA` that consisted of 550 classes and 2,800 methods. For enterprise systems, the number of classes can easily exceed 100,000, which is nearly 200 times the size reported in the work on JABA. To collect user information on such systems is very time consuming and may take even longer than rerunning the entire regression test suite. Thus, we need an automated way of instrumenting enterprise systems. Executing test cases to collect coverage information is not realistic, since rerunning all the tests is exactly what organizations are already doing with system upgrades/customized changes. Even though change impact analysis may provide additional useful information that may be viewed as a risk management strategy, it is likely that companies will not choose to undertake additional impact analysis after re-running all their regression tests. One of our main goals in applying impact analysis on enterprise systems is to save on having to re-run every test in the regression suite. Instead, companies will be able to re-run only those tests that are affected by the changes that are made to the system.

3.2.2 Program Execution Traces

Rather than coverage information, one can instrument the program to collect execution traces [LR03a]. Assume we have a call graph of the program in Figure 3.10. Associated execution traces are shown in Figure 3.11a, in which upper case letters denote method names, "r" denotes a return, "x" denotes a system exit, in the order in which they occur during execution. In this example there are three consecutive execution traces, and each one is marked by an underline.

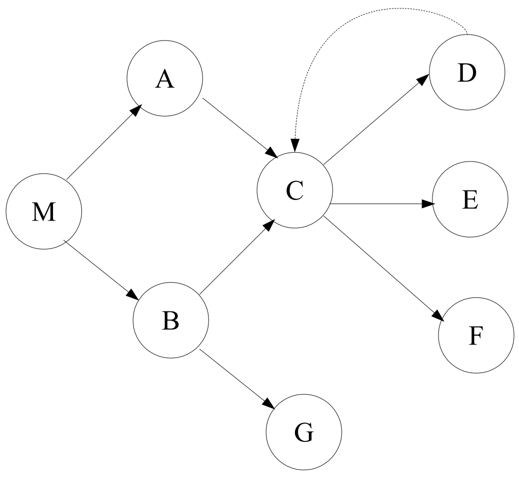
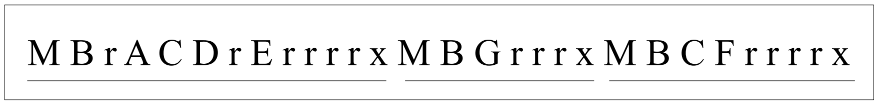
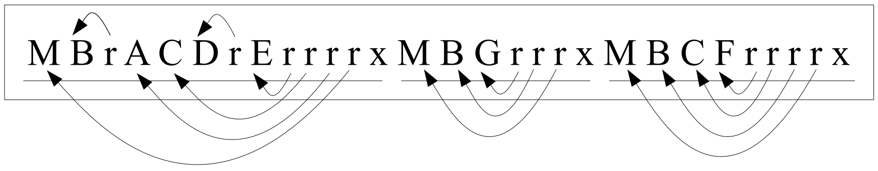


Figure 3.10: Call Graph Example



(a)



(b)

Figure 3.11: (a) Multiple Executions Example (b) Multiple Executions Example with Return Information

Law and Rothermel [LR03b] introduced the `PathImpact` approach that can be summarized as follows: if we propose to change procedure p, we concern ourselves only with impacts that may propagate down any (and only) dynamic paths that have been observed to pass through p. Therefore, any procedure that is called after p, and any procedure which is on the call stack after p returns, is included in the set of potentially impacted procedures. So in our case, suppose there is a change made to method X, we can estimate the dynamic impact of the change relative to these three executions by searching *forward* in the traces to find methods that are called directly or indirectly by E and methods that are called after X returns. By searching *backward* in the traces we can discover the methods X returns to.

Figure 3.11b is an excellent depiction of this approach showing how the "return" information is added to Figure 3.11a. Consider a change made to method E, then for each trace, we "walk" forward to find methods that can be directly or indirectly called by E and methods called on the stack after E returns - none; we "walk" backward to find methods E can return to - $\{A, C, E, M\}$ for the first trace and none for the other traces. The union of impacts identified for each trace constitutes the impact sets $\{A, C, E, M\}$ for the executions we have collected.

Note that in the backward walking, methods that have immediate returns are not considered to be candidate impacted methods. For instance, in the first trace, method B and D have immediate returns after being called, so a change to E will not affect either B or D at all, since E can never return to them. When method E returns, B or D has already returned. We summarize the impacted methods associated with each changed method, in Table 3.4.

Methods	**Total Impacts**
M	$\{A, B, C, D, E, F, G, M\}$
A	$\{A, C, D, E, M\}$
B	$\{A, B, C, D, E, F, G, M\}$
C	$\{A, B, C, D, E, F, M\}$
D	$\{A, C, D, E, M\}$
E	$\{A, C, E, M\}$
F	$\{B, C, F, M\}$
G	$\{B, M, G\}$

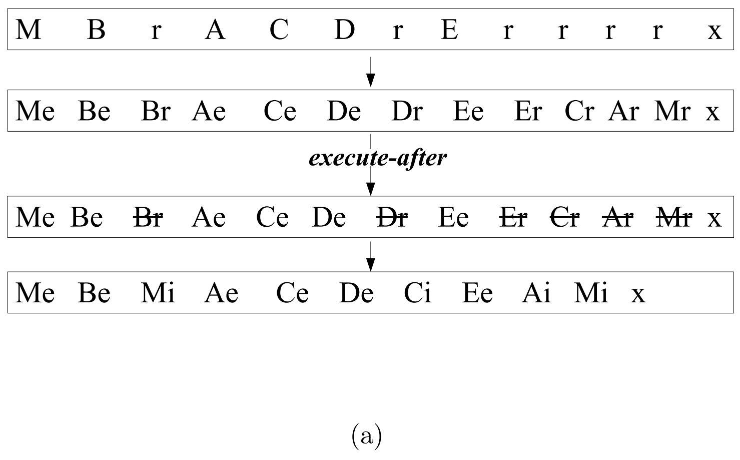
Table 3.4: Impact Set Computation

An obvious difficulty with this approach involves tracking executed paths, since traces of this sort may be long. A compression algorithm can be used to reduce the size of the trace that is collected. For instance, the `SEQUITUR` algorithm [NMW97] is used to examine a trace and remove redundancies in the observed sequence of events by creating a grammar that can *exactly* regenerate the original trace. `SEQUITUR` is an online algorithm - this means it can process traces as they are generated, rather than requiring the entire trace to be available. Larus reports this compression algorithm can reduce a 2 GB trace to approximately 100 MB [Lar99].

Even with the aid of a compression algorithm, `PathImpact` is still very expensive - in terms of execution overhead or the amount of dynamic information collected - in both time and space. It requires a time that depends on the size of the trace analyzed to compress traces at both method entry and return, and the space cost is proportional to the size of the traces (which can be very large) [Api05]. Apiwattanapong *et al.,* proposes a more efficient and practical approach to collect **Execute-after** information. They identified the essential information needed to perform dynamic impact analysis is the execute-after (EA) information: for a program p and a set of executions E, the only information required is whether, for each pair of entities e_1 and e_2 in P, e_2 was executed after e_1 in any of the executions in E. Or more formally:

Given a program P, a set of executions E, and two methods X and Y in P, $(X, Y) \in EA$ for E if and only if, in at least one execution in E,
1. Y calls X (directly or transitively), 2. Y returns to X (directly or transitively), or 3. Y returns to a method Z (directly or transitively), and method Z later calls X (directly or transitively).

Execution traces contain mostly redundant information (see Table 3.4). By considering only method-entry and method-returned-to events, the first event trace in Figure 3.11a can be rewritten as Figure 3.12a. By further inspection, only the first and last events for each method are needed, such that a further reduced trace is obtained as in Figure 3.12b.


M B r A C D r E r r r r x
Me Be Br Ae Ce De Dr Ee Er Cr Ar Mr x
execute-after
Me Be ~~Br~~ Ae Ce De ~~Dr~~ Ee ~~Er~~ ~~Cr~~ ~~Ar~~ ~~Mr~~ x
Me Be Mi Ae Ce De Ci Ee Ai Mi x
(a)

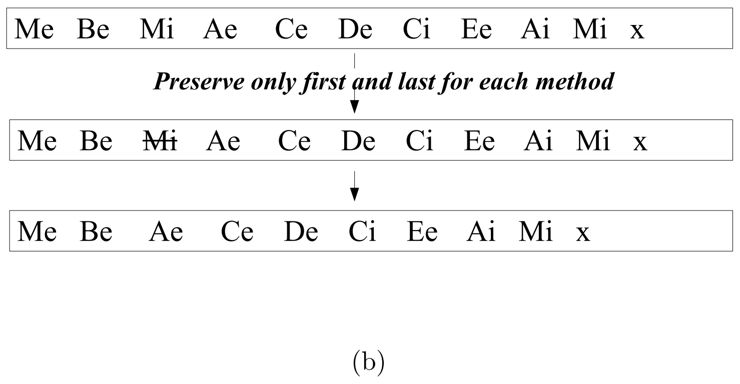

Me Be Mi Ae Ce De Ci Ee Ai Mi x
Preserve only first and last for each method
Me Be ~~Mi~~ Ae Ce De Ci Ee Ai Mi x
Me Be Ae Ce De Ci Ee Ai Mi x
(b)

Figure 3.12: (a) Execution Trace processed by Execute-after I(b) Execution Trace processed by Execute-after II

An empirical study [Api05] showed `CollectEA` is as precise as `PathImpact` and only slightly more expensive than `CoverageImpact`. The study used two programs SIENA and JABA as a case study. The results reported were that the analysis took approximately 2 minutes for SIENA and approximately 8 minutes for JABA.

Program	**Version**	**Classes**	**Methods**	**LOC**	**Test Cases**
SIENA	8	24	219	3,674	564
JABA	11	355	2,695	33,183	215

Table 3.5: Subject Programs

Comparing the precision[5] between CollectEA, CoverageImpact and PathImpact showed that CollectEA can be as precise as PathImpact, which is approximately 22% for both SIENA and JABA.

It has been empirically demonstrated that path-based impact analyses can achieve higher precision, albeit with slightly more overhead, than coverage-based approaches. However, to the best knowledge of the author, these approaches still have limitations and disadvantages that make them unrealistic for enterprise systems:

- The precision metric used in evaluating CollectEA is calculated by dividing the computed impact set by the set of total methods. Depending on the portion of the actual impact the approach can cover, the metric can be misleading. For instance, assume we want to compare the precision of the impact sets computed by two impact analysis tools, $Tool_1$ and $Tool_2$. Consider the two scenarios shown in Figure 3.13,

[5]The precision used in this empirical study is calculated by I/M, where I is the number of estimated impacts, M is the number of all entities in the program.

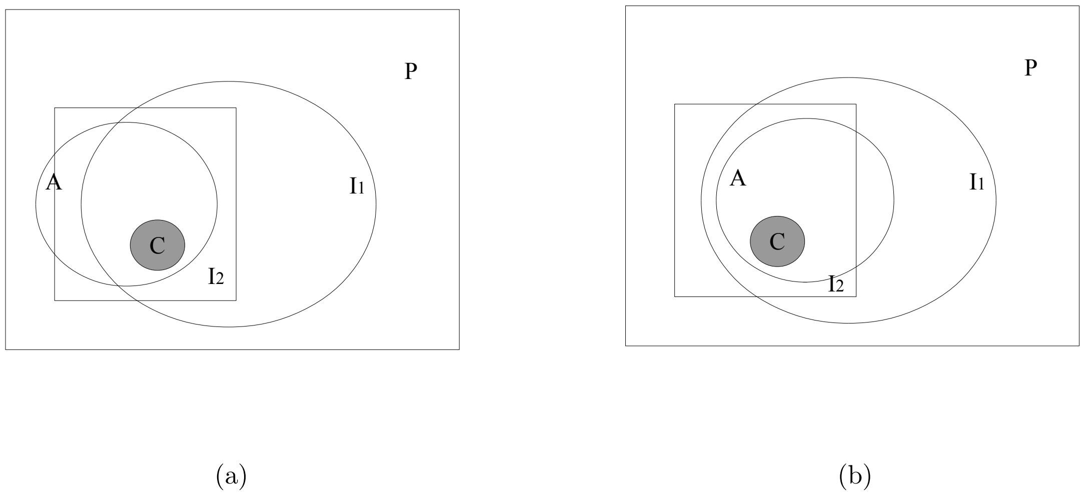


Figure 3.13: (a) Scenario I (b) Scenario II

In both scenarios, sets C, A, P, I_1, I_2 are: the set of original changes; the set of actual impacts; the total number of functions in the program; computed impacts by $Tool_1$; computed impacts by $Tool_2$, respectively. Rectangles and ellipses were drawn for better differentiation.

In scenario I, we observe that $Tool_2$ is more precise than $Tool_1$ since $Tool_2$ computes a smaller impact set I_2 and $Tool_2$ is more accurate in terms of coverage of the actual impacts. In scenario II, I_2 is also smaller than I_1, but just saying $Tool_2$ is more precise than $Tool_1$ is misleading, since in practice, the use of $Tool_1$ will reduce the risk of false-negatives. For enterprise system users, $Tool_1$ is more likely to be selected to reduce risks rather than $Tool_2$, though the precision metric used in this path-based approach calculates a "better" precision for $Tool_2$. In other words, a better approach should consider both the precision and the recall, not only because it is dangerous to miss false-negatives, but also because the analysis result can be totally misleading if only the precision is considered.

- It is very expensive to conduct path-based analysis in terms of both time and space. Even with a compression algorithm like SEQUITUR to reduce duplicated traces or execute-after inspection to collect only "return into" information, these analyses do not seem to reduce a significant number of the event traces. In Table 3.5 we assume the required running time of the analysis is proportional to the number of methods, and the running time for an enterprise system consisting of 4 million methods may reach over 600 hours (25 days). Additionally, time complexity of the analysis on an enterprise system cannot be linear, due to the increase of complexity while adding more methods. The author tried the CollectEA approach on an enterprise system that contains 4.6 million methods and 10 million LOC. The analysis kept running for a couple of days and eventually ran out of memory on a computer with 8 GB memory. We later realized that the scale of our target system is way beyond this tool's capability to analyze, even with much more memory. One of the reasons is that this approach still computes too many event traces. By comparing the last trace in Figure 3.12b and the very first trace in Figure 3.11a, we see that only 3 out of 13 events in the trace were removed.

- More importantly, in collecting executed event traces, there are types of traces/-paths that existing impact analysis approaches do not classify and remove appropriately. *Data-flow based testing*, which annotates control-flow graphs with the mode of use of particular variables, has been widely explored in *Regression*

Test Selection (RTS). By analysis of the definition and use of data in execution paths, we can select tests that exercise particular kinds of usage. A variable can be defined, killed (deallocated) and used. Thus some data-flow strategies are: all paths, all du-paths[6], all uses, all predicate use/some computation, all computation use/some predicate, all definition, all predicate and computation uses with a descending order of strength [PA98]. Among those testing criteria, surprisingly little work has been done on applying it to assist change impact analysis. If one can classify paths by whether they can be affected by a potential change, then presumably a large portion of the traversed paths can be removed from the impact set.

3.2.3 Aspect-Based Instrumentation

Aspect-oriented programming (AOP) [KLM+97] has been proposed as a technique for improving separation of concerns in software. AOP builds on previous technologies, including procedural programming and object-oriented programming, which have already made significant improvements in software modularity.

Kiczales *et al.,* [KHH+01] pointed out the central idea in AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Instead, it is believed that in the implementation of any complex system, there will be concerns that one would like to modularize, but for which the implementation will instead be spread out. This happens because the natural modularity of these concerns crosscuts the natural modularity of the rest of the implementation.

[6]A definition path (du-path), with respect to a variable v is a path whose first node is a defining node for v, and its last node is a usage node for v.

AOP does for concerns that are naturally crosscutting what OOP does for concerns that are naturally hierarchical. It provides language mechanisms that explicitly capture crosscutting structures. This makes it possible to program crosscutting concerns in a modular way, and thereby achieve the usual benefits of modularity: simpler code, that is easier to develop and maintain, and that has greater potential for reuse. Crosscutting concerns such as error checking and handling, synchronization, context-sensitive behaviour, performance optimizations, monitoring and logging, debugging support, and multi-object protocols can be modularized in a clean way [Asp14].

The motivation for **AspectJ** [Pro14] (and likewise for aspect-oriented programming) is the realization that there are issues or concerns that are not well captured by traditional programming methodologies. Consider the problem of enforcing a security policy in some application. By its nature, security cuts across many of the natural units of modularity of the application. Moreover, the security policy must be uniformly applied to any additions as the application evolves. Also, the security policy that is being applied might itself evolve. Capturing concerns like a security policy in a disciplined way is difficult and error-prone in a traditional programming language.

Concerns like security cut across the natural units of modularity. For object-oriented programming languages, the natural unit of modularity is the class. But in object-oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these are not reusable, they cannot be refined or inherited, they are spread throughout the program in an undisciplined way, in short, they are difficult to work with.

AspectJ, originally developed at Xerox Parc, is an implementation of the aspect-oriented programming paradigm for the Java language. It adds to Java one new concept, a *join point* and a few new constructs: *pointcuts*, *advice*, *inter-type declarations* and *aspects*. Pointcuts and advice dynamically affect program flow, while inter-type declarations statically affect a program's class hierarchy, and aspects encapsulate these new constructs. A join point is a well-defined point in the program flow. A pointcut picks out certain join points and values at those points. A piece of advice is code that is executed when a join point is reached. These are the dynamic parts of AspectJ. AspectJ also has different kinds of inter-type declarations that allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes. AspectJ's aspects are the units of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations.

Over the past few years, economic pressure has forced enterprises to outsource many IT services and purchase them from external service provides, and build customized code on top of the purchased software to constitute enterprise systems. The techniques we discussed in § 3.2.1 and § 3.2.2 are very costly in instrumenting the system to assist change impact analysis. An alternative approach aims to instrument the enterprise system components and platforms directly by aspect-oriented instrumentation. This **aspect-oriented instrumentation** helps to solve the shortcomings of existing instrumentation approaches by providing transparency for the application developer and furthermore offering the opportunity to monitor management data at any level of detail [DG03]. Instrumentation can be added on top of existing applications as some extra "drivers" to normal software in an ad-hoc manner, which has to be developed in parallel to the development of the normal software. However this approach only provides generic information about the system, such as outputs for a given set of inputs. If fine-grained monitoring is the goal - such as details of how components interact (control-flow) within the system, we need some transparency of those components and this is where aspect-oriented programming can help.

What follows is an introduction to the key features in AOP and AspectJ, and a discussion of how to conduct aspect-oriented instrumentation in an enterprise system, in particular with AspectJ.

The features in AspectJ [Pro14] are presented using a simple figure editor system (Figure 3.14). A Figure consists of a number of *FigureElements*, which can be either *Points* or *Lines*. The Figure class provides factory services. There is also a *Display*.

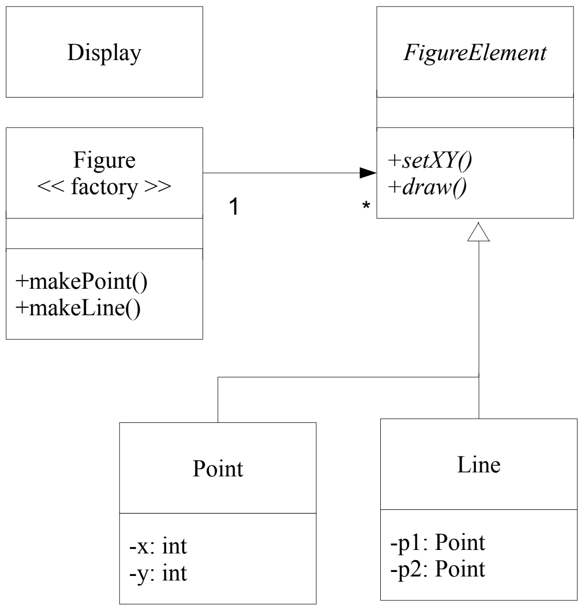


Figure 3.14: UML for the FigureEditor Example [Pro14]

Pointcuts

In AspectJ, *pointcuts* pick out certain join points in the program flow such as in the code segment below:

```
pointcut move(): call(void FigureElement.setXY(int, int)) ||
                 call(void Point.setX(int))               ||
                 call(void Point.setY(int))               ||
                 call(void Line.setP1(Point))             ||
                 call(void Line.setP2(Point));
```

the pointcut move() picks out each join point that is a call to one of the five void methods. Note that a pointcut can identify join points from many different types – in other words, they can crosscut types, by building the pointcut out of other pointcuts with *and*, *or*, and *not*. Then the programmer can simply use **move()** to capture this complicated pointcut.

Advice

After pointcuts pick out join points, we use *advice* to implement crosscutting behaviour. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). AspectJ has several kinds of advice: *before*, *after*, *after returning*, *after throwing*, *around* etc. This piece of advice:

```
before(): move() {
      System.out.println("about to move");
}
```

runs as a join point is reached, right before the program proceeds with the join point (actual method running). Pointcuts not only pick out join points, they can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations, such as this piece of advice:

```
after(FigureElement fe, int x, int y) returning:
        call(void FigureElement.setXY(int, int))
        && target(fe)
        && args(x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

exposes three values from calls to **setXY**: the target FigureElement – which it publishes as **fe**, so it becomes the first argument to the after advice – and the two int arguments – which it publishes as **x** and **y**, so they become the second and third argument to the after advice. So the advice prints the figure element that was moved and its new x and y coordinates after each setXY method call.

Inter-type declarations

Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. Unlike advice, which operates primarily **dynamically**, inter-type declarations operates **statically**, at compile-time.

Consider the problem of expressing a capability shared by some existing classes that are already part of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to each affected class a method that implements this interface. AspectJ can express the concern in one place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields with the existing classes.

Aspects

The definition of *Aspects* is very similar to classes; they wrap up pointcuts, advice, and inter-type declarations in a modular unit of crosscutting implementation. It can have methods, fields, and initializers in addition to the crosscutting members. Because only aspects may include these crosscutting members, the declaration of these effects is localized.

```
aspect Count
  {
    private int count = 0;
    pointcut CountSetXY() : call (FigureElement.setXY(int, int));
    before() : CountSetX()
    {
       count++; }
    }
  }
```

The aspect **Count** introduces *count* as a new member of FigureElement and defines a pointcut for the setXY method of FigureElement. The before advice increments the newly introduced count variable every time the setXY method is invoked.

Aspect-oriented Instrumentation Approach

In order to collect either field/test coverages or path executions in existing dynamic impact analysis, one has to instrument the program with a good understanding of the business requirements of the program, how the functionalities of the program were implemented, the execution order of different functions, input data and expected output data for the functions to be instrumented, etc. This requires comprehensive domain knowledge and may lead to enormous costs since many code pieces may have to be instrumented to complete one single trace in an enterprise system. In addition, we mentioned in § 3.2.1 and § 3.2.2 that both these ways of dynamic impact analysis exhibit a number of limitations and are inappropriate for our target system, not to mention that the instrumentation itself is beyond a testers' ability to implement without automated tools.

Unlike many other tools, AspectJ works at the **bytecode** level (source code is not required), hence it allows instrumentation on a third-party system, which is exactly what we want for enterprise systems since most of the organizations that use/augment enterprise systems only have an executable version from a vendor. Moreover, it does not require any modification of the existing code. The instrumentation code is encapsulated as an aspect which may be done by a different developer/tester who is familiar with the instrumentation environment, but not necessarily with the application logic. The application code is simply recompiled using a special compiler, the *aspect weaver*, that connects the aspect code with the application code. Thus, instrumentation can also easily be integrated into an existing application.

The idea is to define a pointcut on every method execution, as well as some advice to run before the code is executed when the pointcuts appear. Below is an aspect we are using to trace system executions:

Trace.java

```
package aspects;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.aspectj.lang.Signature;

aspect Trace{
   pointcut traceMethods() :
   (execution(* *(..))&& !cflow(within(Trace)));
   before(): traceMethods(){
       Signature sig = thisJoinPoint.getSignature();
       String line =""+
       thisJoinPointStaticPart.getSourceLocation().getLine();
       String sourceName = thisJoinPointStaticPart.
       getSourceLocation().getWithinType().getCanonicalName();
       Logger.getLogger("Tracing").log(
               Level.INFO,
               "Call from "
                   +  sourceName+" line " + line
                   +" to " +sig.getDeclaringTypeName() + "." +
                   sig.getName()
       );
 }

```

In `Trace.java`, we define a pointcut **traceMethods()** (Line 7) to pick out executions of every method in every class, as long as the control flow is not in the current class (Trace), such that we can identify all the other methods being called in each particular execution. Then we define an advice to be executed right before executing the method (Line 9). In the advice we log information of caller and callee when the pointcut was reached, including names and line numbers of the calling sites. AspectJ provides a special reference variable, **thisJoinPoint** (Line 10), that contains reflective information about the current join point for the advice to use. thisJoinPoint has a rich reflective hierarchy of signatures, and can be used to access both static and dynamic information about join points, such as the arguments of the join point. If only the static information is required, a special variable **thisJoinPointStaticPart** can be accessed instead to read the static part of the join point. However, run-time creation of the join point object is missed by using thisJoinPointStaticPart. Finally, the pointcut along with the advice are wrapped into the aspect Trace (Line 6).

To use this aspect, we need to compile it using AspectJ's compiler **ajc**:

```
ajc -outxml -outjar aspects.jar Trace.java
```

ajc is AspectJ's compiler and bytecode weaver for the Java language. The ajc command compiles and weaves AspectJ code together with Java source or .class files, producing .class files compliant with any Java VM (1.1 or later). It combines compilation and bytecode weaving and supports incremental builds; you can also weave bytecode at run-time using *Load-Time Weaving*[7].

Now we can use this compiled Jar file aspects.jar to run the instrumentation process:

[7]Please check the AspectJ's development guide at
https://www.eclipse.org/aspectj/doc/next/devguide/ltw.html

```
java -javaagent:<path to aspectjweaver.jar> -cp
<path to aspects.jar>:<path to target jar/folder>
<name of main class to run>
```

Example output of running this Jar on a class MGPApp.class from Oracle E-Business Suite is listed below:

Output.sample

```
[oracle@orasrv4 NetBeansProjects]$ java -javaagent:/ebs/orahome/
aspectj1.7/lib/aspectjweaver.jar -cp /ebs/orahome/aspects.jar:/ebs/
oracle/prodcomn/java/ MGPApp
Dec 7, 2013 10:30:58 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
INFO: Call from oracle.lite.sync.ConsNls line 37
to oracle.lite.sync.ConsNls.initialize
Dec 7, 2013 10:30:59 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
INFO: Call from MGPApp line 106 to MGPApp.main
Dec 7, 2013 10:31:00 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
INFO: Call from oracle.lite.web.util.JupMGPDebug line 134
to oracle.lite.web.util.JupMGPDebug.init
Dec 7, 2013 10:31:00 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
INFO: Call from oracle.lite.web.util.JupMGPDebug line 27
to oracle.lite.web.util.JupMGPDebug.load
Dec 7, 2013 10:31:00 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
INFO: Call from oracle.lite.common.Profile line 153
to oracle.lite.common.Profile.getBinDirectory
Dec 7, 2013 10:31:00 AM aspects.Trace ajc$before$aspects_Trace$1$b314f86e
....
```

By extracting from the output we can identify the dynamic event trace:

```
oracle.lite.sync.ConsNls.initialize
```

```
to MGPApp.main
to oracle.lite.web.util.JupMGPDebug.init
to oracle.lite.web.util.JupMGPDebug.load
to oracle.lite.common.Profile.getBinDirectory
to ...
```

In this section we presented an alternative aspect-oriented instrumentation approach using AspectJ, that allows us to conduct a fully dynamic instrumentation on the bytecode level of programs. Aspect-oriented instrumentations "weave" together the program code/bytecode and the aspects, and encapsulates advice (insertion code) to monitor and collect dynamic information, without modifying the program. Developers/testers can focus on the instrumentation and data collection, saving the effort required to understand the application logic. Additionally, memory usage and running time are quite reasonable: it requires memory of the order of hundreds of kilobytes per class, and running time of seconds per class. We include more detail of applying this approach to an enterprise system in Chapter 8, Empirical Study.

3.3 Combining Static and Dynamic Analysis[8]

We introduced both static and dynamic impact analysis in § 3.1 and § 3.2, with respect to enterprise systems. Static analysis computes a conservative set of impacts by analyzing the program, pre-runtime. It abstracts from the program a static representation, the representation can include various types of graphs: Call Graph, Control-Flow Graph, Dependency Graph, or even Trees, etc. Since it includes every possible program behaviour of the types targeted, it often results in over-estimation – many of the computed impacts are not real impacts.

Specifically, the portion of the real impacts that a static analysis may cover depends on many factors, such as how well the abstraction was implemented in the analysis. As we know, call graphs represent a program by abstracting a set of methods and a set of calling relations between them. With a call graph we can capture all the explicit method invocations even though many of them are not feasible. By feasible we mean the path of execution is feasible, with matched calls and returns or with the right dependencies. In terms of dependency, call graphs only capture calling dependencies, but data dependencies can also affect program behaviour. This leads us to focus on building dependency graphs to capture dependencies between all the entities (methods and fields).

Many current enterprise systems were written in an object-oriented language to encapsulate fields and methods, provide information hiding, increase program understanding, etc. However, in analyzing the impacts of a change, object-oriented

[8]This section is mostly based on the author's publication: Wen Chen, Alan Wassyng, and Tom Maibaum. "Combining Static and Dynamic Impact Analysis for Large-scale Enterprise Systems." The 15th International Conference on Product-Focused Software Process Improvement. Helsinki, Finland. 2014. [CWM14a]

features like inheritance and dynamic binding introduce significant complexity. As we mentioned in § 3.1.2, many of the over-estimated impacts are caused by dynamic binding, since until run-time we cannot determine the type of a method. For the sake of safety, one has to include all the possible superclasses in determining the dependencies. Hence we introduced a new dependency graph *Access Dependency Graph* to deal with dynamic binding and it effectively reduces the number of dependencies we used to include.

Recently, dynamic analysis has been used more extensively than static analysis, since dynamic analysis is more efficient, in terms of running time and precision in finding impacts, locating defects, etc. Dynamic analysis requires run-time executions of the program to collect information such as field data, coverage, event traces etc. Then software developers/testers can compute dynamic impacts of a change by identifying affected entities in the program. Even though the dynamic approach is more efficient, because one cannot instrument the program to cover all feasible executions, it often leads to under-estimation. The way we instrument the code, and the kind of dynamic information we collect can greatly affect the feasibility, precision and completeness of the analysis.

As we mentioned in § 3.2.1, coverage-based dynamic approaches require users' field data or actual test cases, which are costly and difficult to obtain. Even with this coverage information, impacts computed on them are obviously only a small portion of the actual impacts. Changes such as the addition of new functions can never be tested or manually executed before being applied, hence there is no chance to collect coverage information on them. For test cases, it is contradictory to collect dynamic information by performing a full regression test execution, since one of our objectives is to better focus testing in the face of changes – identifying affected tests in the original test suite without running the whole suite. In § 3.2.2, path-based dynamic approaches were introduced, in which event traces are collected. This sort of approach seems more practical and effective, however current path-based approaches have a number of limitations that lead to long running time and low precision: one has to have particular domain knowledge such as a full understanding of the application logic to conduct the instrumentation. Certain functions require certain inputs and order of execution to make the executions meaningful, which increases the difficulty and complexity of analyzing the impacts. Long execution time is mainly caused by duplicate traces, and even though compression algorithms or finer resolution analysis (*e.g.* execute-after) are employed, it still requires a large amount of time almost equal to that of running the entire regression suite.

Considering the pros and cons of static and dynamic analysis, and limitations of current approaches, there exists real potential to combine the two approaches, resulting in a hybrid approach that is safe and precise, and requires less execution time. Since our target systems are typically mission or company critical, the first and most important consideration is *safety*. By safety, we mean the computed impact set should be complete; in other words, it contains all the actual impacts with some over-estimated ones (false-positives). Therefore, in our approach, an access dependency graph G is built to capture all the potential impacts S and serves as the input source for dynamic analysis. Note that, unlike the *Vanilla* static analysis, our approach is more precise since we are able to eliminate many false-positives caused by dynamic binding.

Algorithm 1 Combination algorithm for static and dynamic analysis

```
function COMBINE($P$, $C$)
$G(V, E) \longleftarrow buildGraph(P)$;
for $c \in C$ do
    $Callers \longleftarrow ReverseSearch(G, c)$;
    for $i \longleftarrow 0$ to $Callers.length - 1$ do
        if $Callers[i] \notin S$ then
            $S \longleftarrow S + Callers[i]$;
        end if
    end for
end for
for $s \in S$ and s is executable do
    $Events \longleftarrow instrument(s)$;
    $Callers \longleftarrow Events.getMethods()$;
    for $j \longleftarrow 0$ to $Callers.length - 1$ do
        if $Callers[i] \notin D$ then
            $D \longleftarrow D + Callers[i]$;
        end if
    end for
end for
$PO \longleftarrow S - D$;
return $S, D, PO$
end function
```

In Algorithm 1[9], $G(V, E)$ is the access dependency graph we built in the static analysis stage, V and E is the set of *vertices* and associated *edges* within the graph. For each change extracted from the atomic changes set C, we perform *a reverse search* to identify all the possible callers of this method. A reverse search finds entities that can reach a particular method. Note that the changes taken into account in this algorithm are only direct changes – we need a *Change Analysis* to obtain both direct changes and indirect changes. Details of reverse searching and change analysis can be found in Chapter 4. Before adding the identified callers to the static impact set S, a duplication check is needed to avoid duplicates, as many methods may share the same callers. After that the static impact set serves as the input for dynamic analysis.

To start our dynamic analysis, we perform aspect-oriented instrumentation on each method in the static impact set S. Aspect-oriented instrumentation (§ 3.2.3) does not require any domain knowledge, nor any test data. In the aspect we create, we define a pointcut for each method execution, as well as advice in collecting event traces from the method. Since our static analysis covers all relevant feasible program behaviours, the dynamic impact set D will not exceed the range of S. In this way, we compute a set of dynamic impacts D which is actually a subset of S, as well as a set of *potential over-estimated impacts* ($PO = S - D$), that is, impacts that were not traversed by instrumentation. In Chapter 5 and Chapter 6, we examine how to further eliminate false-positives, in order to achieve better precision.

[9]The *ReverseSearch*() algorithm finds direct and indirect callers of c. The design of this algorithm can be found in Algorithm 2.

Chapter 4

Change Analysis[1]

As software evolves, there are incremental changes to an existing, perhaps large, set of code and documentation [MW00]. The changes can arise from multiple sources, such as user requirements, system upgrades, customizations, etc. Users often *have* to apply vendor *patches* to potentially fix issues or ensure continuing vendor support. In impact analysis, no matter what techniques or criteria are used, the set of atomic changes together with the program itself are the only two inputs. Changes can have indirect affects on other entities in the program. For instance, a change made to a database table may affect library functions that use it. Later this library function may behave differently since it is accessing a changed table. This type of change must also be taken into account while doing impact analysis. Hence, we need to extend the atomic change set by including indirect changes.

User requirement changes or customization changes are hard to obtain to use as a

[1]This chapter is mostly based on one of the author's technical reports at McMaster Centre for Software Certification (McSCert): Wen Chen, Asif Iqbal, Akbar Abdrakhmanov, Chris George, Mark Lawford, Tom Maibaum and Alan Wassyng. "Report 7: Middleware Change Impact Analysis for Large-scale Enterprise Systems" September, 2011. CIA$^+$11

case study. They are typically confidential and probably only touch a small portion of the system. Hence, our focus in this thesis is on **Software Patches**, which are easier to obtain and may have a large number of impacts on the system.

A patch can consist of multiple files, and depending on its type, a patch may update the application library or database (or both) of a system. Patches often contain a large number of different types of files, such as program code, SQL scripts, configuration files and documentation. It is necessary to distinguish between files that will change the program or database, and files that will not, and for the first category, be able to parse them to see which methods, tables, procedures etc., may be changed. Vendor documentation of the patch is typically inadequately detailed for this task. It is also better to rely on the source files themselves than on the accuracy and completeness of the documentation.

4.1 System Architecture

To better identify the indirect changes, one has to understand the architecture of an enterprise system and hence how the direct changes can affect other entities within the architecture. Figure 4.15 is the system architecture of Oracle E-Business Suite Release 12.1 [Cor10], which is one of the most popular existing enterprise systems. In the figure we can see the enterprise system is made up of the *database tier*, which supports and manages the Oracle database; the *application tier*, which supports and manages the various Oracle E-Business Suite components, and is sometimes known as the *middle tier*; and the *desktop tier*, which provides the user interface via an add-on component to a standard web browser.

A machine may be referred to as a node, particularly in the context of a group of computers that work closely together in a cluster. Each tier may consist of one or more nodes, and each node can potentially accommodate more than one tier. For example, the database can reside on the same node as one or more application tier components. Note, however, that a node is also a software concept, referring to a logical grouping of servers.

Centralizing the Oracle E-Business Suite software on the application tier eliminates the need to install and maintain application software on each desktop client PC, and also enables Oracle E-Business Suite to scale well with an increasing load. Extending this concept further, one of the key benefits of using the *Shared Application Tier File System* model (originally Shared `APPL_TOP`) is the need to maintain only a single copy of the relevant Oracle E-Business Suite code, instead of a copy for every application tier machine. On the database tier, there is increasing use of Oracle Real Application Clusters (Oracle RAC), where multiple nodes support a single database instance to give greater availability and scalability. The connection between the application tier and the desktop tier can operate successfully over a Wide Area Network (WAN). This is because the desktop and application tiers exchange a minimum amount of information, for example, only field values that have changed. In a global operation with users at diverse locations, requiring less network traffic reduces telecommunications costs and improves response times.

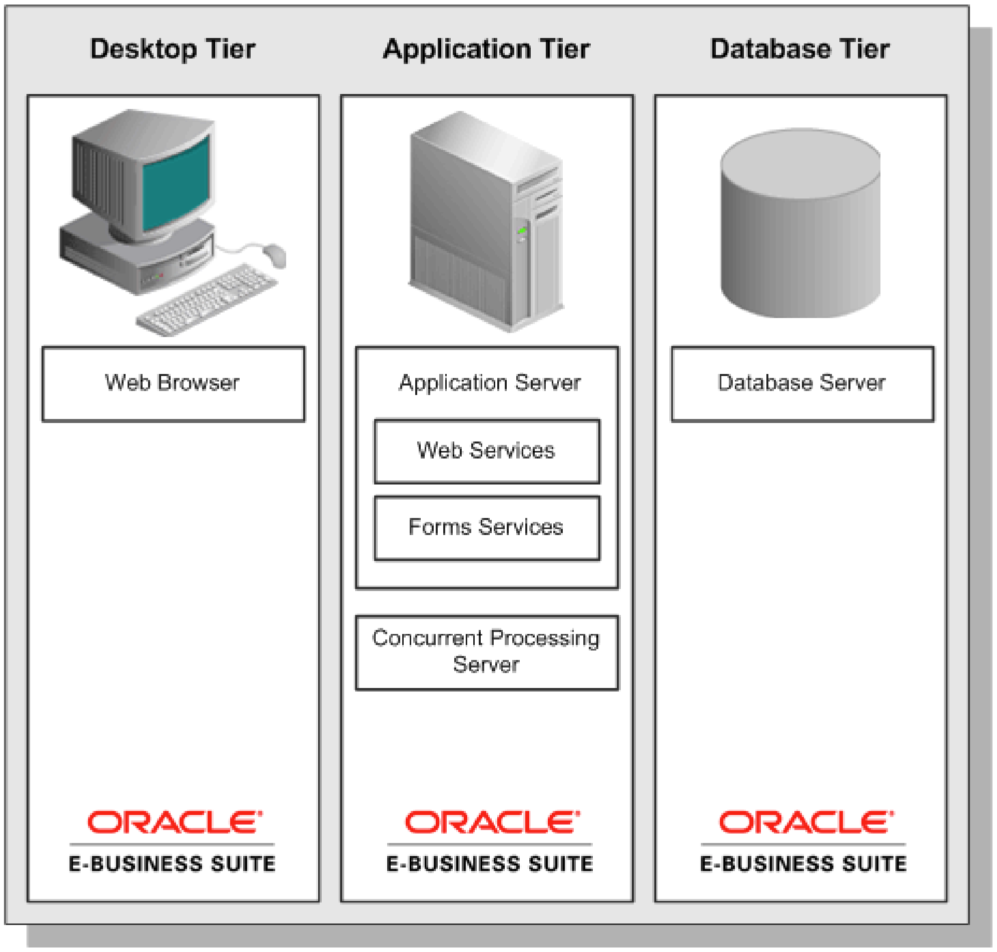


Figure 4.15: Oracle E-Business Suite System Architecture (Release 12.1)[Cor10]

To the author's best knowledge, most current enterprise systems incorporate components with this type of architecture. After implementing the system with an organization's business requirements, customized code can be built on top of it to provide customized service. For example, an organization may need to have transaction data pre-processed before sending it to *Forms Services* and process multiple transactions currently in the application tier. An architecture that contains both an enterprise system and customer applications is described in Figure 4.16.

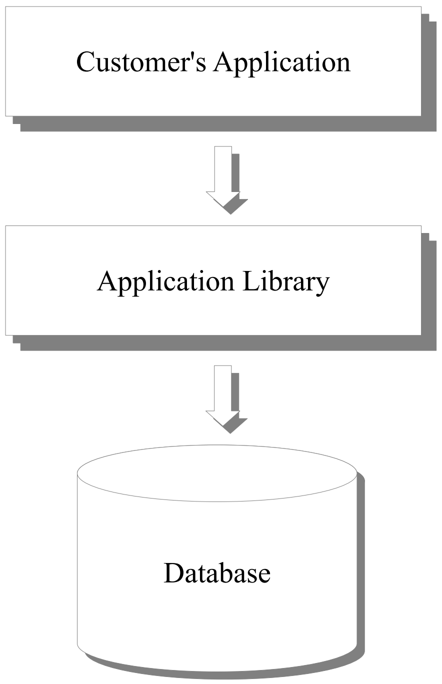


Figure 4.16: System Architecture

4.2 Analyzing Patches

In every patch, there should be log files or patch analysis tools that describe all potential changes made to the current version of the system at the file level. For example, in the patches for Oracle E-Business Suite, there are **35** types of files. Each type might impact database objects, library functions, or both. Some file extensions and their impacts are listed in Table 4.6. The second column of this table indicates what impacts the type of file in the first column can cause.

Object Type	**Impacts**
SQL	Database
PLS	Database
CLASS	Library
CMD	N/A
CTL	Database
DRV	Database, Library
FMB	Database
JSP	Library
PKB	Database
PKH	N/A
PL	Database
PLL	Database
PLS	N/A
PROPERTIES	N/A
RDF	Library
...	...

Table 4.6: Patch Files and Impacts

Note, some types of file extensions require a conversion from a compiled file to a readable file, such as FMB, PLL and RDF. A sample decompile command for a FMB file is as follows:

```
sh frmcmp.sh BATCH=YES Module Type=FORM Logon=NO Forms Doc=YES
Output_File= OUTDIR Module= INDIR
```

In this conversion, *Oracle Reports* decompiles FMB files to text files, such that statements that can affect the database can be extracted. We show the conversions in the figure below.

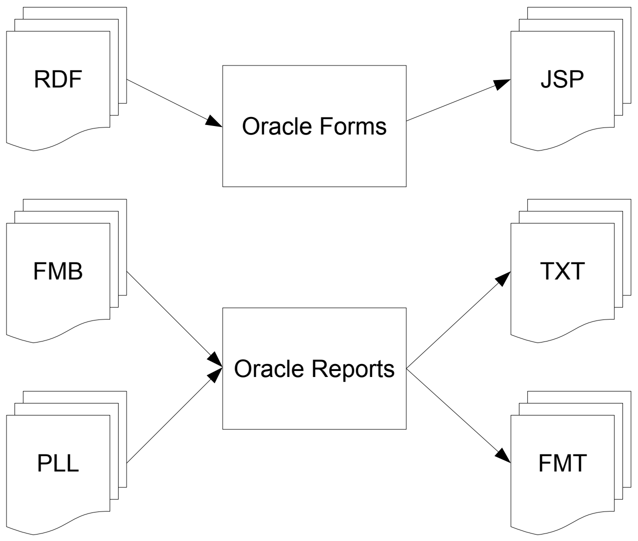


Figure 4.17: File conversions

RDF [Abd10] are Oracle Report binaries. Each report in Oracle is registered as a concurrent program executable. A single executable RDF file contains the data source, layout, business logic, and the language-specific prompts. These files potentially change applications. We used Oracle Report to convert RDFs to JSP scripts.

FMB [Abd10] are binary source code files created by the Oracle Forms tool at design time. Oracle Forms is a tool that enables rapid development of enterprise applications. In E-Business Suite, it is used to build screens that allow data to be entered and retrieved from the database as required. When the form is compiled by the Oracle Forms tool, an executable with the extension FMX is created [PA09].

FMB binaries may attach PL/SQL libraries or make calls to the database stored procedures. And therefore they potentially change applications. We used Oracle Forms to convert FMBs to TXT files which describe what forms are defined within them, and also for extracting PL/SQL scripts from them.

PLL [Abd10] are PL/SQL libraries that can be attached to a form. Think of a PL/SQL library as reusable code that can be shared among multiple forms. The code within the libraries can reference and programmatically modify the property of components within the forms to which they are attached. Code in the library can also make calls to the database stored procedures [PA09]. These files potentially change the database and applications. We used Oracle Forms to convert them to FMT files, which maintain all PL/SQL scripts compiled in PLLs.

File granularity is not sufficient to fully understand the actual effects introduced by a specific change. Tests of a software system are usually at a function, or even at a statement level. Our approach aims at extracting detailed information from changed files to obtain four types of changes that we then discuss in turn:

- Type 1. Direct changes to database objects (§ 4.3)
- Type 2. Direct changes to library functions (§ 4.4)
- Type 3. Indirect changes from database objects to database objects (§ 4.3)
- Type 4. Indirect changes from database objects to library functions (§ 4.5)

4.3 Database Changes

We need to identify which changes are capable of affecting database objects. Major changes to Oracle databases (for example), come from SQL and PL/SQL scripts.

SQL scripts are script files which contain DDL (Data Definition Language) and DML (Data Manipulation Language) statements. PL/SQL files define database functions and procedures. In our approach, we only consider SQL statements that can cause changes: CREATE, DELETE, UPDATE, DROP on tables, procedures, functions etc, while omitting others like SELECT. [CIA+11]

To do this, we employed a third-party SQL parsing tool named **General SQL Parser** (developed at Gudu Software [Sof15]) to capture the names of those objects. The tool reads SQL text, and tokenizes it with the lexical analyzer Lex [LS06] into a list of tokens. The list of source tokens is then used as input to the Yacc parser [Han00]: see Figure 4.18. Yacc reads source tokens, and based on the syntax of different database's SQL dialects, the tool creates a query parse tree if no syntax errors are detected. The raw query parse tree is then translated into a formal parse tree, where the top level node of the formal parse tree is a SQL statement such as: *TSelectSqlStatement*, *TInsertSqlStatement*, *TDeleteSqlStatement*, *TUpdateSqlStatement*, *TCreateTableSqlStatement*, etc., which in turn includes other parse tree sub-nodes.

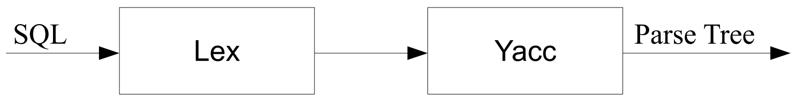


Figure 4.18: General SQL Parser [Sof15]

Furthermore, we had to extend this tool to deal with SQL statements that themselves contained SQL definitions. Other patch files also use SQL for making their changes. In some the SQL is contained in other text, in others it is compiled and has to be decompiled to extract the SQL statements. A suite of tools was developed to handle all the relevant files found in Oracle patches.

We give an example of an SQL script (payauret.sql) below.

payauret.sql

```
SET VERIFY OFF;
WHENEVER SQLERROR EXIT FAILURE ROLLBACK;
WHENEVER OSERROR EXIT FAILURE ROLLBACK;
REM /* $Header: payauret.sql 120.5.12000000.4
2011/01/31 20:39:27 skshin noship $ */
REM dbdrv: none
...
REM Script Name : payauret.sql
REM Script Type : Standard
REM Description : Script to populate the following
entities for Australia
...
DECLARE
...
FUNCTION create_retro_definitions
   (p_short_name in
   pay_retro_definitions.short_name
   TYPE
   ,p_definition_name in
   pay_retro_definitions.definition_name
   TYPE)
```

```
  RETURN NUMBER  IS
    l_retro_definition_id
    pay_retro_definitions.retro_definition_id
    TYPE;
    ...
  BEGIN
  ...
      IF l_retro_definition_id IS NULL THEN
           OPEN csr_get_defn_id;
           FETCH csr_get_defn_id INTO
           l_retro_definition_id;
           CLOSE csr_get_defn_id;
           INSERT INTO pay_retro_definitions
           (retro_definition_id, short_name,
           definition_name, legislation_code)
           VALUES
           (l_retro_definition_id
        ,p_short_name
        ,p_definition_name
        ,g_legislation_code);
      END IF;
   END create_retro_definitions;
FUNCTION create_retro_components
...
```

```
END;
/
COMMIT;
EXIT;
```

The tool goes through each line of `payauret.sql`, tokenizes it and passes it to Yacc. While processing and tokenizing statements, it takes each complete segment of statements as one single statement. In other words, the definition of function *create_retro_definitions* (line 14 - line 42) will be seen as one single statement. Thus, the parse tree of this script file cannot detect any statements in the body of function *create_retro_definitions*. In reality there can be multiple changes made in the body of a function or any other objects (procedures, packages etc). Hence, a sophisticated and revised add-on was developed to recursively read SQL scripts, see Figure 4.19.

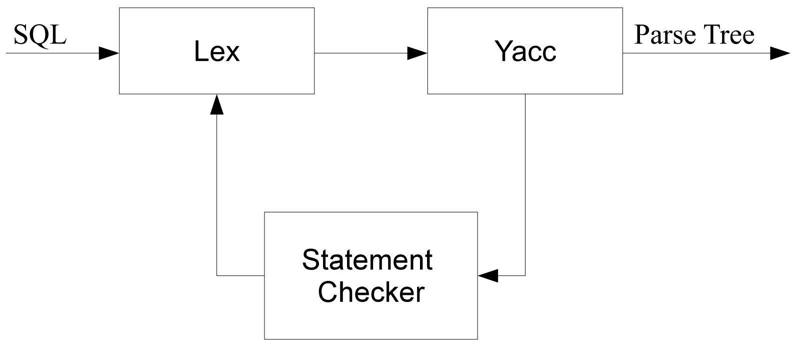


Figure 4.19: Statement Checker

A new module was added to the third-party SQL parsing tool: *Statement Checker*.

Statement checker examines the statements returned by Yacc. It then recursively checks for definitions of functions/procedures/packages/etc. (i.e. objects having bodies), until no more statements are found. After analysis, table *pay_retro_definitions* (line 33) was also identified as a change in this script, among 15 changes in total.

We also need to extract indirect changes from database objects to database objects such as tables, triggers, procedures, etc. Thus, the dependencies between them are required. Fortunately, most databases (*e.g.* Oracle, DB2, MySQL) already detect and store such dependencies. Otherwise it would be fairly straightforward to write some SQL procedures to calculate them. Thus we can just follow the dependencies and compute the indirect changes caused by a direct change.

4.4 Library Changes

Patches to Java libraries often come in the form of class files, and techniques are necessary for detecting changes at the method and field levels between the original software and the patch. [CIA+11]

Some tools and techniques do exist to detect differences between two versions of a program. The Aristotle research group from Georgia Tech. in [AOH04] showed an approach for comparing object-oriented programs. Their approach was not applicable to our domain because it compares both versions of the whole program, rather than making individual class to class comparisons. Moreover, their application domain was several orders of magnitude smaller than ours. Meanwhile, there exist some open source tools like `JDiff` [Doa07], `Jar Comparer Tool` [jar], `JarJarDiff` and `ClassClassDiff` [Tes10b] that only give API differences between two *class* or two *jar* files. To achieve the level of detail we require, namely which methods and fields are changed, we decided to write our own modification detecting tool for class and jar files.

What we do is to compare the two versions of a Java *class* file and report the differences. The *class* file format is described in detail in [LY99]. Since the *class* files are in binary format, we first use a tool to convert them to an easy-to-parse XML format. This XML format is a one-to-one representation of the binary bytecode format, thus comparing two versions of the XML is effectively equivalent to comparing the *class* files themselves. The XML format contains XML representations of everything present in the Java *class* file (methods, fields, superclass, interfaces, access flags, etc.). We compare the two versions of an XML file, node by node, to detect differences in methods, fields, access flags, superclass, interfaces, etc., and list them in another XML file for use in the impact analysis phase. Below are the steps depicting the entire process of detecting modifications.

1. Have the original and modified version of a *class* file in transformed XML format ready. Also have an XML document D ready for recording the differences.

2. Compare the superclasses of both versions and record any differences in D.

3. Compare the interfaces and record any differences in D.

4. Compare the access flags of the two classes and record any differences in D.

5. Compare the methods and record any added, changed or deleted methods in D. For changed methods, record the details of changes (return type, access flags or instructions) in D.

6. Compare the fields and record any added, deleted or changed fields in D. For changed fields, record the details of changes (type, access flags) in D.

As an example, Figures 4.20a and 4.20b show an original and modified code sample, respectively. Note that the only difference is the inclusion of the call to method $bar()$ inside method $foo()$. The XML segment in Figure 4.20c contains this modification information.

```
class Test{
    int i;
    public void foo()f
       i++;
    }
    public void bar(){
       System.out.println();
    }
}
```

(a) Original Code

```
class Test{
    int i;
    public void foo()f
       i++;
       bar();
    }
    public void bar(){
       System.out.println();
    }
}
```

(b) Modified Code

```
<changed signature="foo()">
    <methodinfo>
       <instructions/>
      <dependencies>
          <addedcall>Test:bar()</addedcall>
      </dependencies>
    </methodinfo>
</changed>
```

(c) Difference Holding XML segment

Figure 4.20: Detecting Modifications

4.5 Library-Database Linkages[2]

Program code interacts with the database via SQL commands generated as strings, and passed to the database through the SQL-API. Such strings are often dynamically created, so we used a string analysis tool, the **Java String Analyzer** [CMS03] (JSA), which is capable of statically analyzing a section of code and determining *all* the strings which might possibly occur at a given string expression, including *dynamically constructed strings.* Currently the JSA can only work with Java code, but it is architected in such a way that a single layer of it can be replaced to add support for a different language, leaving the majority of the JSA unchanged.

Ideally, all of the Java classes in a program would be passed simultaneously to the JSA, all usage of the SQL would be checked, and a report would be provided showing which strings, and therefore which database object names, are possible for each SQL string. Unfortunately, the process used by the JSA is extremely resource intensive. Using just a small number of classes ($\sim$**50**) will often cause memory usage to explode. On a 32GB RAM machine, all available memory was quickly exhausted in many tests.

A way to segment the classes into small sets which the JSA can handle is thus required. The technique used for this was to identify all of the unique "call-chains" which exist in the program, for which the bottom-level of a chain is any method which makes use of SQL, and the top-level of the chain is any one of the "methods of interest"[3]. These chains can be constructed via analysis of the program dependency graph.

[2]This section is based on joint work with Jay Parlar. [WC11]

[3]Methods of interest are usually top-level methods, those that do not have any callers or simply the APIs used by a customer's application.

Using this technique on a large Java program, “call-chain explosion” was quickly encountered, due to cycles in the dependency graph. Initially we used a modified depth-first search to go through the graph. Traditional depth-first search algorithms are only concerned with finding if one node can be reached from another. Generating a listing for each path is not their goal. We needed a modified version because *all* possible call-chains through the graph were required. This modified version ended up falling victim to cycles, resulting in an infinite number of chains.

To solve this, Tarjan’s Algorithm [Tar72] was employed to identify all the strongly connected components (i.e. cycles). These components were then compacted into a single node, preserving all of the incoming and outgoing edges of all nodes in the strongly connected component. The modified depth-first search is then run, recording all the possible SQL-related call-chains through the graph.

Remember that the JSA receives a set of classes as its input. The dependency graph works on a method and field level, so each node in a given call-chain represents a particular method or field. Thus for each path generated, the Java class for each node must be found, and the union of those classes is stored. This results in a set of sets-of-classes. For each call-chain that contained one of the compacted nodes, all members of the strongly connected component are lumped into the call-chain, and their classes are identified, the same as every other node in the call-chain. At the end of this, we have a set of sets-of-classes, where each of those sets-of-classes can individually be passed to the JSA, run in a reasonable amount of time, and the possible database object names in each can be identified.

We have developed a tool called **jsa** that essentially provides a wrapper for the Java String Analyzer library. Given a list L, where each member of L is a set of classes to analyze, jsa performs string analysis on each of those sets (i.e., on each member of L), identifying the *SQL-related* methods, and determining what SQL strings are possible within those methods. Those strings are then compared against a list of database object names, and we report whenever one of the database object names is found within the possible SQL strings. The database object names will usually be a list of modified database objects from an Oracle patch.

Very generally, jsa performs string analysis on Java classes. We actually make use of this for two separate purposes:

- Method Filtering
- Full String Analysis

4.5.1 Method Filtering

One aspect of this tool that must be kept in mind is its computational complexity. The Java String Analyzer performs some heavy-weight tasks, and as a result often takes a long time to complete. In our initial tests, the size of L was just too large (we made up L by classes that can be found in Oracle E-Business Suite, that is ∼230,000), and it looked like a best-case scenario was a run-time of 30 days for jsa to complete analysis. We needed a way to reduce the size of L to something more manageable.

It is not just the size of L that causes the long runtime. It is also the size of each set s in L. Each set s is analyzed by jsa individually, and the number of Java classes present in a given s has a dramatic effect on the analysis time for that particular set. We eventually came upon a method of potentially reducing the size of L by using jsa for pre-analysis to filter methods.

Instead of using jsa to analyze sets of classes, we instead identify all the individual Java methods containing SQL calls, and run jsa over the classes containing those methods, one class at a time. This typically results in must faster string analysis, as each iteration analyzes only one class each time, rather than a set of classes. The version of Oracle EBS we used to test our tool contains approximately 13,000 methods which call SQL-related functions, so this becomes ∼13,000 separate analyses with jsa.

Given an input file listing these 13,000 methods, jsa will iterate over it and find all the methods for which the possible SQL-strings can be determined strictly by looking at the class (as opposed to looking at a full call-chain containing many classes). The identifications of these methods are then stored to an output file. This output file is then used to reduce the size of the dependency graph.

Why does this work? The entire purpose of the analysis is to determine all the call-chains through the dependency tree that start at a top-caller and terminates at a method containing a SQL-related call. These call-chains are then passed to jsa to determine which SQL-strings are possible. If we can identify the methods whose possible SQL-strings are unaffected by the call-chains it belongs to, then we can eliminate those methods from the call-chain analysis. This should result in a much smaller L, and thus a much faster execution of jsa when it is run in its normal "Full String Analysis" mode.

4.5.2 Full String Analysis

The main mode of jsa is to perform string analysis over sets of classes. Each set of classes represents a call-chain through the Java application code, or more precisely, represents all the classes used in a given call-chain. For each set of classes s, it locates all the SQL-related method calls (execute, prepareStatement, etc.), and tries to determine all the possible strings that can be passed to each method call. Each set of classes is tied to the top-caller that resulted in the set of classes s. For each s, after determining all the possible strings, the strings are compared to a list of names of database objects, and matches are reported.

The end-to-end result of this is that when given a list of database objects that were changed in a given patch, we can identify all of the top-callers in the application code that result in accesses to some of the database objects. This lets an end-user know which tests they should run after a patch, based on top-callers contained in tests. It should be noted that there will be situations in which the possible strings for a given s cannot be determined. There are a variety of reasons for this, but the consequence is that any top-caller which has strings that cannot be identified should be tested. Since we cannot confirm that the top-caller will not access one of the modified database objects, tests will have to be run.

A sample usage for the full string analysis is as below:

```
java jsa/jsa [-j jar] [-d classpath_dir] [-i ignore_file]
[-t dbobjects] input_classes output_directory
```

Chapter 5

Reachability Analysis[1]

Static analysis computes a conservative set of impacts, and dynamic analysis focuses on executing the program to obtain a more precise set of impacts. Considering the characteristics of enterprise systems, we showed how to combine these two techniques in Chapter 3. In spite of the success in this combined approach, the case studies (discussed in Chapter 8) suggested that there still might be a good number of false-positives present in the estimated impact set. That analysis found out only a tiny portion of the system (0.26% of all top functions/APIs) were affected at run-time. Even though those top functions were executed over 150 thousand times, one could not conclude that the rest of the static impacts were safe to discard. Consequently, testers may still need to rerun many of the regression tests.

Therefore we need further analysis to remove more false-positives. While seeking further analysis to remove false-positives, we realized that *Reachability Analysis* can be used to determine whether, within a graph representation G, a node s can reach

[1]This chapter is mostly based on the author's work: Wen Chen, Alan Wassyng, and Tom Maibaum. "Impact Analysis via Reachability and Alias Analysis." The 7th International Conference on the Practice of Enterprise Modelling. Manchester, United Kingdom. 2014 [CWM14b]

another particular node t, *i.e.*, whether the path $s \rightsquigarrow t$ is feasible, and so this seemed a promising tool in our search for reducing false-positives in the impact set. Reachability analysis has been extensively researched, but to the best of the author's knowledge it has not been applied to change impact analysis.

Reachability analysis has two major categories: ordinary reachability problems that do not require preprocessing (*e.g.* depth-first searching, transitive closure, cycles detection) and those that do require preprocessing (*e.g.* Floyd-Warshall algorithm, Thorup's algorithm, Context-free language reachability). For the former, no usage of complex data structures is needed to compute the reachability of the desired pair of nodes, and the computation can usually be accomplished in linear time. While for the latter, a more sophisticated analysis and data structures are needed since presumably the graph is more complicated and/or a more comprehensive reachability determination is required for the desired pairs of nodes. Such analysis includes reachability under certain conditions: weighted edges, planar graphs, maximum/minimum flows.

In this chapter, we investigate:

- a reverse search algorithm to search around the access dependency graph that was built in the static analysis (§ 3.1), to obtain a set of affected entities for each individual change (both direct and indirect). This task falls into the first category of reachability analysis, the one that does not require complex data structures and/or extra graph information to perform the computation.

- a specialized reachability analysis that can be used to remove false-positives from the computed impact set. In particular, we will explore CFL-reachability analysis – how it can be applied to our problem, how it is actually implemented by a third-party tool, and how it can be of use in our impact analysis approach.

5.1 Reverse Search

Computation in the standard reachability problem does not require preprocessing of the graph. We address **Depth-first Search (DFS)** as an example here, and discuss how it can be used in impact analysis, and in particular, how we use DFS to search the access dependency graph we built (§ 3.1) in the static analysis to obtain a set of affected entities.

Based on the dependency graph of the whole enterprise system and all identified changes, we construct a reverse call graph for each changed library function or database object to obtain a set of **"functions of interest"**. A category of the potentially affected functions are chosen in advance as "functions of interest", typically two types of functions are of our interest: those that appear in test suites, as they are the key to identifying which tests need to be run to check the affected functions; and those that appear on top of other library functions (APIs) in the absence of a customer's applications or test suites. Without a customer's applications or test suites, the topmost level of functions that our analysis can reach are the APIs of the enterprise system. Note that the direction of searching the dependency graph is backward. It traces from each changed function/object back to all potentially affected functions/objects in the graph. This backward search is based on an observation: an entity can only be affected if any other entity it can reach (calling relationship or variable accessing) is changed.

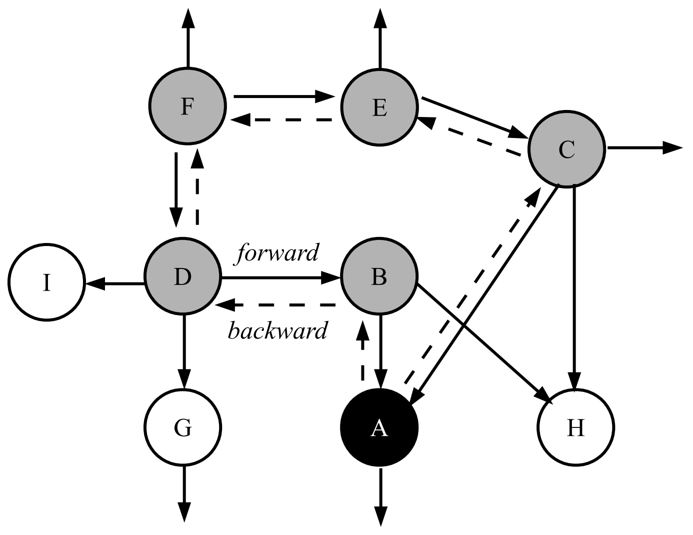


Figure 5.21: A Reverse Search Example

In Figure 5.21, we give an illustration of how the search works: there are 9 functions from A to I in the graph with many other nodes not showing because the figure would become unnecessarily cluttered. Function A (black) is a changed function detected by patch analysis (Chapter 4); functions B, C, D, E, F (grey) are functions affected by A found by a reverse search of the dependency graph; functions H, I, G (blank) are functions not affected. Lines with arrows represent forward dependencies between functions, and dashed lines represent reverse dependencies. Through this reverse search we found there are two backward paths, starting from function A:

- *Path 1*: $A \rightarrow B \rightarrow D \rightarrow F$
- *Path 2*: $A \rightarrow C \rightarrow E \rightarrow F$

By reverse search of the dependency graph, we obtain a set of static impacts S, and in this case we have

$$S = \{A, B, C, D, E, F\} \tag{5.1}$$

The changed function A was extracted by *Change Analysis* (Chapter 4), and, actually it can be any one of the four types (see § 4.2). Subsequent to determining S by reverse search, S is further refined by dynamic analysis to determine the actual run-time event traces that start from each node in S. Thus the reverse search algorithm is used in between change analysis and dynamic analysis. The reverse search algorithm is described using pseudocode below.

Reverse Search (Algorithm 2) takes three inputs: $depFunc$, $depObj$, and C. $depFunc$ are the dependencies between functions that were extracted from the access dependency graph; while $depObj$ are the dependencies between database objects that were extracted from a table that maintains dependencies among database objects in the database; C is the set of atomic (direct) changes and associates indirect changes that were obtained from change analysis. The algorithm first merges (Line 4) $depFunc$ and $depObj$ by string analysis (§ 4.5), which examines the application functions and establishes linkages to database objects they can possibly access. Hence a complete dependency set *dependency* is obtained. Then for each individual change $c \in C$, we apply a *modified* version of depth-first search to collect a set of entities $tmpCallers$ it can reach, and for those entities of $tmpCallers$ that are not already visited by other changes, we add them to the impact set $impactedCallers$. Ordinary depth-first search is only concerned with finding if one node can be reached from another, thus we modified it to collect nodes that were visited along the path. Note that some implementation details are omitted in this segment of pseudocode. We present a more detailed and complete approach in Chapter 7.

Algorithm 2 Reverse Search Algorithm

```
function ReverseSearch($depFunc$, $depObj$, $C$)
$impactedCallers$;
$tmpCallers$;
$dependency \leftarrow Merge(depFunc, depObj)$;
for $c \in C$ do
    if $c \in dependency$ then
        $tmpCallers \leftarrow DFS(c)$;
        for $tmp \in tmpCallers$ do
            if $tmp \notin impactedCallers$ then
                $impactedCallers \leftarrow impactedCallers + tmp$;
            end if
        end for
    end if
end for
return $impactedCallers$;
end function
```

5.2 CFL-Reachability Problem

Ordinary (flat) graph reachability analysis does not take into account some of the detailed behaviour we could obtain from a call graph or dependency graph, if we had the foresight to decipher and collect information that is contained in the graph, *i.e.*,, standard graph reachability analysis only knows whether a node is reachable from another, theoretically. However, often in practice many of the paths can be infeasible, due to *mis-matched* calls and returns that were not identified in either the static or dynamic analysis. The static/dynamic approach that we have adopted works at the granularity of functions, and control flows or data flows were not considered. Thus a finer reachability analysis can be used to determine which of the paths are infeasible and hence should be cut off. Among the more specialized reachability analyses, preprocessing of the graph is required – using a more complex data structure, collecting extra information such as weights or flows associated with the edges. Also, the determination of whether a node can reach another node is not a straightforward process since it is dependent on other conditions, and the traditional shortest-path search algorithm is not sufficient to do that.

Reps [Rep98] argued that one can express a program-analysis problem as a graph-reachability problem with a number of benefits, such as more efficient algorithms can be used, offering of insight into the "$O(n^3)$" problem in program analysis etc. Later a **Context-Free-Language Reachability Problem** was introduced:

Definition: Let L be a context-free language over alphabet $\sum$, and let G be a graph whose edges are labeled with members of $\sum$. Each path in G defines a word over $\sum$, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is a member of L. Four varieties of CFL-reachability problems are defined as follows:

1. the *all-pairs L-path* problem is to determine all pairs of nodes n_1 and n_2 such that there exists an *L-path* in G from n_1 to n_2;

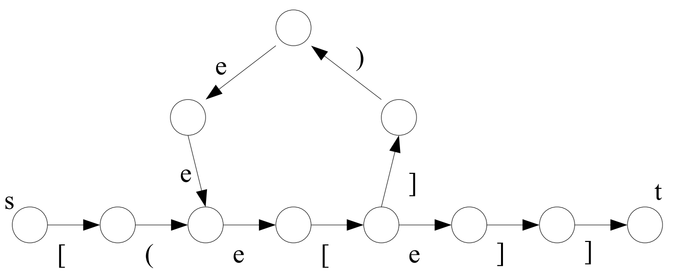
2. the *single-source L-path* problem is to determine all nodes n_2 such that there exists an *L-path* in G from a given source node n_1 to n_2;

3. the *single-target L-path* problem is to determine all nodes n_1 such that there

exists an *L-path* in G from n_1 to a given target node n_2;

4. the *single-source/single-target L-path* problem is to determine whether there exists an *L-path* in G from a given source node n_1 to to a given target node n_2.

Consider the graph shown below, and let L be the language that consists of strings of *matched parentheses* and *square brackets*, with zero or more $e's$ interspersed:



$$
\begin{aligned}
L: \textit{matched} &\rightarrow \textit{matched}\ \ \textit{matched} \\
&\mid\ (\ \textit{matched}\ ) \\
&\mid\ [\ \textit{matched}\ ] \\
&\mid\ e \\
&\mid\ \varepsilon
\end{aligned}
$$

Figure 5.22: A Graph with Labeled Symbols [Rep98]

In this example (Figure 5.22), there is exactly one *L-path* from s to t and the path generates the word "[(e [])eee [e]]". Note that a context-free grammar is defined by the graph.

Then an ordinary graph reachability problem can be transformed to the CFL-reachability problem by labelling each edge with the symbol e and letting L be the regular language e^*.

The reason that we introduce CFL-reachability analysis here is that it can help us answer the seemingly *undecidable* question: **"whether or not a given path in a program representation corresponds to a possible execution path",** as we want to identify infeasible paths in the impact set such that they can be removed to achieve higher precision.

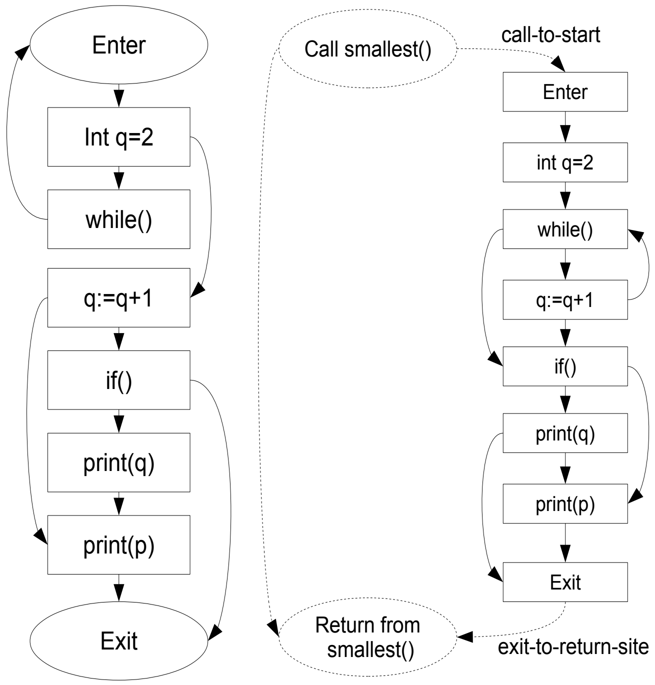
We can transform this problem to a CFL *recognition* problem: "Given a string ω and a context-free language L, is $\omega \in L$?" Then the next obvious question is, what is a feasible execution path? In other words, we need to define a context-free language L to represent feasible paths. It appears to the author that the only paths that can possibly be feasible execution paths are those in which "returns" are matched with corresponding "calls". These paths are called *realizable* paths.

Reps defined a **Supergraph** G^* to deal with *realizable* paths. A supergraph consists of a collection of control-flow graphs – one for each procedure – one of which represents the program's main procedure. Each flowgraph has a unique *start* node and a unique *exit* node. The other nodes of the flowgraph represent *statements* and *predicates* of the program in the usual way, except that each procedure call in the program is represented in G^* by two nodes, a *call node* and a *return-site* node. In addition to the ordinary intraprocedural edges that connect the nodes of the individual control-flow graphs, for each procedure call – represented, say, by call node c and return-site node r – G^* contains three edges: an *intraprocedural call-to-return-site* edge from c to r; an interprocedural *call-to-start* edge from c to the start node of the called procedure; an interprocedural *exit-to-return-site* edge from the exit node of the called procedure to r.

Suppose we have a simple program to find the smallest prime factor of a positive integer number.

```
Smallest(int p, int q){
\*precondition p>1*\
int q=2;
while(p mod q>0 && q<sqrt p)
    q=q+1;
if(p mod q=0)
then
    print(q, is factor)
else
    print(p, is prime);
}
```

(a) Program Smallest



(b) Control-flow Graph and Supergraph

Figure 5.23: Program smallest and its graphs. Dashed nodes and arrows correspond to extra nodes and edges while expanding from G to G^*.

In Figure 5.23b, the graph on the left is the regular control-flow graph of program *smallest*, and the one on the right is the extended supergraph. In the supergraph, the three dashed lines are newly added to the original control-flow graph: *call-to-return-site*, *call-to-start*, and *exit-to-return-site*, with corresponding two new nodes: *call smallest* and *return from smallest*. The benefit of breaking a procedure call into two nodes (call node, return-site node) is that we can trace the flow of executions and examine if there occurs a miss-matched call, by labelling the edges with particular symbols.

In detail, we let each call node in G^* be given a unique index from 1 to *CallSites*, where *CallSites* is the total number of call sites in the program. For each call site c_i , we label the call-to-start edge and the exit-to-return-site edge with the symbols "$(_i$" and "$)_i$", respectively. We also label all other edges of G^* with the symbol e. A path in G^* is a matched path *iff* the path's word is in the language $L(matched)$ of balanced-parenthesis strings (interspersed with strings of zero or more $e's$) generated by the following context-free grammar:

$$
\begin{array}{lll}
\textit{matched} \rightarrow & \textit{matched} \ \ \textit{matched} & \\
 & |\ (_i \ \textit{matched} \ )_i & \textit{for} \ \ 1\text{<=}i\text{<=}\textit{CallSites} \\
 & |\ e & \\
 & |\ \varepsilon & \\
 & & \\
\textit{realizable} \rightarrow & \textit{matched} \ \ \textit{realizable} & \\
 & |\ (_i \ \textit{realizable} & \textit{for} \ \ 1\text{<=}i\text{<=}\textit{CallSites} \\
 & |\ \varepsilon &
\end{array}
$$

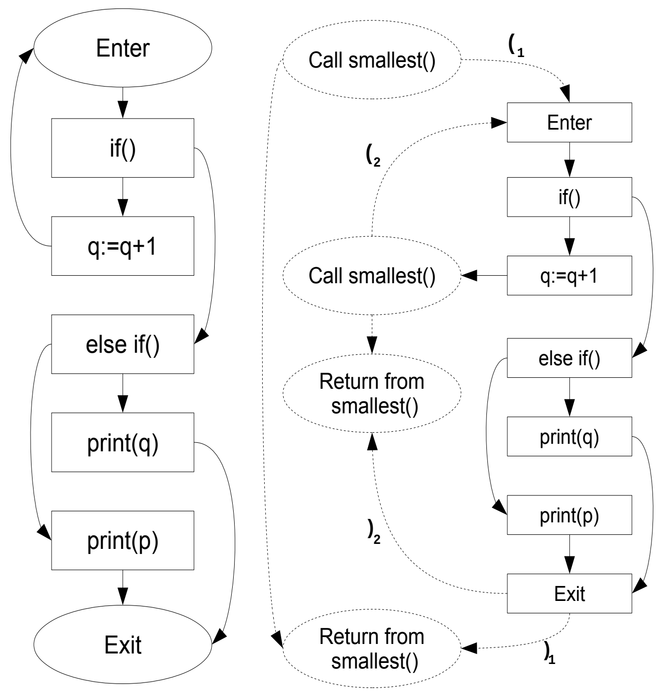
Figure 5.24: Context-free Grammar. A path is a *realizable* path iff the path's word is in the language $L(realizable)$.

The language $L(realizable)$ is a language of **partially** balanced parentheses: Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse need not hold.

To illustrate how we can use context-free language in recognizing paths that are feasible, we rewrite a recursive version of the program *smallest* in Figure 5.25b.

```
Smallest(int p, int q){
\*precondition p>1 && q=2*\
if (p mod q > 0 && q < sqrt p)
then
    q := q+1;
    smallest(p, q);
else if (p mod q == 0)
then
    print(q, is factor)
else
    print(p, is prime);
}
```

(a) Program Smallest Recursive



(b) Control-flow Graph and Supergraph

Figure 5.25: Program smallest (recursive version) and its graphs. Dashed nodes and arrows correspond to extra nodes and edges while expanding from G to G^*.

Instead of a *while* loop, a recursive call to the program itself is made when checking the if condition $\{(p\, mod\, q) > 0\ \&\&\ q < (sqrt\, p)\}$. To preserve the same computation, one extra precondition is required: q should be initialized with integer value 2 and each recursive call increments q until the closest integer to the smallest prime factor (if any) is found. An associated regular control-flow graph and supergraph are also present in Figure 5.25b. In this supergraph, both dashed lines and dashed nodes are newly added. The interprocedure calls are the dashed lines marked with symbols "$(_i$" or "$)_i$", where $i = 1$ or $i = 2$. Hence from the supergraph, we can identify paths for example:

- "$Call\ S \rightarrow Enter \rightarrow if() \rightarrow elseif() \rightarrow print(q) \rightarrow Exit \rightarrow Return\ from\ S$", which has word "$(_1eeee)_1$" is a feasible path since the call-to-start edge "$(_1$" is matched by a correct exit-to-return-site edge "$)_1$";
- however, for path "$Call\ S \rightarrow Enter \rightarrow if() \rightarrow elseif() \rightarrow print(q) \rightarrow Exit \rightarrow Return\ from\ S(inside)$" that has word "$(_1eeee)_2$", in which the program exits to the **inside** "Return from S", the path is infeasible – "$(_1$" is matched by "$)_2$".

5.3 Interprocedural Analysis

Flow analysis of a program is usually done either by *control-flow* or on top of that with analysis of flows of data: *data-flow* analysis. A control-flow graph [Cho11] is a representation of a program that makes certain analyses (including data-flow analyses) easier, in which each node represents a statement, and edges represent the control flow. The statements within a program can be grouped into **basic blocks**, such that within the blocks, instructions will not branch out. A simple example is shown in Figure 5.26.

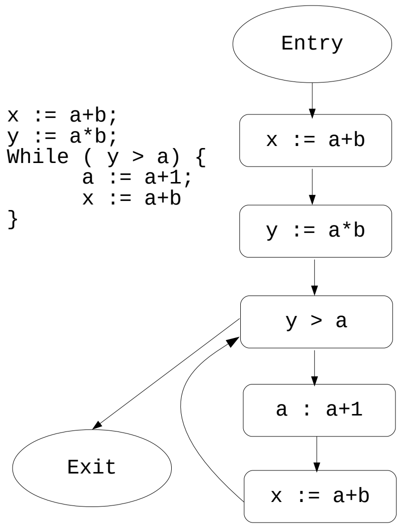


Figure 5.26: A Simple Control-flow Graph Example

In a control-flow graph, all nodes that do not have a normal predecessor should be pointed to by the entry node, and all nodes with a successor should point to the exit node directly or indirectly. Data-flow analysis takes into account the data that flows in the program, based on the control-flow graph. It is a framework for proving **facts** about programs being analyzed, and reasoning and examining interactions between the facts. Facts in data-flow analysis are the determination of whether or not an *expression* e is available at program point p. An expression is regarded as **available** if

- e is computed on every path to p, and
- the value of e has not changed since the last time e was computed on the paths to p.

For instance, in Figure 5.26, we want to compute facts like "$a + b$ *is available*", "$a * b$ *is available*" and then these facts can be used to optimize the code. A typical optimization is if an expression is available, then we do not need to recompute it at some program point. Data-flow analysis can be used to solve a number of classical problems: *"gen/kill"*, *reaching definitions*, *available expressions*, *live variables* etc. Additionally, using these facts we show how to use them to compute *realizable* paths in § 5.4 and in *alias analysis* in Chapter 6.

So far we have briefly looked at **intraprocedural** analysis – how to analyze a single procedure via control-flow and data-flow analysis. However, for concerns such as calling relationships among procedures, the location where each procedure is called from, dataflow facts transferring from one call site to other call sites, determination of valid/invalid paths, require a more precise analysis. We need an *interprocedural* analysis to look into procedures and examine how dataflow facts are kept among statements, combining the intraprocedural information to achieve a more precise program analysis.

Formally, an interprocedural data-flow analysis is concerned with determining an appropriate dataflow value to associate with each point p in a program to summarize (safely) some aspect of the execution state that holds when control reaches p. To define an instance of a dataflow problem, one needs:

- The supergraph(see § 5.2) for the program.
- A domain V of dataflow values. Each point in the program is to be associated with some member of V.
- A meet operator $\sqcap$, used for combining information obtained along different paths.

- An assignment of dataflow functions (of type $V \rightarrow V$) to the edges of the supergraph.

Reps argued that [Rep98] a large class of interprocedural dataflow-analysis problems can be handled by transforming them into realizable-path reachability problems as we mentioned in § 5.2. The ultimate goal of this is to shift from the "meet-over-*all-paths*" solution to the more precise "meet-over-*all-realizable-paths*" solution. A "meet-over-*all-paths*" (MOP) solution is:

$$MOP_n = \bigsqcap_{q \in Paths(start,n)} pf_q(\perp) \tag{5.2}$$

where $Paths(start, n)$ denotes the set of paths in the control-flow graph from the start node to node n. MOP_n represents a summary of the possible execution states that can arise at n; $\perp \in V$ is a special value that represents the execution state at the beginning of the program; $pf_q(\perp)$ represents the contribution of path q to the summarized state at n.

While a "meet-over-*all-realizable-paths*" (MRP) solution is:

$$MRP_n = \bigsqcap_{q \in RPaths(start_{main},n)} pf_q(\perp) \tag{5.3}$$

where $RPath(start_{main}, n)$ denotes the set of realizable paths ($L(realizable)$, see § 5.2) from the main procedure's start node to node n. By restricting attention to just the realizable paths from $start_{main}$, we thereby exclude some of the infeasible execution paths. In general, therefore, MRP_n characterizes the execution state at n more precisely than does MOP_n.

In order to calculate the contribution of each path, the path function is required.

The path function is a *distributive* function f of type $2^D \rightarrow 2^D$, where D is a finite set of the universe of dataflow facts. Then an "*exploded*" supergraph $G^{\#}$ is built to deal with the facts and distributive functions. In $G^{\#}$, each node $\langle n, d\rangle$ represents dataflow fact $d \in D$ at supergraph node n, and each edge represents a dependence between individual dataflow facts at different supergraph nodes. Function f in $2^D \rightarrow 2^D$ can be represented using a graph with $2D+2$ nodes, named *$f's$ representation relation.* An example of this representation relation graph is given below.

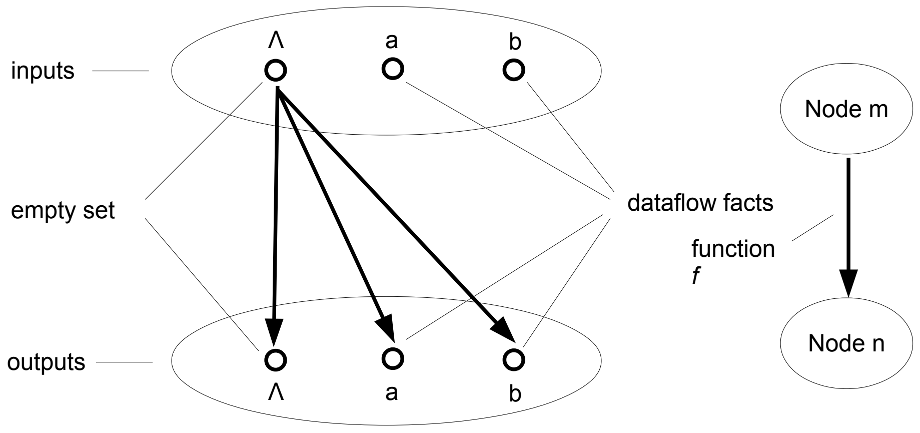


Figure 5.27: A Representation Relation Example [Rep98]

A function's representation relation captures the function's *semantics* in the sense that it can be used the evaluate the function. In Figure 5.27, the universe of dataflow facts D consists of a and b, $D = \{a, b\}$; assume S is the inputs, for $d \in D$, edge $\Lambda \rightarrow d$ means $d \in f(S)$ and in particular $d \in f(\emptyset)$; edge $d_1 \rightarrow d_2$ means $d_2 \notin f(\emptyset)$ and $d_2 \in f(S)$ whenever $d_1 \in f(S)$; $\Lambda \rightarrow \Lambda$ exists in every graph to allow composition of functions. Thus in this case, we have three edges $\Lambda \rightarrow \Lambda$, $\Lambda \rightarrow a$ and $\Lambda \rightarrow b$. Then we can compose different functions (each one is associated with a single edge) in the form: $f_1 \circ f_2 \circ f_3 \circ \cdots$.

Formally, an exploded supergraph $G^{\#}$ is: Each node n in supergraph G^* is "exploded" into $D + 1$ nodes in $G^{\#}$, and each edge $m \to n$ in G^* is "exploded" into the representation relation of the function associated with $m \to n$. In particular:

1. For every node n in G^*, there is a node $\langle n, \Lambda \rangle$ in $G^{\#}$.
2. For every node n in G^*, and every dataflow fact $d \in D$, there is a node $\langle n, d \rangle$ in $G^{\#}$. Given function f associated with edge $m \to n$ of G^*:
3. There is an edge in $G^{\#}$ from node $\langle m, \Lambda \rangle$ to node $\langle n, d \rangle$ for every $d \in f(\emptyset)$.
4. There is an edge in $G^{\#}$ from node $\langle m, d_1 \rangle$ to node $\langle n, d_2 \rangle$ for every d_1, d_2 such that $d_2 \in f(d_1)$ and $d_2 \notin f(\emptyset)$.
5. There is an edge in $G^{\#}$ from node $\langle m, \Lambda \rangle$ to node $\langle n, \Lambda \rangle$.

5.4 Tabulation Algorithm

CFL-reachability problems can be solved via dynamic-programming algorithms. An algorithm of time cubic in the nodes was proposed by Melski [MR97]. It first normalizes a grammar with new nonterminals wherever necessary, and additional edges are added to the graph according to a particular pattern. However CFL-reachability problems can sometimes do asymptotically better than cubic time by taking advantage of the structure of the graph that arises in the analysis. It has been proven that the *Tabulation Algorithm* [RHS95] can solve the problem in time $O(ED^3)$, where E

is the number of edges in the program's supergraph, and D is the size of the universe of dataflow facts.

The **Tabulation Algorithm** is a dynamic programming algorithm that tabulates certain kinds of same-level realizable paths. It uses a set named PathEdge to record the existence of **path edges**, which represent a subset of the same-level realizable paths (*e.g.* $\langle S_p, d_1\rangle \rightarrow \langle n, d_2\rangle$) in the exploded supergraph $G^{\#}$ (see § 5.3). A set named SummaryEdge is used to record the existence of **summary edges**, which represents same-level realizable paths that run from nodes of the form $\langle n, d_1\rangle$, where n is one of the calling sites in the program, to $\langle returnSite(n), d_2\rangle$. The algorithm is a *worklist* algorithm that accumulates the path edges and summary edges.

A full version of the algorithm and illustrations can be found in [RHS95]. It uses the functions described below:

- *returnSite*: maps a call node to its corresponding return-site node;
- *procOf*: maps a node to the name of its enclosing procedure;
- *calledProc*: maps a call node to the name of the called procedure;
- *callers*: maps a procedure name to the set of call nodes that represent calls to that procedure.

5.5 Implementation via WALA

The Tabulation algorithm was implemented via an open source tool **WALA** from IBM [Cen13], which provides static analysis capabilities for Java bytecode and related languages and for JavaScript. The system is licensed under the Eclipse Public License, which has been approved by the OSI (Open Source Initiative) as a fully certified open source license. The initial WALA infrastructure was independently developed as part of the DOMO research project at the IBM T.J. Watson Research Center.

Some of the features of WALA are:

- Java type system and class hierarchy analysis.
- Source language framework supporting Java and JavaScript.
- Interprocedural dataflow analysis (RHS solver).
- Context-sensitive tabulation-based slicer.
- Pointer analysis and call graph construction.

WALA has been extensively used in commercial products: Rational Software Analyzer **NPEs** [LYC+08], Rational **AppScan** [TPF+09] WebSphere as well as in academic products: String analysis [GPT+09], Tabulation solver [RHS95], and locating and fixing software bugs [ZBZ11].

Of particular interest to us is the Tabulation solution. Tabulation algorithms were incorporated in WALA, and follow a class hierarchy shown in Figure 5.28.

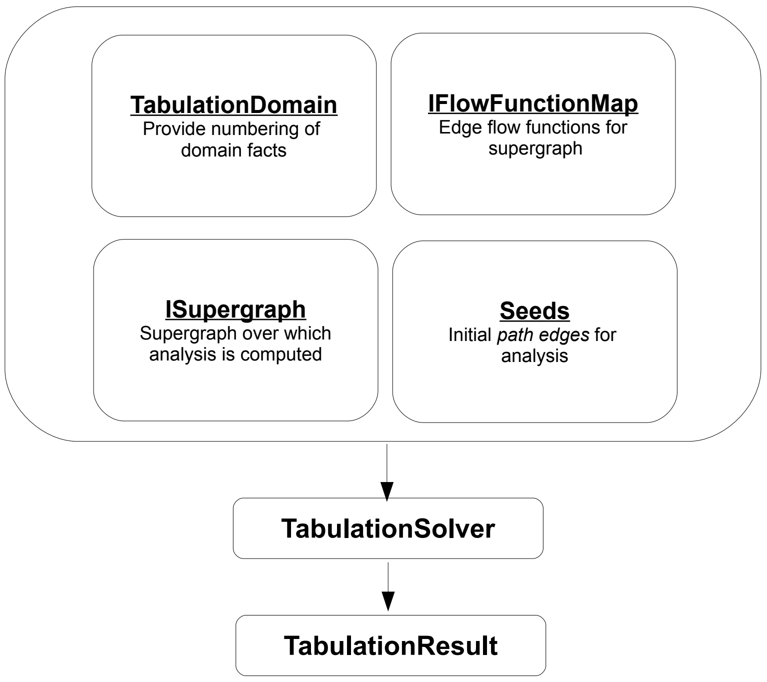

TabulationDomain
Provide numbering of domain facts
IFlowFunctionMap
Edge flow functions for supergraph
ISupergraph
Supergraph over which analysis is computed
Seeds
Initial path edges for analysis
TabulationSolver
TabulationResult

Figure 5.28: Tabulation Overview in WALA [DS10]

- **TabulationDomain** maintains a mapping from facts to integers and controls worklist priorities.
- **IFlowFunctionMap** maintains flow functions on supergraph edges. In particular it maintains functions for each type of edge ($normal, call \rightarrow return, call \rightarrow entry, exit \rightarrow return$), and also missing code.
- **Seeds** initializes path edges for analysis.
- **ISupergraph** build supergraphs over the program to be analyzed.

As an example, **ICFGSupergraph** builds a control-flow supergraph to capture pairs of "caller-callee" relationships shown in Figure 5.29.

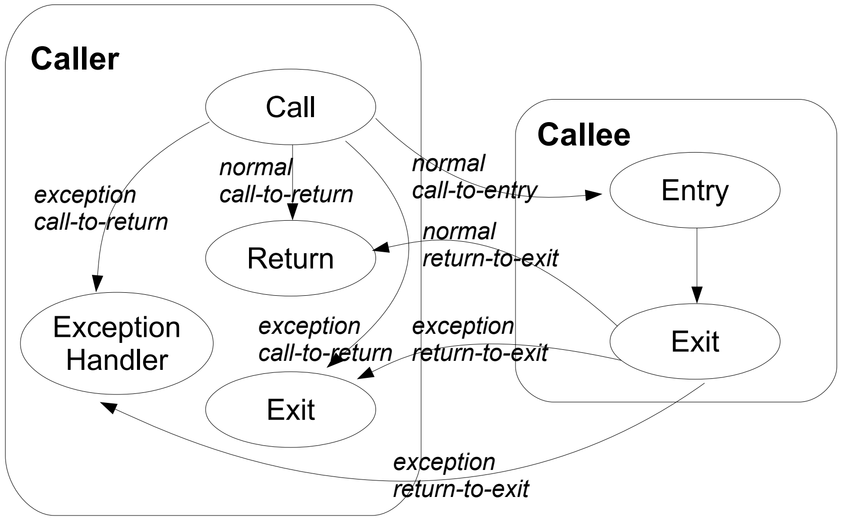


Figure 5.29: A "Caller-Callee" Pair in WALA [DS10]

Edges in this supergraph are labelled by various types. We note that "Exception call-to-return" and "Exception return-to-exit" edges are enhancements introduced in WALA to deal with exceptions, which are not covered in the original Tabulation algorithm. WALA also supports **Partially Balanced** problems i.e., reachability analysis can also be run on supergraphs built from flows that start/end in a non-entrypoint. For instance, in the presence of slicing, one may slice the program into "fairly" independent slices to speed up the analysis or improve efficiency etc. The slice may be started from a non-entrypoint statement that produces an"unbalanced" return (a return without calls). WALA can make "fake entries" for those unbalanced returns with a class named *PartiallyUnbalancedTabulationSolver*.

In the end, TabulationSolver collects all the information about the supergraph, initial path edges, dataflow facts etc., and computes a set of realizable paths and a set of unrealizable paths in TabulationResult. Note that the results computed by the Tabulation algorithm are based on a control-flow graph (and later "exploded" into a supergraph), however, our impact analysis works on an access dependency graph. Thus in the next section we discuss how to convert the tabulation results into the information we want.

5.6 Mapping Control-flow to Dependency

The CFL-reachability analysis determines a set of realizable(feasible) paths – and conversely, it is not difficult to obtain a set of unrealizable (infeasible) paths – by constructing a supergraph based on a control-flow graph of the program, in which paths are labeled with either one of parentheses, e, or ε to determine whether a path is feasible in terms of matched parentheses. Additionally, to achieve a better time complexity and precision, the supergraph is "exploded" to include dataflow facts, such that nodes with dataflow facts are combined to determine the reachability. For instance, a node n with dataflow fact g is considered as reachable from the entry of the program if and only if there is a realizable path $\langle start_{main}, \Lambda\rangle \rightarrow \langle n, g\rangle$, where $\langle start_{main}, \Lambda\rangle$ denotes the program's entry node with empty input Λ.

We observe that by using the Tabulation algorithm (the algorithm to solve the CFL-reachability problem), the computed results are effectively at a granularity of *statements*, since the algorithm is based on the control-flow of the program. However, as far as our approaches to change impact analysis are concerned, (*i.e.,* static, dynamic, change analysis etc.,) our analyses have all been at the granularity of *methods.*

That is, an access dependency graph was built to capture dependencies among *methods* (and fields), with the capability of handling dynamic binding in object-orient languages; dynamic analysis in particular aspect-oriented instrumentation was run on those dependencies to capture run-time event traces of *methods* identified in the static analysis; change analysis collected both direct and indirect changed *methods* (and database objects), and then the set of changes were used as the seeds in reverse searching the dependencies to select functions of interest. Therefore, to make better use of the reachability analysis, a mapping from the control-flow information to dependency information is required. In other words, we need to convert the set of realizable (feasible) paths into a set of pairs of reachable method nodes.

More formally, we covert the set of realizable paths RP to a set of unrealizable paths URP: $\{URP_1, URP_2, URP_3, ...\}$, each unrealizable path URP_i consists of nodes $n \in N^{\#}$ with dataflow facts $d \in D$ and associated edges $e \in E^{\#}$ in between: $\langle start, \Lambda\rangle \rightarrow \langle m, \Lambda\rangle, \langle m, \Lambda\rangle \rightarrow \langle n, d_1\rangle, \langle n, d_2\rangle \rightarrow \langle o, d_3\rangle, ..., \langle p, d_i\rangle \rightarrow \langle exit, d_j\rangle$, where $\{start, m, n, o, p, ..., exit\}$ are nodes in the "exploded" supergraph $G^{\#}$ and $\{\Lambda, d_1, d_2, d_3, ..., d_i, d_j\}$ are associated dataflow facts in each node. Then we examine each unrealizable path URP_i to identify the **enclosing method** f_n of each node $n \in N^{\#}$ along the path and build an unrealizable path of *methods* $URPM$ instead.

Finally, a set of unrealizable paths of methods $URPM$, such that for each $URPM_i$: $\{f_{entry}, f_1, f_2, ..., f_{exit}\}$ with associated dependencies $\{f_{entry} \rightarrow f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_{exit}\}$ is extracted and compared with the static dependencies, so that infeasible paths in the original dependencies can be removed.

This leads to a mapping algorithm that is given in Algorithm 3, where *depFunc* is the static dependencies extracted from the access dependency graph, in the same way as it was in Algorithm 2.

Algorithm 3 Mapping Algorithm

```
1: algorithm Mapping($URP$, $depFunc$)
2: declare UnrealizablePath, Dependencies
3: begin
4: $UnrealisablePath \longleftarrow URP$
5: $Dependencies \longleftarrow depFunc$
6: for $urp \in UnrealisablePath$ do
7:     for $node \in urp$ do
8:         $node \longleftarrow methodOf(node)$
9:     end for
10: end for
11: for $(caller, callee) \in Dependencies$ do
12:     if isNotReachable(caller,callee) is true then
13:         $Dependencies \longleftarrow Dependencies - \{(caller, callee)\}$
14:     end if
15: end for
16: end
17:
18: procedure isNotReachable($caller, callee$)
19: begin
20: for $urp \in UnrealisablePath$ do
21:     if $(caller \in urp)\&\&(callee \in urp)\&\&(callee\ is\ after\ caller)$ then
22:         return true
23:         break
24:     end if
25:     return false
26: end for
27: end
```

Chapter 6

Alias Analysis[1]

Alias analysis, *pointer* analysis, *points-to* analysis, *pointer alias* analysis etc., are often used interchangeably to denote an analysis that attempts to analyze pointers and aliases, such as run-time values of a pointer, aliased pairs of names that point to the same run-time location due to the use of pointers or references. In this thesis, we will be using the term Alias Analysis whenever possible to avoid unnecessary misunderstandings.

Alias analysis has been studied extensively over the past decade. Ways of doing alias analysis include but are not limited to using the control-flow information of a procedure at each program point (flow-sensitive); or computing one solution to the whole program (flow-insensitive); or calling context considerations, *e.g.*,, whether values can flow out of the calling context and return to other callers (context-sensitive and context-insensitive); or using some explicit alias/pointer representations, in which sets of alias relationships are encoded. [Lho02]

[1]This chapter is mostly based on the author's work: Wen Chen, Alan Wassyng, and Tom Maibaum. "Impact Analysis via Reachability and Alias Analysis.". The 7th International Conference on the Practice of Enterprise Modelling. Manchester, United Kingdom. 2014 [CWM14b]

Direct use of the results from alias analysis can allow optimizations to be performed that would otherwise not be possible. Alias analysis is also useful as a preprocessing step for other static analyses, with the payback being that it permits more precise information to be obtained. In many dataflow problems, the dataflow function at a particular point depends on the set of memory locations that a pointer variable may point to; when a better estimate can be provided for this set, a more precise dataflow function can be employed. [Rep98]

This chapter is based on an observation that pairs of aliased variables are essentially pointing to the same memory location, so that changes made to a variable can also change other variables aliased to it. In change impact analysis, we may use this information to reduce the number of false-positives, since functions (if they themselves are not directly changed) would not be affected if no aliased variables are found to be affected by a change.

The following text uses a practical example to discuss the possible types of variables that can be aliased, as well as typical techniques in alias analysis, including which of them can be extended to cover object-oriented features. Finally, we describe an implementation, that takes into account our target system.

6.1 Aliased Variables

Alias analysis computes an approximation of the possible objects that each aliased variable may point to during any execution of a program. Runtime objects are represented with a finite set of **abstract locations**. Abstract locations are necessary because non-terminating programs may allocate an unbounded number of objects (impossible to represent directly), and distinguishing such programs from those that do terminate is undecidable. The granularity with which an alias analysis models dynamic objects using abstract locations is termed its **heap abstraction**. [Sri07]

Typically results of alias analysis are sets of aliased variables, that is, $aliased(x)$, where $y \in aliased(x) \iff y$ is aliased to x. If $l \notin aliased(x)$ for abstract location l and variable x in the program P, then x can never be aliased to variables represented by l in some execution of P.

To better illustrate how aliasing can occur, we present a program `aliasingTest` written in Java below. This program tests three types of variables that can be aliased: **class variable (static field)**, **instance variable** and **local variable**.

aliasingTest.java

```java
package aliasingTest;
public class aliasingTest {
    static int[] staticArray;
    int[] instanceArray;
    public aliasingTest(){
        instanceArray = new int[6];}
    public static void main(String args[]){
        //initializations of arrays
        staticArray = new int[5];
        int[] localArray = new int[3];
        aliasingTest at = new aliasingTest();
        localArray[0] = 1;
        staticArray[0] = 1;
        at.instanceArray[0]= 1;
        //aliasing to arrays
16         int[] aliasOflocalArray = localArray;
17         int[] aliasOfstaticArray = staticArray;
18         int[] aliasOfinstanceArray = at.instanceArray;
19         //run functions that can be invoked within main()
20         alterArrayLocal(aliasOflocalArray);
21         alterArrayStatic(aliasOfstaticArray);
22         at.alterArrayInstance(aliasOfinstanceArray);
23         alterEmpty();
24     }
25     static void alterArrayLocal(int[] array){
26         array[0] = 11;}
27     static void alterArrayStatic(int[] array){
28         array[0] = 11;}
29     void alterArrayInstance(int[] array){
30         array[0]=11;}
31     static void alterEmpty(){
32         System.out.println("No job is doing here.");
33 }
```

All three types of variables (integer arrays in this example) are first initialized (Line[9]-Line[14]) such that their fist element is set to integer 1 . Then aliased variables to each of the three are created (Line[16]-Line[18]). Instead of manipulating the original variables, we run functions on the aliased ones (Line[20]-Line[23]). The rest of this code contains definitions of functions that are called from within the main. Note that each one of *alterArrayLocal*, *alterArrayStatic*, *alterArrayInstance* changes the first element in the appropriate array from integer 1 to 11, while *alterEmpty* does nothing other than print out a single string.

After the invocations of all the first three functions (either static or non-static), the original variables are actually changed (the first element is 11 not 1), even though the functions were only manipulating the aliased copies. This happens because the aliased variables point to the same memory location as do the original variables (Figure 6.30).

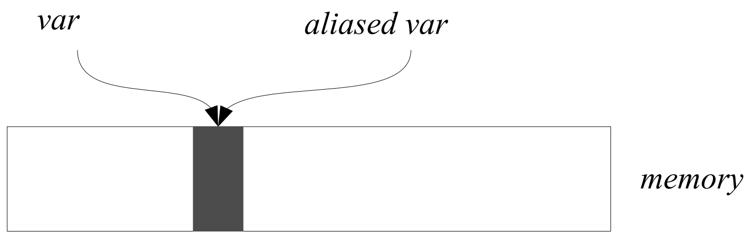


Figure 6.30: Aliased Variables in the Memory

Our observation is that, if along a path in the access dependency graph of a program, one can obtain the aliasing information for each method, dependencies among methods can be identified more precisely. In particular, we follow these steps to achieve more precise dependencies:

1. A *flow-insensitive and context-insensitive alias analysis* computes a single and valid solution to the whole program (details in § 6.2 and § 6.3).

2. We examine the pairs of aliased variables (static field, instance field, and local

variable) throughout the program and obtain a mapping from each method to all aliased variables it can access.

3. We examine paths in the access dependency graph, to determine whether there exists any variable pair $(var, aliased(var))$, in which var is accessed by a method and any one in $aliased(var)$ is accessed prior to this method and along the path.

4. If and only if there exists such pair(s) of aliased variables, we consider the path to be a useful path that should be taken into account in the impact analysis. The reason is, when a change to the current examining method is present, for the sake of safety we assume all variables this method is accessing are changed. Hence methods invoked prior to it can only be affected if accessible variables are aliased in between.

Therefore, in program `aliasingTest`, a dependency edge between function *main* and function *alterEmpty* should be removed, since there is not an aliased variable within *alterEmpty* that was used in *main*. In this way we are able to remove (Figure 6.31) false-positives from the access dependency graph (§ 3.1).

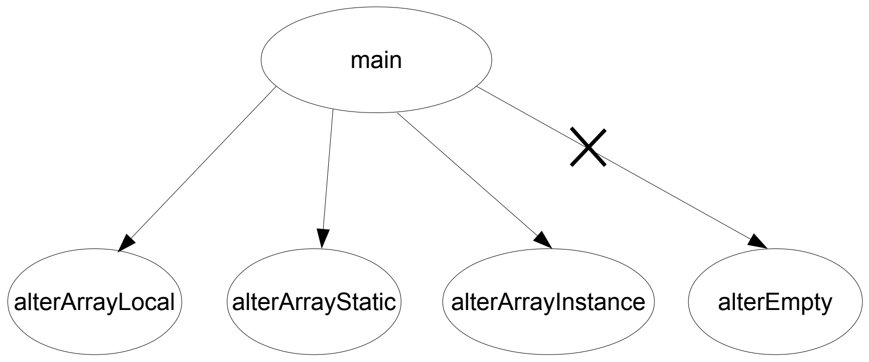


Figure 6.31: Eliminate Dependencies via Alias Analysis

6.2 Flow-Sensitivity

We mentioned earlier that the alias analysis we chose to conduct is *flow-insensitive* alias analysis. A flow-insensitive alias analysis is one that ignores the actual structure of the program's control-flow graph, and assumes statements can be executed in any order and any number of times. It computes a single solution for the entire program, rather than computing a solution for each program point (flow-sensitive) by the control-flow information. [And94]

The difference between the two can be described by comparing the conclusions (described in comments) reached in the segment of code in **C** shown below:

```
int main(void)
{
 int x, y, *p;
 p = &x;
 /*p --> {x,y}*/
 foo(p);
 p = &y;
 /*p --> {x,y}*/
}
```

(a) Flow-Insensitive

```
int main(void)
{
 int x, y, *p;
 p = &x;
 /*p --> {x}*/
 foo(p);
 p = &y;
 /*p --> {y}*/
}
```

(b) Flow-Sensitive

Figure 6.32: Flow-Insensitive and Flow-Sensitive Analysis

A flow-insensitive analysis concludes that "*p*" may point to both the addresses of "*x*" and "*y*", while a flow-sensitive analysis computes alias information at each program point. This reflects a trade-off between accuracy and efficiency: a flow-sensitive analysis is more precise, but uses more space and is slower.

Most alias analyses are flow-insensitive due to a *scalability* concern. Even though there are potential benefits of flow sensitivity, such as computing results per program point, a higher precision due to the sensitivity of paths, we are focusing on flow-insensitive at this time since enterprise systems are typically very large. We leave the application of flow-sensitive alias analysis to future research.

6.3 Context-Sensitivity

Alias analysis can vary in how precisely it models the semantics of method calls and returns [Sri07]. A *context-insensitive* analysis does not precisely model the target address of return statements, instead treating calls as *goto* instructions. A return from a call to some method f is conservatively treated as if it could branch to the point after any call to f, not just the actual caller.

A context-sensitive analysis does not have this imprecision as it treats a program as if all method calls were *inlined*, thereby gaining precision by computing results separately for each invocation of a method call. Context-sensitive alias analysis is critical to the usefulness of recent tools like the static race detector of Naik *et al.* [NAW06] and the type-state verifier of Fink *et al.* [FYD+08]. Unfortunately, existing context-sensitive alias analyses do not *scale* well: although the aforementioned tools are effective on medium-sized programs, alias analysis remains a scalability bottleneck for handling larger programs. The exhaustive inlining approach to context sensitivity can cause a worst-case exponential blowup in the size of the program – as there are a worst-case exponential number of paths in a program's call graph. It has been shown in practice that exhaustive inlining may create up to 10^{23} copies of a method for large Java programs [WL04], and no existing context-sensitive approaches can achieve a better worst-case time complexity than the exhaustive inlining technique.

Hence, we decided to focus on context-insensitive alias analysis at this time. Reps [Rep98] has introduced a context-insensitive approach based on CFL-reachability analysis: a reformulated Anderson's flow-insensitive points-to(alias) analysis([SH97], [And94]) can be expressed as a CFL-reachability problem(see § 5.2) to handle assignment statements in one of the following forms:

`p=&q; p=q; p=*q; *p=q;`

The algorithm [SH97] builds up a graph that represents aliasing relationships among the program's variables. The graph consists of nodes: variables in the program; and edges: an edge from node p to node q means "p might point to q". Additionally, for different kinds of assignments, the algorithm follows some rules(*e.g.* Horn-clause rules) to add additional supportive edges in completing the graph. The base facts for each of the four statement kinds are in Table 6.7.

Statement	**Fact generated**
$p = \&q$	assignAddr(p,q)
$p = q$	assign(p,q)
$p = *q$	assignStart(p,q)
$*p = q$	starAssign(p,q)

Table 6.7: Base Facts for Assignments

The process of reformulating this problem as a CFL-reachability problem consists of three steps:

1. To show that alias analysis can be expressed as a *Datalog* program. This includes generating base facts for each assignment statement in the program, and those additional edges to the graph in Horwitz's algorithm [SH97] can be expressed as Horn clauses.

2. To show that alias analysis can be expressed as a *chain* program. This includes introducing a new relation in the points-to graph – "$\overline{pointsTo}$", which is the reversal of the "$pointsTo$" relation. In other words, both directions in the graph are maintained.

3. To extract a context-free grammar based on the base facts and chain rules such that reachability analysis can be conducted on the alias graph to determine if one node can alias another.

The revised set of base facts for each type of assignments is shown in Table 6.8.

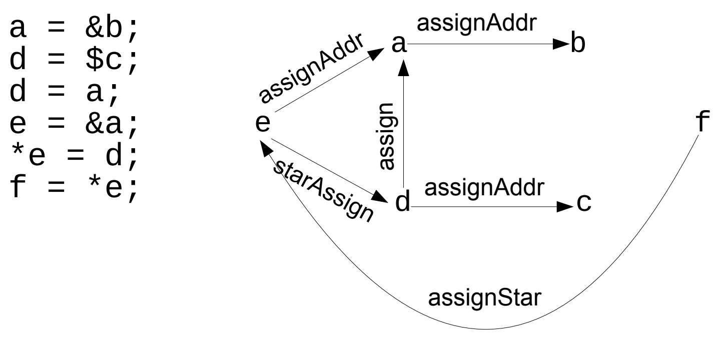
Statement	**Fact generated**
$p = \&q$	assignAddr(p,q), $\overline{assignAddr(q,p)}$
$p = q$	assign(p,q), $\overline{assign(q,p)}$
$p = *q$	assignStart(p,q), $\overline{assignStar(q,p)}$
$*p = q$	starAssign(p,q), $\overline{starAssign(q,p)}$

Table 6.8: Revised Base Facts for Assignments

A complete context-free grammar is extracted below.

$$pointsTo \rightarrow assignAddr$$
$$pointsTo \rightarrow assign\ pointsTo$$
$$pointsTo \rightarrow assignStar\ pointsTo\ pointsTo$$
$$pointsTo \rightarrow \overline{pointsTo}\ starAssign\ pointsTo$$
$$\overline{pointsTo} \rightarrow \overline{assignAddr}$$
$$\overline{pointsTo} \rightarrow \overline{pointsTo}\ \overline{assign}$$
$$\overline{pointsTo} \rightarrow \overline{pointsTo}\ \overline{pointsTo}\ \overline{assignStar}$$
$$\overline{pointsTo} \rightarrow \overline{pointsTo}\ \overline{starAssign}\ pointsTo$$

Aliasing information can now be determined by solving L(points-to)-reachability problems in the graph (Figure 6.33) which was labeled with symbols in Table 6.8. For instance, we can identify that variable f is aliased to both variable b and variable c.



```
a = &b;
d = $c;
d = a;
e = &a;
*e = d;
f = *e;
```

Figure 6.33: Assignments and Associated Point-To Graph with Labels

6.4 Adaptation to Object-Oriented Language

The CFL-reachability-based alias analysis in § 6.3 is designed and suitable for the C language. However, nowadays more and more software, especially enterprise systems, is written or at least partially written in object-oriented programming languages that encourage pervasive use of heap-allocated data.

To reason about pointer behaviour in large-scale enterprise systems we need to find a "sweet spot" between precision and scalability (both time and space concerns). The approach in § 6.3 is based on Anderson's algorithm [And94] which does not inspect pointers' behaviours from the perspective of object-oriented programming, *i.e.,*, Anderson's algorithm does not handle object-oriented features.

For example, Andersen's analysis implicitly assumes that all code in the program is executable. However, a Java program may contain a large portion of unused libraries. Including everything in the analysis leads to high costs and low precision. Also it does not deal with *virtual calls*. The actual type of a target method in a virtual call can be different from where the the call was invoked. Hence semantics of virtual calls have to be modelled precisely from accurate analysis of the program. Additionally, assignments in imperative languages are very different from field accesses in Java (read/write to instance field/class field), and object creation (via **new** statement) etc., which should be modelled in an appropriate way.

Sridharan [Sri07] introduced both a context-insensitive and a context-sensitive alias analysis that are based on the CFL-reachability problem. Those approaches are designed for object-oriented languages, in particular, Java. Thus semantics of assignments, field access, method calls are all modelled to build a points-to graph representation (see Table 6.9).

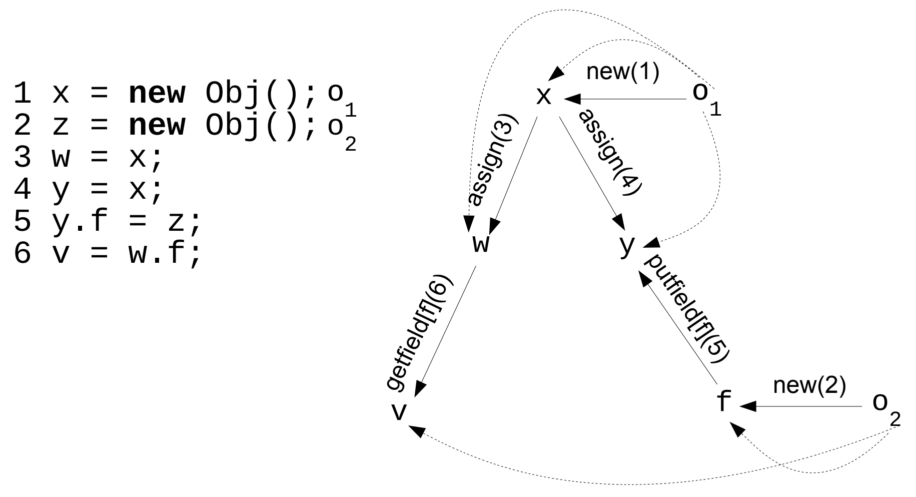
Statement	**Graph Edge(s)**
$x = new\ T()$	$o_s \xrightarrow{new} x$
$x = y$	$y \xrightarrow{assignglobal} x$ if x or y is a global; $y \xrightarrow{assign} x$ otherwise
$x = y.f$	$y \xrightarrow{getfield[f]} x$
$x.f = y$	$y \xrightarrow{putfield[f]} x$
$x = m(a_1, a_2, ..., a_k)$	$a_i \xrightarrow{param[s]} f_{m,i}$ for $i \in [1..k]$; $ret_m \xrightarrow{return[s]} x$

Table 6.9: Base Facts in Alias Analysis for Java

Edges in the graph represent four canonical assignment forms: allocation statement (**new**), copy statement (*assign* or *assignglobal*), heap read ($getfield[f]$)[2], and heap write ($putfield[f]$). Note that there is one extra entry in the table that represents *method calls* – for each method m, the graph has nodes $f_{m,i}$ for $m's$ *formal*

[2] f is the name of the field.

parameters and a special ret_m node for $m's$ return statements. At call site s of m, $param[s]$ edges are added from each *actual* parameter to the corresponding *formal* parameter and a $return[s]$ edge from the ret_m node to the appropriate caller's variable. Then, for a sequence of simple assignments, an alias graph(see Figure 6.34) can be built according to these base facts.



```
1 x = new Obj(); o1
2 z = new Obj(); o2
3 w = x;
4 y = x;
5 y.f = z;
6 v = w.f;
```

Figure 6.34: Java Points-To Graph Example

In Figure 6.34, o_1 and o_2 are the two objects newly created (Line 1 to Line 2). Line numbers are used to identify corresponding edges in the graph, and dashed edges are used to represent the "$flowsTo - path$". An object can flow from an allocation site to a variable only through a **new** edge followed by a sequence of $assign$ statements. Hence in this case, object o_1 can flow to variable w by following a $flowTo - path$ of "$o_1 \rightarrow x \rightarrow w$". In other words, $o_1 \in aliased(w)$ where $aliased(w)$ maintains a set of variables that are aliased to w.

In the same way that a context-free grammar was extracted in § 6.3, a complete context-free grammar can be extracted for solving the Java alias problem as shown below.

$$\boxed{\begin{array}{l}
flowsTo \rightarrow \mathbf{new} \\
\overline{flowsTo} \rightarrow \overline{\mathbf{new}} \\
flowsTo \rightarrow flowsTo\ ciAssign \\
\overline{flowsTo} \rightarrow \overline{ciAssign}\ \overline{flowsTo} \\
flowsTo \rightarrow\ flowsTo\ putfield[f]\ alias\ getfield[f] \\
\overline{flowsTo} \rightarrow\ \overline{getfield}[f]\ alias\ \overline{putfield}[f]\ \overline{flowsTo} \\
alias \rightarrow \overline{flowsTo}\ flowsTo \\
ciAssign \rightarrow assign \mid assignglobal \mid param[i] \mid return[i] \\
\overline{ciAssign} \rightarrow \overline{assign} \mid \overline{assignglobal} \mid \overline{param}[i] \mid \overline{return}[i] \\
pointsTo \rightarrow \overline{flowsTo}
\end{array}}$$

In this context-free grammar for Java, "$\overline{flowsTo}$" production reverses the "$flowsTo$" production (same as "$\overline{pointsTo}$" to "$pointsTo$" in § 6.3 for language C) to define the *alias* language. Now suppose we have two $flowsTo$ paths: $o \rightarrow x$ and $o \rightarrow y$ that lead to $o \in aliased(x)$ and $o \in aliased(y)$. If we want to determine whether x can flow to y, then the reverse of path $o \rightarrow x$ has to be introduced to concatenate $o \rightarrow x$ and $o \rightarrow y$. If, by concatenation there exists such a path in which $x\ flowsTo\ y$, we conclude that x and y are aliased.

Again, aliasing information can now be determined by solving L (points-to)-reachability problems in the graph (Figure 6.34), which was labeled with symbols in Table 6.9, with the assistance of this context-free grammar.

6.5 Implementation via WALA

WALA (see § 5.5) also implements alias analysis for Java, and we can make use of this in our change impact analysis. WALA provides a framework for 1) flow-insensitive Andersen-style [And94] pointer analysis, and 2) demand-driven(context sensitive) pointer analysis. [SB06a]

We begin first with the flow-insensitive Anderson-style alias analysis. As we argued in § 6.2 and § 6.3, our target systems are typically very large, and often contain hundreds of thousands of classes. Thus scalability remains one of the biggest concerns in our change impact analysis. This framework provides options to select exist context sensitivity policies, or even define your own policy. This differs from other alias analysis in two ways [IBM13]: **Heap Model** and **Context Selector**.

A *HeapModel* defines the rules to collect *abstract pointers* and *heap locations*. An *abstract pointer* (represented by *PointerKey* class) represents a runtime object such as local variables, static/instance fields from a particular allocation site, or other variants. Essentially it is the name for an equivalence class of pointers from the concrete program, that are collected into a single representative in the abstraction. Since non-terminating programs may allocate an unbounded number of objects, this abstraction is an important contributor to the scalability of the technique. A *heap location* (represented by *InstanceKey* class) represents all objects of a particular type, or all objects from a particular allocation site, or all objects from a particular allocation site in a particular context, or other variants – the granularity with which an alias analysis models dynamic objects using abstract pointers. A *HeapModel* provides call-backs for the pointer analysis to create *PointerKeys* and *InstanceKeys* during analysis. One can customize the policy by providing one's own *HeapModel*.

The context of method calls can also be changed. The *ContextSelector* object controls the policy by which the call graph builds contexts. The simplest context is `Everywhere.EVERYWHERE`, which represents a single, global context for a method. Other context-policies can represent call-string contexts, contexts naming the receiver object to implement *object-sensitivity* [LH06], or other variants. One can also customize a context-sensitivity policy by providing a custom ContextSelector object.

WALA also has some built-in polices in alias analysis with an increasing order of precision and costs both in time and space:

- ZeroCFA: a simple, cheap, context-insensitive pointer analysis using a global context for each method, which introduces a single InstanceKey for every concrete type – all objects of a particular type are represented by a single abstract object.

- ZeroOneCFA: a policy that provides an approximation of "standard" Andersen pointer analysis, using allocation sites to name abstract objects. By default, the HeapModel introduces a single InstanceKey for every allocation site. This is the policy that we employed in the change impact analysis. The reason that we chose this policy is, our target system is typically very large, object-sensitivity is too expensive to be added, while treating each method as it is in a global context leads to imprecision.

- ZeroOneContainerCFA: a relative expensive policy that extends the ZeroOneCFA policy with unlimited object-sensitivity for collection objects. For any allocation sites in collection objects, the allocation site is named by a tuple of allocation sites extending to the outermost enclosing collection allocation.

The aliasing result obtained from WALA is a mapping $m:\ var \rightarrow aliased(var)$ from variables to corresponding sets of aliased variables, *i.e.*,, for each variable var_i in program P, we obtain a set of aliased variables $aliased(var_i)$. In an alias analysis of policy ZeroOneCFA, each instanceKey is associated with an individual allocation site that contains allocations of multiple variables. After constructing a call graph of P, the alias analysis determines: 1) what allocation sites can be reached from var_i, and 2) what variables in the allocation sites identified above are aliased to var_i. Then we can follow the steps described in § 6.1 to further remove paths that are feasible (reachability analysis § 5.6) but not affected by changes.

Chapter 7

A Complete Hybrid Approach

In this thesis we have described techniques that were developed to perform *impact analysis* as applied to *enterprise systems.* In this chapter, we summarize these techniques,[1] that, used together, provide a complete solution (§ 7.1) to achieving a *safe* and sufficiently *precise* impact analysis on large-scale enterprise systems. The result of the analysis is a set of affected functions and fields (impact set) in the system associated with a given set of atomic changes.

Additionally, we discuss how our impact analysis can be used in *Regression Test Selection* (RTS) and *Focussed Testing* (§ 7.2), and how to augment a users' test suite to include tests that are critical but not covered by the original test suite.

7.1 The Approach at a Glance

The approach consists of multiple steps:

[1]The order in which we introduced those techniques is not necessarily the same as the order constituting the overall solution in this chapter.

1. *System analysis* (briefly described in § 4.1). System analysis analyzes the structure of an enterprise system P. A typical enterprise system consists of three layers: customer's application, application library and database.

2. *Static analysis* (§ 3.1). Static analysis builds an *Access Dependency Graph* G to abstract a static graphic representation of the system. Both calling dependencies and field dependencies are taken into account, and it is also able to handle object-oriented features like inheritance and dynamic binding. The static analysis is as conservative as a vanilla approach, but preserves higher precision. The graph G is stored in a database table (`depenFunc`) consisting of two columns: `Caller` and `Callee`.

3. *Reverse search* (§ 5.1). A reverse search algorithm was designed to search the dependency graph G in reverse order, such that "functions of interest" can be found. Note that this DFS-based reverse search may be run throughout the entire process, whenever it is necessary.

4. *Change analysis* (Chapter 4). Change analysis simply identifies the atomic changes that were made to the system, which then motivates the change impact analysis that is the subject of this thesis. Types of changes that need to be extracted from a given set of atomic changes AC are: i) direct changes to database objects; ii) direct changes to library functions; iii) indirect changes from database objects to database objects; iv) indirect changes from database objects to library functions. This is accomplished by a number of techniques such as PL/SQL script extraction and decompilation, building a SQL parser, differencing Java source codes, and string analysis. The set of atomic changes

AC is then extended to C.

5. *Compute the static impact set* (§ 3.3). A static impact set S is computed by reverse searching the dependency graph G. Starting from each individual change $c \in C$, we identify entities within G that could reach it. Hence at the end of this step, S contains all the static impacts, with presumably a good number of false-positives. This static impact set is essential, since it preserves safety and completeness, and serves as the input set for further analyses that focus on removing false-positives to achieve better precision.

6. *Dynamic analysis* (§ 3.2, § 3.3). An aspect-based dynamic instrumentation and data collection is conducted to collect event traces for functions $f \in S$ identified in the static impact set. "Aspects" are created to insert "advice" at each "join point" of interest in the system. A "pointcut" picks out multiple "join points" and returns traversing information defined in the "advice". The traversing information obtained in this process is: names of invoked functions, where they come from, and where they will go .

7. *Compute the dynamic impact set* (§ 3.3). A dynamic impact set D is computed to gather the traversing information for each f. Essentially D is a subset of S, that is, functions and associated dependencies proved to exist in an actual execution. In the meantime, a set of potential over-estimated impacts PO can be obtained by subtraction: $PO = S - D$.

8. *CFL-reachability analysis* (§ 5.2–§ 5.6). A context-free reachability analysis is conducted on the set of potential over-estimated impacts PO to remove false-positives. The analysis first builds a control-flow graph and on top of that a

supergraph (adding interprocedural flows) of the system, and labels each edge in the graph with parenthetic symbols, e and ε. Then a context-free grammar is extracted to find feasible paths with matched calls and returns, and conversely infeasible paths with mis-matched calls and returns. To improve the precision and time complexity, a more sophisticated refinement is incorporated. This "explodes" the supergraph to include data-flow facts at each statement. Finally, by combining the data-flow facts and control-flow information, we determine if one node can reach another. Note that we have thus developed a way to transform control-flow reachability to dependency reachability, in order to match the granularity of our previous analyses. In the end, we obtain a set of unreachable nodes for each f in PO, and hence reduce PO to PO'.

9. *Alias analysis* (Chapter 6). A flow-insensitive and context-insensitive alias analysis is conducted to find aliased variables in the system. Even after removing infeasible paths by reachability analysis, there still might be paths that are feasible, but that cannot be affected by the changes. This is mainly determined by careful examination of aliased variables, since functions that are not able to access aliased variables of a changed function cannot be affected if they, themselves, are not directly changed. Thus we conduct a Java-adapted alias analysis based on CFL-reachability problems. In the same way that we extract a context-free grammar to filter out infeasible paths, an alias grammar is extracted to handle object-oriented features and decide the reachability of nodes in an alias graph. In this way, aliasing relationships among variables in the system can be determined. Note that we map aliasing information of variables to those of functions by figuring out the "enclosing" function for each aliased

variable, *i.e.,* we obtain a set of aliasing dependencies A among functions. In the end, the already-reduced potential over-estimated impacts PO' is further reduced to PO''.

10. *Gathering results.* A final impact set I on a set of atomic changes AC to the system is calculated by:

$$I = D \cup PO'' \tag{7.4}$$

A graphical view of the process of this complete solution is drawn in Figure 7.35.

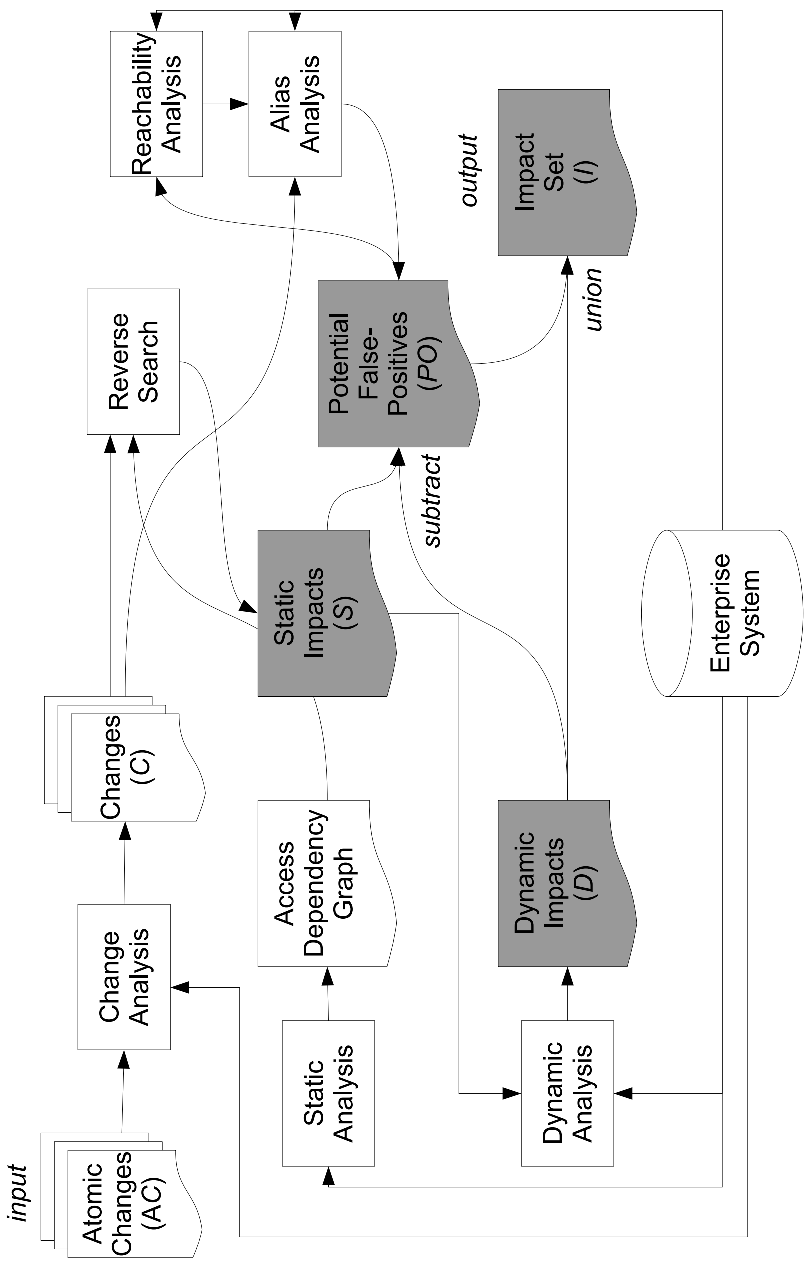


Figure 7.35: System Flow of the Complete Approach

7.2 Benefits of the Approach

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely effected by the modifications. One of the conventional approaches in regression testing is to rerun every test in the suite – this is typically very expensive, especially when the number of tests is large and even for a single test, the execution time can be excessive. Hence, a number of Regression Test Selection techniques have been proposed to reduce the cost both for procedural languages [Bal98, CRa94, LW91, RH97, FF97, LW92] and for object-oriented languages [HLK$^+$97, KGH$^+$94a, KGH$^+$94b, RHD00, AK97].

A *safe* regression test selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program [RH96]. Several safe regression test selection techniques (e.g., [Bal98, CRa94, RH97, FF97, RHD00]) exist. These techniques use some representation of the original and modified versions of the software to select a subset of the test suite to use in regression testing. Empirical evaluation of these techniques indicates that the algorithms can be very effective in reducing the size of the test suite while still maintaining safety [BRR01, GHK$^+$01, KPR00, RH98, RHD00, FF97].

In case of object oriented languages, a number of regression test selection techniques have been developed [RHD00, AK97, HJL$^+$01]. Rothermel, Harrold and Dedhia's algorithm [RHD00] was developed for only a subset of C++, and has not been applied to software written in Java. White and Abdullah's approach [AK97] also does not handle certain object-oriented features, such as exception handling. Their approach assumes that information about the classes that have undergone specification or code changes is available. Using this information, and the relationships between the changed classes and other classes, their approach identifies all other classes that may be affected by the changes, and it is these classes that need to be retested. White and Abdullah's approach selects test cases at the class level and, therefore, can select more test cases than necessary.

Harrold *et al.* [HJL$^+$01] presents the first safe regression test selection technique for Java that efficiently handles the features of the Java language. The technique is an adaptation of Rothermel and Harrold's graph-traversal algorithm [RH97, RHD00], which uses a control-flow-based representation of the original and modified versions of the software to select the test cases to be rerun. They use the notion of *coverage matrix* and *dangerous entity.* Assuming P and P' to be the actual and modified version of a program, respectively, the coverage matrix records which entities of P are executed by each test case in a test suite T. A dangerous entity is a program entity e such that for each input i causing P to cover e, $P(i)$ and $P'(i)$ may behave differently due to differences between P and P'. Rothermel and Harrold describe a regression test selection technique that uses a control flow graph (CFG) to represent each procedure in P and P' and uses edges in the CFGs as potential dangerous entities [RH97]. Dangerous entities are selected by traversing in parallel the CFGs for P and the CFGs for P', and whenever the targets of like-labeled CFG edges in P and P' differ, the edge in P' is added to the set of dangerous entities. After dangerous edges have been identified, the system uses the dangerous entities and the coverage matrix to select the test cases in T to add to T'.

Orso *et al.,* [OSH04] presented a new regression test selection algorithm for Java programs that handles the object-oriented features of the language. The algorithm consists of two phases: partitioning and selection. The partitioning phase builds a high-level graph representation of programs P and P' and performs a quick analysis of the graphs. The goal of the analysis is to identify, based on information on changed classes and interfaces, the parts of P and P' to be further analyzed. The selection phase of the algorithm builds a more detailed graph representation of the identified parts of P and P', analyzes the graphs to identify differences between the programs, and selects for rerun, test cases in T that traverse the changes. Their base idea is effectively the same as in [HJL$^+$01], but due to the two phases, they claim and show with some empirical studies that this technique scales up to larger software systems. However, the largest domain they applied their technique to has approximately 2,400 classes which is still way below the number of classes in our domain – almost 230,000. And also, due to the same reasons mentioned in the previous paragraph, [OSH04] is not quite suitable for our problem domain.

Both approaches compute redundant information and/or require extra memory. Harrold's [HJL$^+$01] approach requires a changed version $P^{'}$ of the original program P, which implies the changes have to be actually made. Throughout the entire hybrid approach presented in § 7.1, we do not need to apply any changes – it enables the users' ability to estimate impacts before taking any actions, and this is crucial, since rolling back the system to a previous state can be expensive and sometimes even impossible. The purpose of traversing two control-flow graphs simultaneously or identifying changes between P and $P^{'}$ via partitioning [OSH04] is to identify impacts, in the sense that changed system behaviours can be captured. However, this impact information is captured more precisely in our approach. Through reachability analysis and alias analysis, our approach is able to filter out false-positives, such as infeasible paths and paths that cannot be affected by the directly changed entities. Our approach also builds a control-flow graph of the program, but additionally, data-flow facts and aliased variables are added in the analysis such that it preserves a more precise set of "dangerous entities" as compared with Harrold's approach.

Figure 7.36 shows an overview of the process involved in applying our impact analysis for regression test selection. Impact analysis on the three-layer enterprise system (see § 4.1) computes a set of impacted entities I via the approach presented in § 7.1. For each $i \in I$, we examine if i is covered by any test in the original test suite T. We add i to the reduced "Focused Test Suite" if and only if it is covered. Note that we are assuming here that users maintain a test coverage matrix that documents relationships among tests and entities those tests can cover.

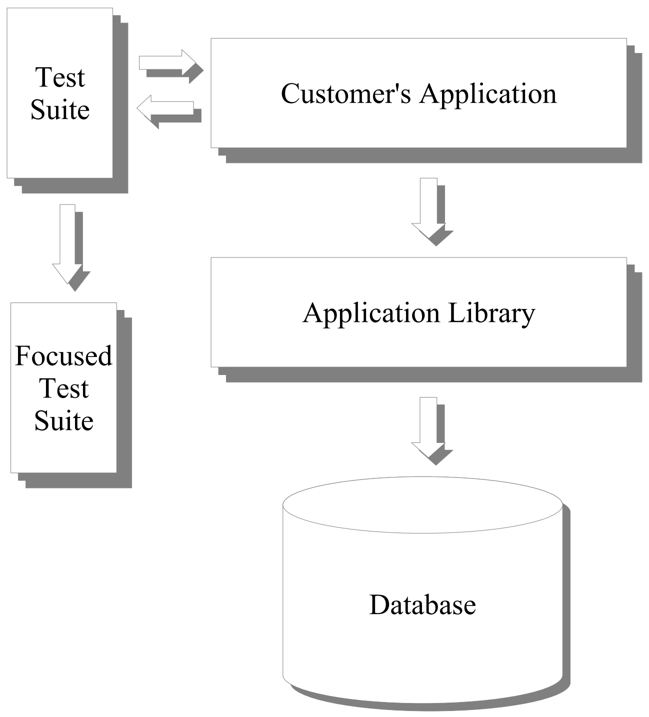


Figure 7.36: Impact Analysis in Regression Test Selection

Notice that the impact set I we obtained from impact analysis contains not only the entities that are covered in the users' original test suite, but also the ones that are not covered. Obviously, our impact analysis is seeking a safe and precise estimated set of impacts, and it does not take into account the users' test suites. Hence newly discovered entities can be additionally added to augment the original test suite.

Chapter 8

Empirical Study

The foregoing techniques in this thesis constitute a hybrid impact analysis approach (see § 7.1) for large-scale enterprise systems. Our research motivations and objectives mentioned in § 1.2 describe the change impact method and tool we aimed to develop. In particular, it must be:

- scalable, in that it is able to deal with the size of typical enterprise systems, that can stretch to hundreds of thousands of classes;
- able to extract both direct and indirect changes made by vendors of the system, or by software developers who develop customized software that use the enterprise system for basic functionality;
- safe, since these systems are usually mission critical for an organization, so mis-identified impacts can lead to huge financial loses.
- precise, since in practice, over-estimated impacts are not only costly, but also may make the analysis misleading by confusing users who need to decide what actions to take to cope with the impacted functions;

- timely, in that it is crucial to have a full understanding – in terms of impacts – of the intended changes before applying any of them, as it can be even more expensive to roll back the system to a previous state if something goes wrong. Unintended changes can be filtered out in advance to avoid altering the system to an incorrect state.

Our goal, then, is to empirically investigate whether our objectives can be met in practice by this proposed hybrid approach. The following sections present the experiment setup and design (including variables and measures), threats to validity, and final results.

8.1 Variables and Measures

There is only one independent variable in this empirical study: the hybrid impact analysis tool. Dependent variables include *precision* and *time overhead.*

In spite of the accuracy of the precision (see Equation 1.1) defined by Maia [MBdFG10], it is not useful to conduct an impact analysis if one has already determined the actual impacts. In a related work [HGF^{+}08], they define the set of actual impacts as "number of modifications that really occurred", and it was derived by extracting logs from CVS (a versioning system). It was described as: "For each change set C, extract from CVS what was really modified, M, due to each change in C. Assume a change c in C. We extract every change that occurred after c and because of c; and include the corresponding class name on M." However, this way of obtaining the set of actual impacts depends on a number of assumptions:

1. it assumes the developers are perfectly accurate in determining all the modifications after an original change;

2. it assumes all changes are correctly identified in the CVS, so that later modifications can be associated reliably with the original change;

3. it ignores things that are affected but not directly changed. Suppose we have a function f, f can be affected by changes made on its callees that return to it. However, the function itself has never been textually changed, thus changes of function f probably can not be identified by simply extracting changes from the CVS logs.

Hence, for the moment, our measure of precision remains the conventional one:

$$Precision = \frac{|I|}{|M|} \tag{8.5}$$

where I is the estimated impacted functions and fields, and M is the total number of functions and fields in the program [OAL+04]. We will discuss in detail on the reasons that we use this metric in § 8.4.

To evaluate the execution cost of the hybrid approach, we measure the time overhead of each major step in the approach (§ 7.1).

8.2 Experiment Setup

The experiment was set up on in the environment shown below in Table 8.10.

Hardware	**Operating System**
Quad core, 3.2 GHz, 32G RAM	Red Hat Enterprise Linux Server release 5.10 (Tikanga) 64 bit

Table 8.10: Environment Setup

As the objects of analysis we used two releases of Oracle E-Business Suite (Table 8.11), and for the source of atomic changes we used multiple patches (Table 8.12) that were obtained either from Oracle E-Business Suite *Patch Wizard* or manually download from Oracle Metalink.

Release		**Facts**		
Application	**Database**	**Classes**	**Entities (functions and fields)**	**LOC (approx.)**
11.5.10.2 (11i)	10.2.0.2.0 (10g)	195,999	3,157,947	8.7 Million
12.0.0 (R12)	11.2.0.1.0 (11g)	226,288	4,604,415	10 Million

Table 8.11: Oracle E-Business Suite Releases and Facts

In Table 8.11, the two releases, $11i$ and $R12$ for short, use Oracle databases $10g$ and $11g$, respectively. To have an intuitive idea of the size of enterprise systems we are experimenting on in this study, we calculated the total number of classes and entities in the system. Note that the number of entities includes both functions and fields within classes. For the moment we assume that we are able to look into each class and count the number of fields , details will be introduced later.

In Table 8.12, we list patches corresponding to the two versions of Oracle E-Business Suite that we want to analyze. Patches can range from a couple of KBs to hundreds of MBs, depending on the purposes of applying them. The smallest patch in this study is a patch (# 10107418) that was developed to fix some CPU bugs – presumably this would not affect the functionality of a system's behaviour, but certainly we cannot reach a conclusion before conducting an impact analysis. Another patch (# 11734698) was applied on the product "Human Resources" provided by Oracle E-Business Suite. In particular, it delivers a consolidated set of files for Australian legislation, and it has to be applied because it is required by the business process: *hrglobal*.

Patches	**Release**	**Facts**	
		Size	**Description**
# 5565583	11.5.10.2	212MB	Fusion Intelligence for E-Business Suite Family Pack 11I.BIS PF.H. MetaLink note, 406004.1 is a complete source of information about this family pack. It includes both the technical and functional details you need in order to apply the patch.
# 10107418		10KB	This patch has fix for bug 9086631 and bug 9190120 (CPU bugs). It contains an unified driver file to be applied with AutoPatch.
# 14321241		99MB	ORACLE Applications With 11i.ATG_PF.H RUP6: CPU ConsolidatedPatch For OCT 2012. It contains an unified driver file to be applied with AutoPatch.
# 11734698	12.0.0	79MB	AUSTRALIA CUMULATIVE RELEASE R12.AU.14 This patch delivers a consolidated set of files for Australian legislation, and is required for the hrglobal process to run successfully.

Table 8.12: Patches for Oracle E-Business Suite

To successfully apply patch # 11734698, the latest legislation data has to be update (Hrglobal installation). It also requires additional code levels to have been successfully applied to the system before it can be applied. These code levels at the time that this patch was built, are: R12.AD.A.delta.6, R12.ATG_PF.A, R12.ATG_PF.A.delta.6, R12.HR_PF.A.delta.8. Additionally, a large number of pre-install and post-install tasks have to be completed to make sure of the successful deployment of the patch. For instance, the following Oracle reports need to be modified to display Foreign Employment in this patch: (A) Self Printed Process - Postscript (B) Validation Report - Text layout. The process of patching itself can be tedious, costly, and more importantly – dangerous, if one does not have a full understanding of which parts of the system will be "touched" by it. Without proper and careful impact analysis in an automated way, it is way beyond any tester's ability to proceed with it safely.

8.3 Experiment Design

We conducted two experiments:

- **Experiment 1 – Overview**. We need to evaluate the scalability of our approach while maintaining safety. Hence in this first experiment, we conducted only static analysis, corresponding to Step (1) to Step (5), and Step (10) in the complete hybrid approach (see § 7.1). Our initial static analysis is safe, recognizing all dependencies among functions and fields in the system, but redundant dependencies caused by dynamic binding are pruned out. Thus, this first experiment simply demonstrates scalability of the time consuming static analysis.

- **Experiment 2 – Overview**. Building on the static analysis, we presented dynamic techniques to make the analysis more precise by cutting off false-positives. Hence in this experiment, we examined if these techniques can realize savings in practice. The experiment covers all the steps in the complete approach in § 7.1.

Note that our other research motivations such as extracting both direct and indirect changes, and identifying impacts before applying changes, are also demonstrated through these experiments.

8.3.1 Experiment 1

We followed Step (1) and Step (2) on Oracle E-Business Suite release 12.0.0 and extracted an access dependency graph. The rules of constructing this graph were defined in § 3.1.3. Major steps [Iqb11] in building the access dependency graph are (see Figure 8.37):

- Using the *classpath* information the File Manager detects which directories (containing class, jar, zip files) need to be processed and also unjars and unzips jar and zip files, respectively, to class files.
- The XML Generator use the ClassReader tool of Dependency Finder [Tes10a] toolset to generate one-to-one equivalent XML files of the class files.
- The Entity Handler uses the XML files to build the entity list, i.e., the list with all the methods and fields, as well as the inheritance information that exists among classes and interfaces.

- The Dependency Handler uses the same XML files to build the access dependency graph. This is a complex step because it has to handle some of the side issues like dynamic binding.
- Due to the size of the system, it can be very costly to maintain full strings of each entity. Hence the entity list, access dependency graph and inheritance information are first processed by *numbering* and then being inserted into database tables. The numbering is essentially a mapping from entities to integers, for the convenience of manipulation and storage.

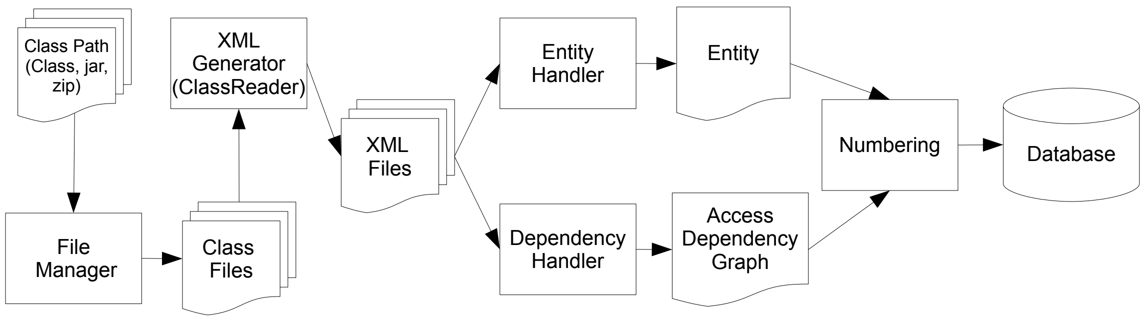


Figure 8.37: Steps in Building the Access Dependency Graph

In Step (4) the change analysis, we looked into patch # 11734698 and extracted potential changes to both application library and database. This is accomplished by first classifying all types of files in the patch, and then decompiling particular types into readable texts. Details of these conversions were presented in Figure 4.17. For each type that may have impact on the system, we developed individual preprocessors written in Python to do the conversion. Patch files that can affect database objects are: SQL, PLS, CTL, DRV, FMB, LCT, PKB, PL, PLL, XDF, XML. For each one of those 11 types, we developed either independent analyzers or tools that use Oracle tools or third-party tools, in particular, Oracle Forms, Reports and General SQL Parser. After converting all necessary types of files in the patch, a program named *scriptAnalyzer* was called to process other types of files and collect information about affected database objects. For instance, PLL scripts are first converted to FMT files by Oracle Forms and then *scriptAnalyzer* takes the FMT files and calls a python written parser *fmt_parser* to extract database relevant manipulations.

In total there are four types of changes that need to be identified from analyzing the patch (details in Chapter 4): (i) changes on the database (ii) changes on the library (iii) changes between database objects and (iv) changes between the library and the database (Java string analysis).

Process of the patch analysis is as below in Figure 8.38. This patch analysis can detect changes of type (i) and *part* of type (ii). Changes made to library functions can be from RTF, JSP or CLASS files (see Table 4.6). By converting RTF to JSP and a JSP parser we could extract changes within the files. However, for CLASS files that are intended to replace corresponding classes in the system, we need to compare the old version (from the system) and the new version (from the patch) to obtain a set of changed fields and functions (§ 4.4). In practice, this is accomplished by XML Generator and a comparator to difference the two XMLs. The general process is described in Figure 8.39.

To achieve safety and completeness, we also include indirect changes (type (iv)) that can occur between library functions and database. This is accomplished by running the Java string analyzer tool **jsa** described in § 4.5. Jsa "wrappers" around the conventional Java String Analyzer and tries to first locate all SQL-related calls in the system and then identify the possible strings (corresponding to database objects) that can be passed to those method calls. Once a database object o is identified to be used by a method f in the library, we add an edge $f \rightarrow o$ in the access dependency graph.

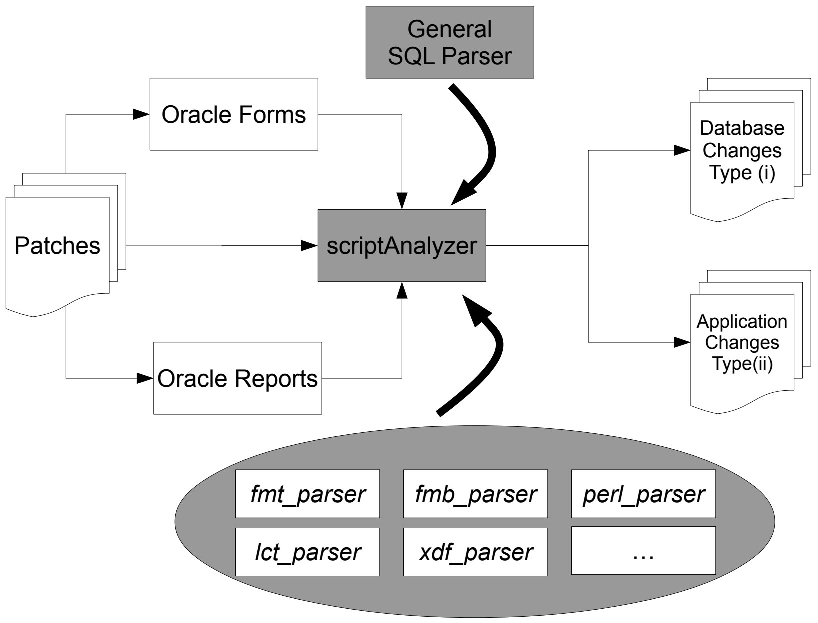


Figure 8.38: Process of Analyzing Patches

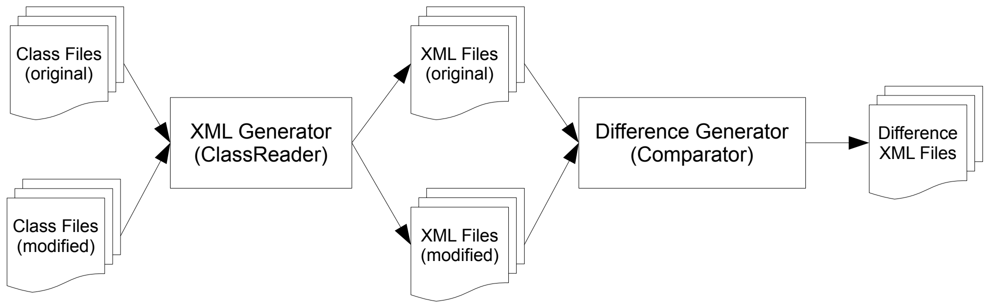


Figure 8.39: Find Modifications in the Library

Later, all the extracted changes are used as inputs to the reverse search algorithm (§ 5.1) in Step (3). Note that before running the search, changes with associated dependencies are inserted in the access dependency graph. The reverse search algorithm searches the access dependency graph and produces a set of static impacts for each input change. Functions or fields that have any dependency on the input change are considered potentially impacted. In addition to the set of all static impacts, we also produced a set of "functions of interest". In the absence of a user's application, we are interested in the top callers (APIs) – functions that provide access to the enterprise system.

To conclude, in this first experiment, we compute a full dependency graph, extract direct and indirect changes, obtain a set of static impacts with "functions of interest", and calculate the running time for each major step.

8.3.2 Experiment 2

In the second experiment, we investigated whether the complete hybrid approach (§ 7.1) met our goal of study, especially for studying how precise our approach can be. The experiment was conducted on Oracle E-Business Suite release 11.5.10.2 and three corresponding patches: patch # 5565583, patch # 10107418, and patch # 14321241. Step (1) to Step (5) in the hybrid approach are exactly the same as in Experiment 1. Hence we skip them and start describing how the second experiment went with respect to the other steps.

From Step (6) to Step (9), we conducted, in order, the dynamic analysis, CFL-reachability analysis and alias analysis. In the dynamic analysis, we built an aspect-oriented instrumentor to instrument the system of release 11*i*, and the instrumentation was executed only on methods that were identified as belonging to the static impact set. The whole point of dynamic analysis in our approach is to investigate which paths in the static impact set are valid in runtime, so instrumenting the entire system is not necessary and would be very expensive.

The aspect-oriented instrumentor was built using AspectJ (§ 3.2.3), which uses *pointcuts* to pick out certain *joint points* in the program flow. After that, *advices* were used to capture dynamic information before, after or at each *pointcut*. We developed a Java program called **Trace** to pick out executions of every method in the program, as long as the control flow is not in the current class, such that all the other methods being called at this particular execution can be captured. We ran this instrumentor on every method in the static impact set and maintained a list of outputs. Each output consists of the event trace for each method being executed. Then we extract dependencies (dynamic impacts) out of the traces and “mark” them as verified in runtime.

Then CFL-reachability analysis was implemented via WALA(§ 5.2) to cut off false-positives. We ran the Tabulation algorithm implemented in WALA on the set of "potential false-positives". The set of "potential false-positives" was obtained by subtracting the dynamic impact set D from the static impact set S. In the experiment, we marked dynamically confirmed dependencies in the graph using a flag to distinguish them from the unconfirmed dependencies. For the unflagged entities and dependencies, we ran the Tabulation algorithm on them and collected a set of infeasible paths. As we have mentioned, the information regarding infeasible paths is at a granularity of statements, however, our dependency graph is for functions. This discrepancy in the experiment means we have on the one hand infeasible paths of *unknown* statements, and on the the other hand dependencies among *known* functions. Hence our mapping program called **Map** (see Algorithm 3) was used to map those statements and paths to functions and associated dependencies.

The alias analysis (Chapter 6) in Step (9) takes the processed "potential false-positives" from Step (8) as input and calculates aliasing information, such that methods that have no chance to access those aliased and changed variables in the system are not considered as affected. Another input is the set of changes after patch analysis. In this experiment, we needed to extract a set of changed variables – obtained by (i) looking into each bottom-level function in the set of "potential false-positives" (these bottom-levels functions are the ones that served as the start points in the reverse reach) (ii) extract a set of variables that those bottom-level functions can access, including class fields, instance fields, and local variables. After that, a WALA implemented alias analysis, in particular, of policy ZeroOneCFA (see § 6.5) was run on those variables that bottom-level functions could access to obtain a set of aliased variables[1]. Finally, we examines which of the other functions in the “potential false-positives” could access those aliased variables. Note that a mapping was also required in the alias analysis, since it also operated on the control-flow, and we simply reused the **Map** program.

The results gathered in this experiment are essentially the remaining and the “marked” dependencies (see Equation 7.4). With those impacts we then calculated the precision of the approach. For the convenience of evaluating individual techniques, the impact set after each technique was also recorded. Additionally, the running time of each major step was recorded.

8.4 Threats to Validity

Like most other empirical studies, our study also has limitations that we should be aware of while interpreting the results.

As mentioned, we did not have access to a user application that used E-Business Suite, so we also did not have a test suite for such an application. Usually, such information is classified because of security and other concerns. Even though we had tried to access a test suite for an application built on Oracle E-Business Suite from one large organization, the tests we obtained were basically a spreadsheet, in which simple instructions for executing the tests were listed. However, to execute these tests, one has to have a full copy of the entire system – both customized applications and the implemented enterprise system, as well as necessary test data that have

[1]For the sake of safety, we assume here, if a function is changed, then potentially all of its accessible variables can be changed.

to be inputed to the tests. Without this information, the empirical study would not be able to compute impacts in terms of the customer's applications. Hence our empirical study currently focused on identifying impacts within Oracle E-Business Suite, though it is not hard to extend our study to cover customized code since the underlying techniques remain the same. One exception of extending our empirical study is, if the customized code is written in a language other than Java, such as Cobol or C. In such a case further analysis of converting Cobol's or C's dependencies to our access dependencies would be required.

Also, while interpreting the results, the the formula used to calculate precision can vary. As we mentioned, the validity of computing the size of actual impacts by extracting from program logs the direct modifications is questionable, and in practice not available most of the time. Thus Maia's definition [MBdFG10] is accurate but not really. We intend to use Orso's definition [OAL$^+$04], which is straightforward but inaccurate, and sometimes even misleading (see Equation 8.5 and Figure 3.13) when the approach is not safe. However, since our approach to impact analysis computes a complete static dependency, Orso's definition is not risky for our empirical study.

It is difficult to conduct comparative analyses between our approach and other existing impact analysis approaches. As we mentioned in Chapter 2, the size of our target system is apparently beyond the capability of other existing tools. In particular: constructing a full static dependency graph on a small sample program may lead to memory leak when using `Soot`; the `CollectEA` approach in collecting dynamic information on our target system hung for days and eventually ran out of memory on a computer of 8 GB RAM(see § 3.2.2). The scalability issue is one of the motivations that drove this research.

8.5 Results and Analysis

8.5.1 Result of Experiment 1

Building the access dependency graph for Oracle E-Business Suite R12 took over 7 hours, which is large but quite manageable, especially as this process is independent of any patch or proposed change, and thus can be prepared in advance. This graph forms a substantial corporate asset for other kinds of analysis, and can be easily and quickly updated as the system changes, provided we do the proper analysis of the changes. The dependency graph has over 4.6 million nodes and over 10 million edges. About 1% of those dependencies are dependencies among database objects, and 1.53% of the entities are database objects.

Reverse searching this dependency graph takes only a few seconds for each starting point method or field.

The patch # 11734698 contained 1,326 files with 35 different file types. Among those 35 file types, 11 types can affect both the library and the database. We ran our tools on each file with one of these 11 types and identified 1,040 directly changed database objects (*e.g.,* 142 from PLL files and 86 from FMB files), and just 3 directly changed Java methods.

The program-database dependency approach described in § 4.5 found that 19,224 out of the 4.6 million methods (0.42%) had SQL-API calls, and that 2,939 of these methods (just over 15%) had a possible dependency on one of the 1,040 affected database objects.

We adopted as our definition of "functions of interest" those which were not themselves called by anything else, "top callers" (APIs) (totally 1.6 million, 34.6%

of the entire system), and there were 33,896 of these that were impacted, a fraction over 2% of all top callers. The most important results are listed in Table 8.13.

Patch	**Direct Changes**	**Affected Functions (% of total functions)**	**Affected Top Functions (% of total top functions)**
# 11734698	1,043	71,506 (1.56%)	33,896 (2%)

Table 8.13: Static Impacts of Patch # 11734698

The patch, as might be expected, only affects a tiny part of the library (1.56%), and around half of the impacts are on the system APIs (47.4%). This is in practice reasonable, since patches are intended to modify how users' code can interact with the system.

Build Dependency Graph	**Extract Changes**	**Reverse Search**	**Compute Impacts**	**Total**
7 Hours	2 Hours	2.7 Hours	1 Hour	∼12 Hours

Table 8.14: Execution Time in Experiment 1

Table 8.14 summarizes the execution time of experiment 1. In total, a complete static impact analysis in this experiment takes approximately 12 hours, that is, an overnight job. Note that building the dependency graph occupies the largest amount of the analysis time in this experiment. Within the 7 hours that it takes to build the dependency graph, XML generation for all the class files (including JARs and ZIPs) takes approximately 40 minutes. The computation in memory, namely generation of the entity list, inheritance information and dependency relation takes just a bit over

1.5 hours. However, the most time-consuming task is to insert all that information into the database, which takes approximately 4.5 hours. Therefore, improving the performance of database insertion in the future, could reduce the total execution time significantly.

8.5.2 Result of Experiment 2

The second experiment was targeted at Oracle E-Business Suite release 11.5.10.2 and its corresponding three patches (see Table 8.12). The process of conducting the static analysis is the same as in experiment 1. In addition to that three other analyses were conducted to reduce false-positives.

The system contains 195,999 classes, and by examining each converted XMLs from class files we found 3,157,947 entities (both functions and fields). The process of building the access dependency graph added over 18.4 million dependencies and took over 9.5 hours to complete.

By patch analysis, we found 16,787 direct database changes, and 25,613 direct library changes for patch #5565583; 610 direct database changes, and 3,374 library changes for patch #14321241; no direct database changes, and no library changes for patch #10107418. Apparently patch #5565583 is the largest patch among the three and hence it is intended to change quite a number of functions in the system. While patch #10107418 is a pretty small one and only intended to fix some CPU issues, just as we expected, it does not contain any functional changes.

The computed static impacts for each patch are listed in Table 8.15.

Patch	Direct Changes	Affected Functions (% of total functions)	Affected Top Functions (% of total top functions)
# 5565583	42,400	699,534 (22.2%)	160,800 (9.6%)
# 14321241	3,984	230,209 (7.3%)	69,971 (4.2%)
# 10107418	0	0	0

Table 8.15: Static Impacts of the Patches

As we can observe from Table 8.15, the static impacts can reach up to 22.2% of functions in the system, even with regard to "top functions", we had almost one out of ten top callers identified as impacted. It is a quite large portion of the system, and if testers were given this set of impacts, a good amount of testing work still has to be conducted. This is mainly caused by two factors: (1) the patch itself is large. As we can see from experiment 1, the patch we were analyzing only contained ~1000 changes, which is just 2.4% of the changes patch #5565583 can produce.(2) the connectivity among components in different releases can vary. Since the release (11i) we used in this experiment was released earlier than the one (R12) in experiment 1, presumably vendors can modify how components communicate with each other to improve usability, hence the release (R12) may have lower coupling than 11i, which may lead to a looser system - less entities can be affected by the same change set. (3) more importantly, this set of static impacts may contain a large amount of false-positives. In other words, we may have included many over-estimated impacts that come from infeasible executions, invalid calling paths, etc.

We ran AspectJ to instrument the identified static impacts, for each “affected function”. In the experiment this was accomplished by examining the production directory, and executing predefined “aspects” for each class when a `main` function was found. We did not make synthetic methods to execute each affected function since in practice functions can only be invoked from a valid program entrance. The predefined “aspect” can be found in `Trace.java` in § 3.2.3. The instrumentation was extremely time consuming, since our target system is very large. Initially this process took over one week, and the program hung several times during this period.

On the one hand, many executions were simply long – may take up to hours for a single run. On the other hand, many of them prompted the user for an UI and asked for inputs (to make selections) to continue. We split the instrumentation into subtasks, each one of them focused on instrumenting just one component in the system, and for the latter problem, we conservatively collected all the calling relations no matter what the user inputs were. By this means, we tried to maximize the usage of CPU and memory, and at the end, the instrumentation was reduced to around 48 hours.

As we argued in § 8.4, user applications or actual test suites on Oracle E-Business Suite can be utilized in instrumenting the system and hence generate more precise and meaningful results. Our approach of instrumentation is safe, since the instrumentation works on the impact set of the conservative static analysis. We are careful not to discard impacts unless we can show they are definitely not impacted.

Patch	Total Function	Total Top Function	Static Impacts	Dynamic Impacts	Potential False-Positives
# 5565583	3,157,947	1,673,132	699,534	4,806	694,728
# 14321241			230,209	1,232	228,977

Table 8.16: Instrumentation Result on Patch # 5565583 and Patch #14321241. (We will walk through the result analysis for Patch # 5565583 only, since they essentially follow the same process, and Patch # 10107418 has no impact on the system.)

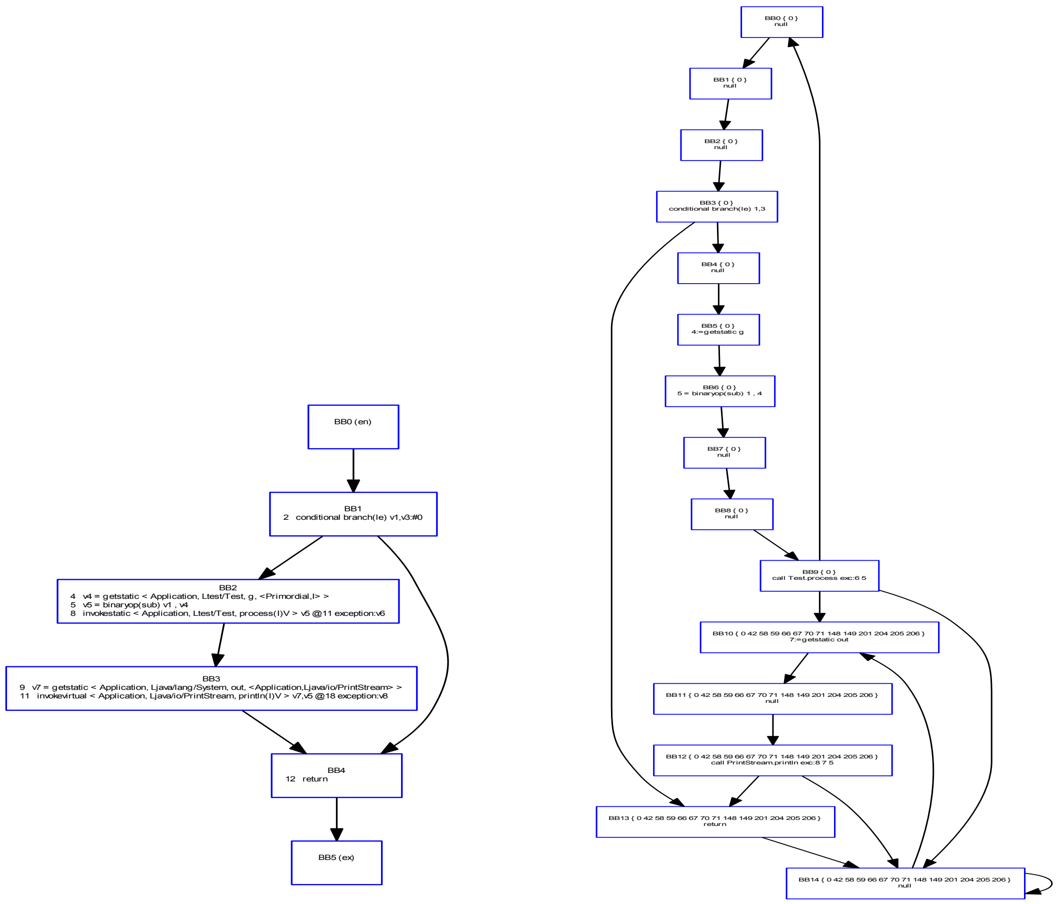
After the instrumentation, we collected the relevant information (Table 8.16) from our dynamic analyses: among the total static dependencies, only 8,357 were covered in the executions, that is, 0.45‰; and 4,806 functions, that is, 0.26% of all top callers (out of 1,673,132) were covered. From these results, it seems the instrumentation only touched a tiny portion of the system. However, the fact is, for that tiny portion of the system, these methods were executed in total 159,367 times. By actually running the system, we observed that although only a small portion of the system was impacted at run time, this does not mean other impacts in the static impact set are not valid.

Therefore, dynamic analysis only computed a small set of dynamic impacts (4,806 functions), even though the functions were executed hundreds of hundreds of times. These 4,806 functions were kept in the final impact set, as they were "confirmed" at run time.

Reachability analysis and alias analysis worked on the set of "potential false-positives", focusing on the subtraction of dynamic impacts from static impacts. In the case of patch # 5565583, we had a set of 694,728 functions to work on.

Both CFL-reachability analysis and alias analysis were implemented via `Wala`. A simple control-flow graph constructed in the reachability analysis is in Figure 8.40a.

A regular control-flow graph was "exploded" to a supergraph to contain data-flow facts and hence to identify feasible paths. A sample supergraph (Figure 8.40b) with only feasible paths is also presented.


BB0 (en)
BB1
2 conditional branch(le) v1,v3:#0
BB2
4 v4 = getstatic < Application, Ltest/Test, g, <Primordial,I> >
5 v5 = binaryop(sub) v1 , v4
8 invokestatic < Application, Ltest/Test, process(I)V > v5 @11 exception:v6
BB3
9 v7 = getstatic < Application, Ljava/lang/System, out, <Application,Ljava/io/PrintStream> >
11 invokevirtual < Application, Ljava/io/PrintStream, println(I)V > v7,v5 @18 exception:v8
BB4
12 return
BB5 (ex)
BB0 { 0 }
null
BB1 { 0 }
null
BB2 { 0 }
null
BB3 { 0 }
conditional branch(le) 1,3
BB4 { 0 }
null
BB5 { 0 }
4:=getstatic g
BB6 { 0 }
5 = binaryop(sub) 1 , 4
BB7 { 0 }
null
BB8 { 0 }
null
BB9 { 0 }
call Test.process exc:6 5
BB10 { 0 42 58 59 66 67 70 71 148 149 201 204 205 206 }
7:=getstatic out
BB11 { 0 42 58 59 66 67 70 71 148 149 201 204 205 206 }
null
BB12 { 0 42 58 59 66 67 70 71 148 149 201 204 205 206 }
call PrintStream.println exc:8 7 5
BB13 { 0 42 58 59 66 67 70 71 148 149 201 204 205 206 }
return
BB14 { 0 42 58 59 66 67 70 71 148 149 201 204 205 206 }
null

(a) (b)

Figure 8.40: (a) Control-Flow Graph Sample (b) Supergraph Sample

We ran `Wala` on the enclosing classes of each function in those "potential false-positives" (694,728 for # 5565583), using the `PartiallyUnbalancedTabulationSolver`, and then mapped identified feasible statements to functions in the system. Then those functions with the direct changes (42,400 for patch #5565583) were given to `Wala`'s alias analysis framework, and we used `makeVanillaZeroOneCFABuilder` to find aliased variables for each changed function.

In the end, we found that many of the functions within the "potential false-positives" were not included in the feasible path (611,253) or able to access any aliased variables (863,374) of changed functions. We therefore removed 6,865,697 (37.3 %) dependencies from the original dependency graph. Our final results are shown in Table 8.17.

Patch	**Static Impacts**	**Dynamic Impacts**	**Removed By Reachability**	**Removed By Aliasing**	**Final Impacts**
# 5565583	699,534	4,806	61,125	86,374	552,035
# 14321241	230,209	1,232	19,773	13,665	196,771

Table 8.17: Final Impacts of Patch # 5565583 and Patch #14321241. (We walked through the result analysis for Patch # 5565583 only, since they essentially follow the same process, and Patch # 10107418 has no impact on the system.)

As we can see from Table 8.17, we have achieved a precision of 3.8% at the end of static analysis and improved it to 2.98% at the end of the complete approach. The dynamic analysis identified that only 4,806 functions were executed, which left a large portion (99%) of the static impacts as potential false-positives. Hence with reachability analysis and alias analysis the number of false-positives removed does not seem to be a large reduction, that is, 21.8%. However, we need to note that the reduction achieved is done at no risk to the safety of our overall approach. At the current stage, our instrumentation lacks users' data and code, so the impacts can only point to functions within Oracle E-Business Suite. This explains why, that even with hundreds of hundreds of real executions, the dynamic impacts are associated with just a tiny part of the system.

The running time of each major step in this experiment is listed in Table 8.18.

Static Analysis				
Build Dependency Graph	**Extract Changes**	**Reverse Search**	**Compute Static Impacts**	**Sub Total**
9.5 Hours	2 Hours	3.8 Hours	1 Hour	16.3 Hours
Dynamic Analysis				
Instrument	**Compute Dynamic Impacts**			**Sub Total**
48 Hours	2 Hours			50 Hours
Reachability & Alias Analysis				
Build CFG & Supergraph	**Compute Aliasing**	**Mapping**	**Compute Impacts**	**Sub Total**
10 Hours	7 Hours	3 Hours	2 Hours	22 Hours
Total Analysis Time				**88.3 Hours**

Table 8.18: Execution Time in Experiment 2 for Patch # 5565583

The entire process requires significant time to complete. Considering the size of the system and patch, it is still more manageable than rerunning everything in a regression suite. More crucially, it provides testers confidence as to which parts in the system are affected, in a safe way. The most time-consuming task is the instrumentation, which occupies around 56.8% of the total execution time. Just as for the static dependency graph, the instrumentation forms a substantial corporate asset for future analysis, and can be easily and quickly updated as needed. Also, for the control-flow graph and supergraph, it is not necessary to rerun the entire process every time when there is a new change. One can easily extend the existing graphs by running the same analysis on a much smaller set of newly added program entities.

Chapter 9

Conclusion and Future Work

9.1 Achievement

In this thesis, we investigated how to conduct a hybrid impact analysis on large-scale enterprise systems. Our achievements are:

1. We have developed an improved dependency model named *access dependency graphs*[1] to deal with object-oriented languages like Java that support inheritance and dynamic binding, and have shown it to be equivalent (in terms of finding static dependencies) to other techniques that typically create much larger dependency graphs.

2. We have developed a multi-tasking aspect-oriented instrumentor to adequately instrument large-scale systems and collect traces at bytecode level. The instrumentor does not require testers to fully understand the application logic or prepare any test data. This is extremely useful when the size of the program

[1]This is joint work with Asif Iqbal [Iqb11].

is large. Existing tools are too expensive and require extra information such as test coverage, and operational profiles, which are usually hard to access. The instrumentor is multi-tasking – instrumentation work is divided into multi tasks to maximize the computing power.

3. We have developed a series of script parsers to parse compiled or plain patch files, to extract both direct and indirect changes. Additionally, a string analysis[2] beyond those of the Java String Analyzer was developed in order to extract complete indirect changes between the library and the database.

4. Furthermore, we have incorporated CFL-reachability analysis and alias analysis to identify impacts. As far as we can ascertain, these two techniques have not been used in this way to cut down on the false-positives in preceding analyses. It has been demonstrated that CFL-reachability with a parenthesis context-free grammar can be used to filter out infeasible paths (mis-matched calls and returns) that may become false-positives in the impact set. An alias analysis was used to identify functions that are able to access the aliased and changed variables. We consider those that are not able to access any of the aliased and changed variables to be false-positives if they, themselves, are not directly changed.

5. We have empirically demonstrated the practical applicability of the improved approach on a very large enterprise system, involving hundreds of thousands of classes. Such systems may be perhaps two orders of magnitude larger than the systems analyzed by other approaches, so our technique seems to be uniquely powerful.

[2]This is joint work with Jay Parlar [CIA+12].

Our hybrid approach is *safe*, *precise*, *scalable* and *efficient*. We abstracted a full dependency graph, with significant reduction of redundancies cause by inheritance and dynamic binding, which distinguishes us from other vanilla static analysis. Varieties of techniques were incorporated to achieve the precision. Dynamic analysis was used to "confirm" what should be kept in the final impact set, since run time execution reflects actual system behaviour. The precision was further improved by applying CFL-reachability analysis and alias analysis to the control-flow of the system. These two techniques were aimed at removing false-positives from the remaining "suspect" impacts after the instrumentation. Finally, our hybrid approach is scalable and efficient. This was demonstrated by our empirical study which enabled us to handle systems of two orders of magnitude larger than other approaches.

The final impact set obtained from our approach can be used in regression test selection, focused testing, and planning enhancements to applications, etc. Nowadays, organizations spend a large portion of their financial budget in purchasing, implementing and maintaining enterprise systems, since they could provide a number of benefits as we discussed in the *Introduction* (see Chapter 1, Table 1.1). With our approach, both confidence and cost-savings can be achieved throughout the entire testing process.

9.2 Future Work

As we can observe from the empirical study, the instrumentation took approximately two days to finish even after dividing the task into smaller ones. This amount of run time is reasonable with respect to the size of the system. However, from a software tester's point of view, it may still be useful to reduce the run time by improving the efficiency of the technique. Results of the case study indicate that dynamic analysis only found a small portion of the static impacts are real impacts. It might be the case that runtime use of a large software system may only utilize a small part of the software. However, we may also not have filtered out enough false-positives. To achieve a better running time, a *static slice* may be utilized to slice the system statically first, and then instrumentation can be done on slices of interest. Customized changes or patches from vendors presumably only touch a small portion of the system, hence if a safe slice can be obtained to fully cover the changes, the dynamic instrumentation can be performed "on-the-fly".

Initially, considering the running time and effort expended on the reachability and alias analysis, we were a little disappointed in the percentage of false-positives removed by this technique. However, after examining the results more carefully, we realized that: (1) the actual number of false-positives removed was significant; and (2) as we discussed earlier, because there is no user's application in our case study, the impact analysis is restricted to functions within the system, and in particular, many of the identified impacted functions are system APIs. To further build on this empirical study, we should investigate whether we are able to remove more false-positives with given customized code and data. Also, the alias analysis we used is flow-insensitive and context-insensitive. It assumes statements in the program can be executed in any order and any number of times. In practice this is not a precise approach. In addition, this can be exacerbated by context-insensitivity: method calls were treated conservatively, without computing the precise target addresses of the return statements. Hence, in the future, it is worth investigating whether a more precise approach can be derived for large-scale enterprise systems, by including flow-sensitive and context-sensitive methods.

Organizations tend to identify their test suites by the business process that is being tested, and to think of their system as consisting of (or supporting) business processes rather than code classes. HP Quality Centre, for example, organizes tests by business process. Through analysis of test cases we may be able to relate the affected functions of interest to the business processes that might be affected, and hence present results in a way that is more meaningful to testing teams. In the medium term, there are a number of other related applications that could be achieved with the techniques we have developed. First, we need to extend the work beyond the current Java tools, to systems written in other languages such as COBOL. The modular design of our system, especially an analysis based on XML, means that the primary task would be to develop language-dependent front ends for each such extension. In fact, non-object oriented languages would not be susceptible to some of the complications introduced by inheritance, for example.

We started out intending to analyze vendor-supplied patches. However, we could have started out with any method, field or database object that the user might intend to change. We could then identify which existing tests might execute or depend on that selected item. This can help users improve test cases. Such work might be a prelude, and complementary, to dynamic analysis to examine test coverage. Indeed, our analysis makes such dynamic analysis feasible. The dependency graph identifies the possible methods that might be called from a given method. If you are testing that method, and want to have some idea of the coverage of your tests, the relevant baseline is the subgraph of the dependency graph with the method being tested at its apex, not the whole of the library, which is otherwise all you have. Any particular organization probably only uses a tiny fraction of the whole library, and the subgraph of the dependency graph containing that organization's methods of interest is the only part they need to be concerned with.

Finally, impact analysis can be used in planning enhancements to applications. Once methods or database objects that are intended to be changed are identified, typically in the detailed design stage, the same impact analysis as we use on changes caused by patches can be done to indicate where the potential effects are. This raises a number of possibilities. The testing necessary to cover all possible impacts can be planned. Or, perhaps, the design may be revisited to try to reduce the possible impact.

Bibliography

[AB93] Robert S Arnold and Shawn A Bohner. Impact analysis-towards a framework for comparison. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 292–301. IEEE, 1993.

[Abd10] Akbar Abdrakhmanov. Analyzing file extensions from the patches. 2010.

[AG12] SAP AG. Annual report 2012, financial highlights, 2012.

[AH02] Taweesup Apiwattanapong and Mary Jean Harrold. Selective path profiling. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 35–42. ACM, 2002.

[AK97] K. Abdullah and K.White. A firewall approach for the regression testing of object-oriented software. In *Proceedings of the 10th Annual Software Quality Week*, May 1997.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[AOH04] Taweesup Apiwatanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings* *of the 19th IEEE international conference on Automated software engineering*, pages 2–13, 2004.

[Api05] Taweesup Apiwatanapong. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, 2005.

[ASH86] Anant Agarwal, Richard L Sites, and Mark Horowitz. *ATUM: A new technique for capturing address traces using microcode*, volume 14. IEEE Computer Society Press, 1986.

[Asp14] AspectJ. Aspectj main page, 2014.

[Bal98] Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, volume 23 issue 2, March 1998.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Languages and Compilers for Parallel Computing*, pages 234–250. Springer, 1995.

[BDSP04] B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock. Online impact analysis via dynamic compilation technology. In *20th IEEE International Conference on Software Maintenance*, 2004.

[BHRG09] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce.

Rule systems for runtime verification: A short tutorial. In *Runtime Verification*, pages 1–24. Springer, 2009.

[Boh96] Shawn A. Bohner. Software change impact analysis. 1996.

[BRR01] John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. volume 2 Issue 2, pages 149–183, April 2001.

[BS96] D.F. Bacon and P.F. Sweeney. Fast static analysis of fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications,*, volume 31 Issue 10 of ACM SIGPLAN Notices, pages 324–341. ACM Press, New York, October 1996.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1993.

[CC07] Kung Chen and Ju-Bing Chen. Aspect-based instrumentation for locating memory leaks in java programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 23–28. IEEE, 2007.

[Cen13] IBM T.J. Watson Research Center. T. j. watson libraries for analysis main page, Jul 2013.

[Cho11] Stephen Chong. Dataflow analysis, 2011.

[CIA$^+$11] Wen Chen, Asif Iqbal, Akbar Abdrakhmanov, Chris George, Mark Lawford, Tom Maibaum, and Alan Wassyng. Report 7: Middleware Change Impact Analysis for Large-scale Enterprise Systems. Technical Report 7, McMaster Centre for Software Certification (McSCert), September 2011.

[CIA$^+$12] Wen Chen, Asif Iqbal, Akbar Abdrakhmanov, Jay Parlar, Chris George, Mark Lawford, T. S. E. Maibaum, and Alan Wassyng. Change impact analysis for large-scale enterprise systems. In *ICEIS (2)*, pages 359–368, 2012.

[CIA$^+$13] Wen Chen, Asif Iqbal, Akbar Abdrakhmanov, Jay Parlar, Chris George, Mark Lawford, Tom Maibaum, and Alan Wassyng. Large-scale enterprise systems: Changes and impacts. In *Enterprise Information Systems*, pages 274–290. Springer, 2013.

[CK95] Bob Cmelik and David Keppel. *Shade: A fast instruction-set simulator for execution profiling.* Springer, 1995.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[Cor10] Oracle Corporation. Oracle e-business suite concepts release 12.1 part number e12841-04, 2010.

[CRa94] Yih-Farn Chen, D.S. Rosenblam, and Kiem-Phong Vo and. Testtube: a system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.

[CSB98] Bruce Caldwell, Tom Stein, and ERP Beyond. New it agenda. *Information Week*, 711:30–34, 1998.

[CWM14a] Wen Chen, Alan Wassyng, and Tom Maibaum. Combining static and dynamic impact analysis for large-scale enterprise systems. In *The 15th International Conference on Product-Focused Software Process Improvement*, Helsinki, Finland, 2014.

[CWM14b] Wen Chen, Alan Wassyng, and Tom Maibaum. Impact analysis via reachability and alias analysis. In *The 7th International Conference on the Practice of Enterprise Modelling*, Manchester, United Kingdom, 2014.

[CZvD$^+$09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Sigplan Notices*, volume 29, pages 230–241. ACM, 1994.

[DG03] Markus Debusmann and Kurt Geihs. Efficient and transparent instrumentation of application components using an aspect-oriented approach. In *Self-Managing Distributed Systems*, pages 209–220. Springer, 2003.

[Doa07] Matthew B. Doar. *JDiff - An HTML Report of API Differences*, 2007. Electronically available at `http://javadiff.sourceforge.net/`.

[DS10] Julian Dolby and Manu Sridharan. Static and dynamic program analysis using wala, 2010.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Notices*, volume 29, pages 242–256. ACM, 1994.

[EN08] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, 2008.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[FF97] F.I.Vokolos and P.G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *3rd internatinal conference on on*

Reliability, quality and safety of software-intensive systems, pages 3–21, 1997.

[FYD$^+$08] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[gar11] IT Key Metrics Data 2012. Gartner, Inc, December 2011.

[Ger06] Judith L Gersting. *Mathematical structures for computer science.* Macmillan, 2006.

[GHK$^+$01] Todd L. Glaves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. volume 2 Issue 10, pages 184–208, April 2001.

[GPT$^+$09] Emmanuel Geay, Marco Pistoia, Takaaki Tateishi, Barbara G Ryder, and Julian Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 177–187. IEEE, 2009.

[Han00] George Hansper. Yacc - a parser generator, 2000.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[HGF$^+$08] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio. On the precision and accuracy of impact analysis techniques. In *Computer* *and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 513 –518, may 2008.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 36 Issue 11, November 2001.

[HLK+97] Pei Hsia, Xiaolin Li, David Chenho Kung, Chih-Tung Hsuand Liang li, Yasufumi Toyoshima, and Cris Chen. A technique for the selective revalidation of oo software. volume 9 Issue 4, pages 217–233, July-August 1997.

[How98] D. Howe. Legacy system from foldoc, 1998.

[HS07] Lulu Huang and Yeong-Tae Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Software Engineering Research, Management Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 374 –384, aug. 2007.

[IBM13] IBM. Userguide:pointeranalysis, 2013.

[Iqb11] Asif Iqbal. Identifying modifications and generating dependency graphs for impact analysis in a legacy environment. 2011.

[jar] *Jar Compare Tool*. Electronically available at `http://www.extradata.com/products/jarc/`.

[JR00] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. *Science of Computer Programming*, 36(2-3):267 – 284, 2000.

[KGH+94a] David Chenho Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. Firewall regression testing and software maintenance of object-oriented systems. 1994.

[KGH+94b] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in object oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 202–211, September 1994.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[KPR00] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. In *Proceedings of the*

22nd international conference on Software engineering, pages 126–135, June 2000.

[Lar99] James R Larus. Whole program paths. In *ACM SIGPLAN Notices*, volume 34, pages 259–269. ACM, 1999.

[LBL+10] Patrick Lam, Eric Bodden, Ondrej Lhotak, Jennifer Lhotak, Feng Qian, and Laurie Hendren. *Soot: a Java Optimization Framework*. Sable Research Group, McGill University, Montreal, Canada, March 2010. Electronically available at `http://www.sable.mcgill.ca/soot/`.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[LH99] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):199–215, 1999.

[LH06] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction*, pages 47–64. Springer, 2006.

[Lho02] Ondrej Lhoták. Spark: A flexible points-to analysis framework for java. 2002.

[LMS97] J.P. Loyall, S.A. Mathisen, and C.P. Satterthwaite. Impact analysis and change management for avionics software. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, volume 2, pages 740–747, 1997.

[LR03a] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[LR03b] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[LR04] William Landi and Barbara G Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 39(4):473–489, 2004.

[LS06] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator, 2006.

[LW91] H.K.N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, pages 201–208, October 1991.

[LW92] H.K.N. Leung and L. J. White. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance*, pages 262–271, November 1992.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification: The **class** File Format*. Sun Microsystems, 1999. Electronically available at `http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html`.

[LYC+08] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 213–224. ACM, 2008.

[MBdFG10] Mirna Carelli Oliveira Maia, Roberto Almeida Bittencourt, Jorge Cesar Abrantes de Figueiredo, and Dalton Dario Serey Guerrero. The hybrid technique for object-oriented software change impact analysis. *Software Maintenance and Reengineering, European Conference on*, 0:252–255, 2010.

[MR97] David Melski and Thomas Reps. *Interconvertbility of set constraints and context-free language reachability*, volume 32. ACM, 1997.

[MT00] M Lynne Markus and Cornelis Tanis. The enterprise systems experience–from adoption to success. *Framing the domains of IT research: Glimpsing the future through the past*, 173:207–173, 2000.

[MW00] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[MW08] Ellen F Monk and Bret J Wagner. *Concepts in enterprise resource planning*. CengageBrain. com, 2008.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[NMW97] Craig G Nevill-Manning and Ian H Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference, 1997. DCC'97. Proceedings*, pages 3–11. IEEE, 1997.

[OAH03] Alessandro Orso, Taweesup Apiwatanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 28 Issue 5, September 2003.

[OAL$^+$04] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[O'C01] Robert O'Callahan. *Generalized aliasing as a basis for program analysis tools*. PhD thesis, Carnegie Mellon University, 2001.

[Ora10] Oracle. Oracle e-business suite integrated soa gateway implementation guide release 12.1, June 2010.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM*

SIGSOFT twelfth international symposium on Foundations of software engineering, volume 29 Issue 6, November 2004.

[PA98] Shari Lawrence Pfleeger and Joanne M Atlee. *Software engineering: theory and practice*. Pearson Education India, 1998.

[PA06] S.L. Pfleeger and J.M. Atlee. *Software Engineering: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 2006.

[PA09] Anil Passi and Vladimir Ajvaz. *Oracle E-Business Suite Development & Extensibility Handbook*. McGraw-Hill, Inc., 2009.

[PH08] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2008.

[Pro14] Eclipse AspectJ Project. Introduction to aspectj, Feb 2014.

[PSM95] Jim Pierce, Michael D Smith, and Trevor Mudge. Instrumentation tools. In *Fast Simulation of Computer Architectures*, pages 47–86. Springer, 1995.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 13–22. IEEE, 1999.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701 – 726, 1998.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. volume 22 Issue 8, pages 529–551, August 1996.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. volume 6 issue 2, pages 173–210, April 1997.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. volume 24 Issue 6, pages 401–419, June 1998.

[RHD00] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. volume 10 Issue 2, pages 77–109, June 2000.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.*, 39:432–448, October 2004.

[RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

[SAG$^+$06] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *Software Engineering, IEEE Transactions on*, 32(7):454–466, 2006.

[SAP14] SAP. Sap modules. `sap.wikia.com/wiki/SAP_modules`, 2014.

[SB06a] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, June 2006.

[SB06b] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1997.

[SHR$^+$00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35 Isuue 10, October 2000.

[SO05] Arjan Seesing and Alessandro Orso. Insectj: A generic instrumentation framework for collecting dynamic information within eclipse. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 45–49, New York, NY, USA, 2005. ACM.

[Sof15] Gudu Software. General sql parser: Parsing, formatting, modification and analysis., 2015. `http://www.sqlparser.com`.

[Sos04] Dennis Sosnoski. Java programming dynamics, part 7: Bytecode engineering with bcel, Apr 2004.

[Sri07] Manu Sridharan. *Refinement-Based Program Analysis Tools*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2007.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[Tar72] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tes10a] Jean Tessier. *Dependency Finder*. 2010. Electronically available at `http://depfind.sourceforge.net/`.

[Tes10b] Jean Tessier. *The Dependency Finder User Manual*, November 2010. Electronically available at `http://depfind.sourceforge.net/Manual.html`.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[TM94] Richard J. Turver and Malcom Munro. An early impact analysis technique for software maintenance, January 1994.

[TPF$^+$09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[WC11] Jay Parlar Wen Chen. Impact analysis of oracle e-business suite - user guide. Technical report, McMaster Centre for Software Certification (McSCert), McMaster University, November 2011.

[wik11] *Dynamic dispatch – Wikipedia, the free encyclopedia.* 2011. Electronically available at `http://en.wikipedia.org/wiki/Dynamic_dispatch`.

[WL04] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[WMFB$^+$98] Robert J Walker, Gail C Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *ACM SIGPLAN Notices*, volume 33, pages 271–283. ACM, 1998.

[ZBZ11] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th international conference on World wide web*, pages 805–814. ACM, 2011.