Impact-Driven Regression Test Selection for Mainframe Business Systems

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Abhishek Dharmapurikar

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Master's Examination Committee:

Professor Jayashree Ramanathan, Advisor

Professor Rajiv Ramnath

Copyright by

Abhishek Dharmapurikar

2013

Abstract

Software testing is particularly expensive in the case of legacy business systems such as mainframes. These systems are critical to many large enterprises, yet they are perpetually in maintenance where even small changes to the system usually lead to an end-to-end regression test. Due to the age of legacy systems there is a lack of knowledge of component inter-dependence resulting in comprehensive system tests that have to be conducted in production environments. This is called the "retest all" approach which is done to ensure confidence in the functioning of the system. But this approach is impractical primarily due to a) resource needs (including man-hours, test cycle time and production infrastructure contention), and b) user stories generated within the agile system that require changes to the system at an ever-faster pace.

This research is aimed at reducing the required regression testing and the costs associated with the system and its assets (such as the database schemas, files (datasets), transactions, screens, source programs and copybooks). The improvements are achieved by identifying only those tests needed by assets changes and others that are 'impacted'. The impact analysis leverages the availability of modern static code analysis tools like Rational Asset Analyzer and dedicated test environments for mainframes. We show that by using impact analysis on a real-world mainframe application the test savings maybe much greater than 34%.

Dedication

This document is dedicated to my family and friends.

Acknowledgments

I owe my deepest gratitude to my advisor Prof. Jayashree Ramanathan for her vision and constant support throughout my Master's program. Her enthusiasm for enterprise research ideas has been great source of inspiration for me. I would also like to thank Prof. Rajiv Ramnath for the constant supervision right from the beginning of my Master's program and also for agreeing to serve on my Masters Examination committee. I would especially like to thank Nationwide for giving me opportunity to work with them for my research, especially my team members in Build Capability, Vishnu Dendukuri, Carmen DeArdo and Praveen Kumar for the mainframe training which my thesis is based on. Ben Wierville has been of great help for giving valuable feedback and for supervising the research as it progressed. I would also like to thank all my friends here at Ohio State University for making this journey pleasant one. My sincerest gratitude to the Furnos, who have always given me moral support and care whenever needed. Finally, this endeavor would not have been possible without the support of my family members, who have encouraged and motivated me to no end. It is to the Dharmapurikars that I would like to dedicate my work to.

Vita

June 2005	S.B.E.S. College of Science, Aurangabad, India
May 2009	B.Tech. Computers, College of Engineering Pune
August 2011 to present	Graduate Research Associate, Department of Computer Science and Engineering, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The legacy systems such as mainframes are still being used by many enterprises, but constantly changing to meet the evolving modern enterprise models. Typically any system goes through a certain evolution activities which can be divided into three categories [3] maintenance, modernization, and replacement (see Figure 1). After being built the system goes through maintenance activities to keep up with the changing business needs. A modernization effort is then required that represents a greater effort, both in time and functionality, than the maintenance activity. Finally, when the old system can no longer be evolved, it must be replaced. As for the mainframe systems that have been modernizing to keep up, reducing the testing costs would immediately benefit.

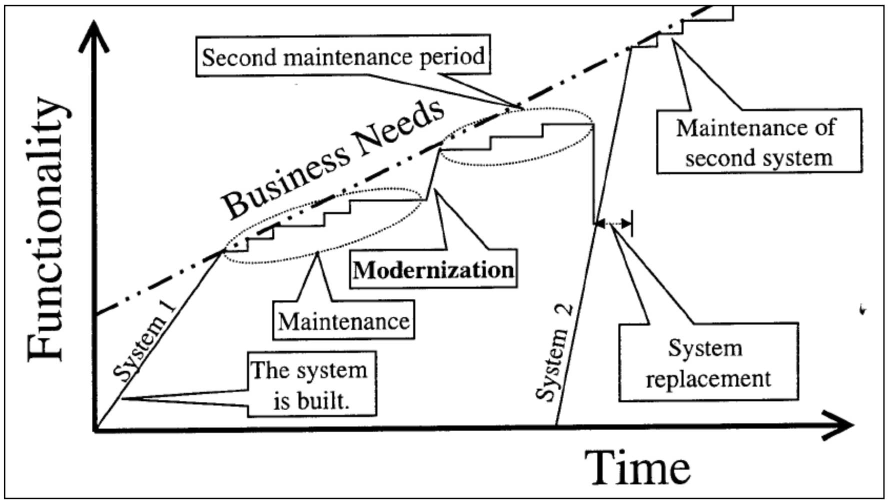

Figure 1. The phases of system evolution

Software testing is the most critical and expensive phase of any software development life cycle. According to Rothermel et al. [5], a product of about 20,000 lines of code requires seven weeks to run all its test cases and costs several hundred thousands of dollars to execute them. Software maintenance activities, on an average, account for as much as two-thirds of the overall software life cycle costs [1]. Among activities performed as part of maintenance, regression testing takes large amounts of time as well as effort, and often accounts for almost half of the software maintenance costs [2]. Regression testing by definition (also referred to as program revalidation) is carried out to ensure that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program) [2]. With mainframe systems containing several thousands of programs, usually an end to end regression test is carried out using test cases from system tests. This black box testing technique is the only practical way of assuring compliance and owing to the lack of knowledge of dependence among components; it is not possible for the system testers to test only the affected components of the system resulting from a change.

Related Work

There have been many studies to reduce the cost associated with regression testing. Test case reduction techniques are aimed to compute a small representative set of test cases by removing the redundant and obsolete test cases from test suites [6], [7], [8], [9], [10], [11]. These techniques are useful when there are constraints on the resources available for running an end to end regression. Test case prioritization techniques aim at ranking the test cases execution order so as to defect faults early in the system [5]. It provides a way to find more bugs under a given time constraint, and because faults are detected earlier, developers have more time to fix these bugs and adjust the project schedule. Khan et al. in [12] have given comparison of both the techniques and the effect on software testing. Test case prioritization techniques only prioritize the test cases but do not give a subset of cases which would reveal all the faults in the changed system. Test case reduction techniques do give a reduced number of test cases but the coverage of the reduced test cases spans across the entire system including the parts which were not changed.

Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test that the affected but unmodified parts of a program continue to work correctly. Use of an effective RTS technique can help reduce the testing costs in environments in which a program undergoes frequent modifications. Rothermel and Harrold [13] have formally defined the regression test selection problem as follows: *Let P be an application program and P′ be a modified version of P. Let T be the test suite developed initially for testing P. An RTS technique aims to select a subset of test cases T′ ⊆ T to be executed on P′, such that every error detected when P′ is executed with T is also detected when P′ is executed with T′.* Impact-Driven Regression Test Selection (ID-RTS) builds on this idea and aims at reducing test costs for mainframe systems. As this idea is proposed as a replacement for the retest-all regression tests, it is expected that the idea should be safe – it should be able to identify all the test cases that could reveal modifications to the system.

There have been many techniques presented for regression test selection. Code based techniques (also called program based techniques) look at the code of programs and select relevant regression test cases using control flow, data or control dependence analysis, or by textual analysis of the original and the modified programs.

Dataflow analysis-based RTS techniques explicitly detect definition-use pairs for variables that are affected by program modifications, and select test cases that exercise the paths from the definition of modified variables to their uses [17][18]. However, these techniques are not safe due to their inability to detect the effect of program modifications that do not cause changes to the dataflow information. The techniques also do not consider control dependencies among program elements for selecting regression test cases. As a result, these techniques are unsafe [16][4].

Control flow techniques [21][22] model control flow of input programs into control flow graphs and analyze a change on them for selecting regression test cases. These techniques have been proven safe and the graph walk approach suggested in [22] is the most precise work for procedural languages [16]. However they do not include non-code based components of the system such as DB and files.

Dependence based RTS techniques look at the data and control dependencies among or within the programs to filter out the modified test cases. Program dependence graph based technique suggested in [19] could work on a single program, which was later improved to system wide scope in [20] by using System dependence graph, but both the techniques were proved to be unsafe as they omit tests that reveal deletions of components or code [4].

Differencing technique [19] is an RTS technique based on textual differencing of the canonical form of the original and the modified programs. Though this technique is safe, it requires conversion of programs into a canonical form and is highly language dependent. Also the complexity of this approach is too high to be feasible for a mainframe system with several thousand programs [16].

Slicing based techniques [23] select those test cases which can produce different outputs when executed with the modified program version P. Agarwal et al.[23] have defined 4 different slicing techniques. The basic slicing technique forms an *execution slice* which is the set of program statements in program P that are executed for a test case t. Other three techniques build on the basic technique picking statements that influence an output, or predicate statements that affect the output. The overall technique would select a test case t only if the slice of t computed using any one of the four approaches contains a statement modified in P. These techniques are precise, however they have been shown to omit modification-revealing tests, hence are not safe [16].

The most relevant research to ID-RTS is the firewall based approach in [24]. A firewall is defined as a set of all modified modules in a program along with those modules which interact with the modified modules. This technique uses a call graph to establish control flow dependencies among the modules. Within the firewall, unit tests are selected for the modified modules and integration tests for the interacting modules. This technique is safe as long as the test suite is reliable [4].

Research has also been done on specification based regression test selection. Such techniques look at the specification of a program by modeling the behavior [14] and/or requirements of a system [15]. These techniques do not employ the dependency extracted from static code analysis of programs and hence are not precise or safe [16].

Impact-Driven Regression Test Selection (ID-RTS) is a control flow and data dependence based, intra-procedural regression test selection technique designed for mainframes. It filters out test cases based on the following steps (see Figure 2)

1. Comprehensively representing the inter-asset dependencies using a dependence graph.
2. Analyzing the types of changes to the system. This involves accounting for insertion, modification or deletion of an asset.
3. Filtering out the affected interfaces and associated test cases for a system through impact analysis for a particular change.

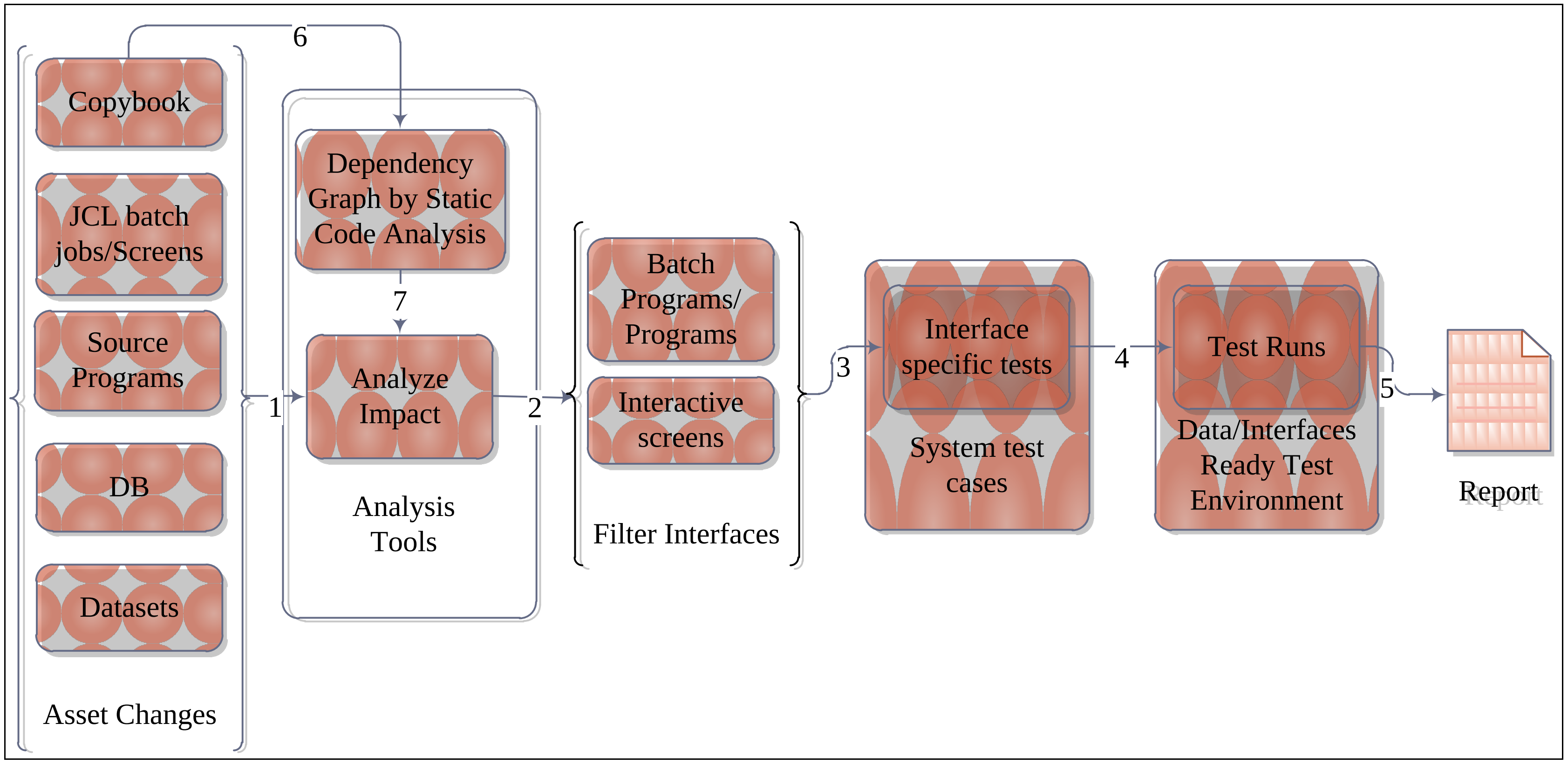

Figure 2. The ID-RTS process

Motivation

With the growing volatility of the requirements even for the legacy systems such as mainframes, Agile methods are being adopted as according to the agile manifesto [27], they enable projects to provide fast, frequent, consistent, and continuous delivery of working software; and to respond to changing requirements. D. Talby et al. in [28], have pointed out the importance of integrating the testing and development activities in an agile environment. As with mainframes combined with the retest-all approach, a) the dedicated resources required for the system tests reduce the availability of the mainframe systems, and b) the time required for these tests inhibits making changes and testing the system in the same agile iteration. Even though the development of the change requirements is carried out in an agile fashion, the testing has to be completed as part separate story in a different iteration. The developers have to wait for a complete iteration to gather the test results of the changes that were made.

This research is primarily aimed at replacing the system tests with a safe regression test selection technique (ID-RTS) based on impact analysis of changes. This technique reduces the test cycles required for each change facilitating testing in the development iteration itself. This enables the developers to get the feedback on the changes sooner, and improves the speed of bug fixes (if any encountered) as the knowledge of the changes done would still be current. To replace a system test, the regression testing technique has to be safe, i.e. if there are any faults that are created by the change of code; they have to be exposed by the technique. ID-RTS is safe as we prove it in the Chapter 4.

Modernization of mainframe development has been planned in order to enhance the productivity of the development teams and quality of the software produced. It involves the use of IDEs, analysis tools and dedicated unit testing environments for the development of COBOL code. Analysis tools provide an insight on the dependencies and complexities of the assets and also gauge the feasibility of a change using impact analysis. ID-RTS leverages the availability of these tools and environments to further enhance the speed of testing and bug fixes.

In the as-is retest-all process, the testing is carried out at application granularity. If changes are made to an application, that particular application and the dependent applications are tested. The dependencies are established using the specifications of the applications. ID-RTS on the other hand, filters out tests at interface granularity. The dependencies amongst the assets are drawn out on the system level, to which the applications boundaries are transparent. The impact of a change transparently scales to the interfaces of other applications and only these impacted interfaces inside the system need to be tested. This provides even greater savings on a system wide scope, as only the interfaces of the applications that are affected by the change of the asset are filtered and tested.

Contributions

The contributions made as part of the research were

1) Identification of all dependencies amongst the assets in mainframe systems, as listed in Chapter 2.

2) Representation of the dependencies amongst the assets using a system dependence graph by static analysis of the programs, copybooks, JCLs and PROCs.
3) Identifying the types of asset changes in the system and identifying impact by running graph traversal on the assets that were modified, added or deleted, as discussed in Chapter 3.
4) Proposal of Impact analysis to find a set of affected interfaces and testing only these impacted assets to achieve savings over system tests.
5) As an RTS technique, analysis of ID-RTS against Inclusiveness, Precision, Efficiency and Generality, as discussed in Chapter 5.
6) Methodology of using analysis tools and data ready testing environments which are being implemented as part of modernization projects to aid impact analysis and overall maintenance testing process and proposal of testing process steps to be carried as part ID-RTS using these tools, as depicted in Figure 2.
7) Validation of savings achieved in ID-RTS over the retest-all system test technique in an experiment conducted in Chapter 5 and 6.
8) Limitations of using ID-RTS as a testing technique to replace system tests and the possible methods to overcome these, as discussed in Chapter 7.

The rest of the research thesis discusses ID-RTS. Chapter 2 of this research thesis analyzes the structures of the assets and the dependencies among them. Chapter 3 describes test case selection through impact analysis. Chapter 4 analyzes the efficiency of the RTS technique using standardized metrics. Chapter 5 describes an experiment carried out to gauge the savings from using this technique. Chapter 6 analyzes the results from the experiment and extrapolates savings for a year for an actual enterprise using real data for changes in that period.

Chapter 2: Mainframe Assets, Structures and Dependencies

In order to make a safe dependence based RTS, all dependencies within the mainframe system must be represented. The main language that runs on mainframes is COBOL (COmmon Business-Oriented Language), originally consisted of source programs, copybooks, JCLs, PROC files and record oriented files.

The COBOL source programs consist of four divisions a) identification division containing the information about the source programs ids, b) environment division containing information about the environment the program runs in, c) data division containing the local variable information and d) procedure division containing the actual code of execution. A peculiar characteristic of COBOL programs is that they have single entry and exit points into the program. There could exist PERFORMS inside, but their scope lies only within the program. Thus, the program has to be executed in its entirety from the entry to exit point and no particular section of code within the source program can be executed standalone.

A COBOL copybook is a section of code that defines the variables/data structures, which can be included in source programs to reuse structures among them. They could be elementary data types or some group of such data items called the group data item. It also contains the record structure for various files inside the system.

JCL or Job Control Language is a scripting language used in mainframes to instruct the system on how to run a batch job or start a subsystem. A job consists of one or several steps, each of which is a request to run one specific program and provides input for the program to execute on. Input could be passed from the JCLs itself or from files linked through the DD statements. PROCs are pre-written JCL for steps or groups of steps, inserted into a job allowing reusability of commonly used steps among JCLs.

Files, usually record oriented, are used for interfacing between programs or as input or output from a program. The structure of a record is usually defined in a copybook which the programs manipulating the file would include.

Mainframe systems have evolved overtime to support many modern features such as relational databases, multiple file systems and layouts, transaction and information management systems etc. The source for all components would form *assets* of the system, which consist of files, source programs, database tables and batch jobs. This section would highlight the dependencies that would exist among the various assets.

N. Wilde in [26], has listed out all the possible dependencies that can exist among and within programs. The concepts mentioned can be extended to represent the dependencies amongst the assets in mainframes.

The following topics would describe the data and control dependencies that would exist among the assets to form the dependence graph. This graph would then be used to analyze the impact of a change on any assets. In general, *assets would be represented by the nodes in the graph and the dependencies by the edges between them.*

Source Program - Copybook Dependencies

As mentioned before, copybooks contain data structure variables that would be used in the source programs. COPY statements are used inside the programs to include and use these data structures inside COBOL programs. The compiler expands the copybooks inline inside the programs, so that the references are resolved.

To establish dependencies among the copybooks and programs the definition-usage model proposed by the dataflow methods is used. If a variable is defined inside the copybook and is used inside a program a dependency exists and an edge between the program and the copybook exists in the dependence graph.

As a good programming practice, due to the inline expansion of the copybooks inside the code, it is advisable to design copybooks such that they contain minimum number of structures that several programs would use. For e.g. Copybook A contains definition of variables, *vara* and *varb* and program *prga* includes this copybook but uses only *vara* and similarly program *prgb* uses only *varb*. The copybook would expand inline and the programs will end up having unused variables inside. Ideally there should be two copybooks defined each for *vara* and *varb* and only the required copybook should be included by the programs. With such a practice, only those programs impacted by a change in the copybook file can be extracted. However these practices cannot be imposed upon programs that have already been written and are running in production.

The dependencies are hence established at the data structure level, i.e. instead of the entire copybook, the data structures inside it would now form nodes in the graph. An edge is drawn from the source program to a node if that data structure is used by the program (E.g. PGM1 – VAR2 dependency in Figure 4).

Source – Source Dependencies

COBOL programs can call other programs through the CALL statement. The programs can be called by directly using the program name as a literal or using an identifier contained in a constant or variable. If a program A calls another program B it creates a dependency on B which is represented by an edge in the graph (E.g. PGM3 – PGM4 dependency in Figure 4).

If a program is called using its name stored in a variable whose value is dynamically populated during the execution of the calling program, the dependencies amongst the programs cannot be determined through static analysis. For the scope of this research, coding best practices should be established to avoid dynamic calls. We have verified that the system under test does not have any such dynamic calls.

Source – File (Dataset) Dependencies

COBOL programs use a file as intermediate output between two programs or as terminal output. Files are opened in INPUT, OUTPUT or EXTEND mode which corresponds to file read, write or append respectively. If the program opens the file in read mode then the program is dependent on the file (E.g. PGM2 – FILE1 dependency in Figure 4) and if it opens in write mode, the file depends on the program file (E.g. FILE1-PGM1 dependency in Figure 4). This helps to establish a transitive dependency between programs that write to a file and those that read it (PGM2 is transitively dependent on PGM1). Both control files and data files can be represented using the same model.

JCL 'DD' statements are used to identify files that the program will reference. The function of a DD Statement is to form a logical connection between an actual file and an identifier a COBOL program will use to refer to that file. The file based dependency can also be associated with the JCLs instead of the source without any change in the impact analysis. However for the scope of this paper the former approach is taken.

Source - Database Dependencies

As with source programs and files, dependencies also exist between the programs and databases. COBOL uses EXEC SQL statement to embed SQL queries into the source programs. Similar to files, tables can be read from or written to. From the way tables are accessed in a program a dependency can be established. Update, insert or delete queries are written to the database making the table dependent on the program. Select queries make the program dependent on the database.

However, a program might not use all the attributes from the table. The dependency has to be classified into two types. One dependency when the program accesses all attributes from the table using a '*' in the SQL statement, in which case the program is dependent on the entire table (E.g. TABLE2-PGM4 dependency in Figure 4). Any change in the structure of the table would affect these programs. Other dependency arises when the program accesses limited attributes from the table. Such dependency relations have to be maintained at the attribute level (E.g. the dependency between ATTR1 and PGM2 in Figure 4).

Stored procedures can be treated as programs that have SQL statements to manipulate the DB, and hence this creates a source to DB dependency. Source to Source dependencies would exist between the calling program and the stored procedures. DDL queries do not create any dependencies as they are not executed along with the transactions in the system. Hence, execution of DDLs is treated as changes to the database and impact analysis is carried out on the affected tables and attributes as discussed later in Chapter 3.

JCL – Source Dependencies

JCLs are used to run COBOL programs in batch mode. They contain steps that execute PROCs or programs and report the return code of the execution indicating if the job has failed or passed. JCLs are dependent on a program if in any step they are executing that program (E.g. BATCH1 – PGM2 dependency in Figure 4). If a PROC is executed in any step the JCL is then dependent on the PROC, and in turn the PROCs are dependent on the programs they execute.

Screen – Source Dependencies

Screens are like programs, but can be executed online as a transaction. They can be treated as a program and have same dependencies as a program would have (E.g. SCREEN1 - PGM1 dependency in Figure 4).

Copybook - Copybook Dependencies

COPY statements can exist within copybooks too. If a copybook A is copied by another copybook B, all variables in B using variables in A are dependent on the used variables. To preserve the variable level granularity in copybooks, the dependency is mapped between the variable's definition and declaration.

There are several other types of assets that can exist in the mainframe system, for e.g. report writer programs, assembler programs etc. These assets can be categorized into types that were discussed and dependency rules of that type can be applied to them. In general, i) if an asset A uses code from or transfers control to an asset B, then A is said to be dependent on B, ii) If an asset A writes data into an asset B, then B is dependent on A. On the other hand, if an asset A reads data from an asset B, then A is said to be dependent on B. Figure 3 depicts dependencies for an interface (called run unit) from run unit diagram in RAA.

To any mainframe system, batch jobs and online transactions act as interfaces. These batch jobs and transactions give a certain output based on the input provided. The output can be written to the screen of the transaction or to a file or database changing the state of the system. System test cases are run against all these interfaces to poll for the outputs corresponding to the inputs to be tested (see Figure 4).

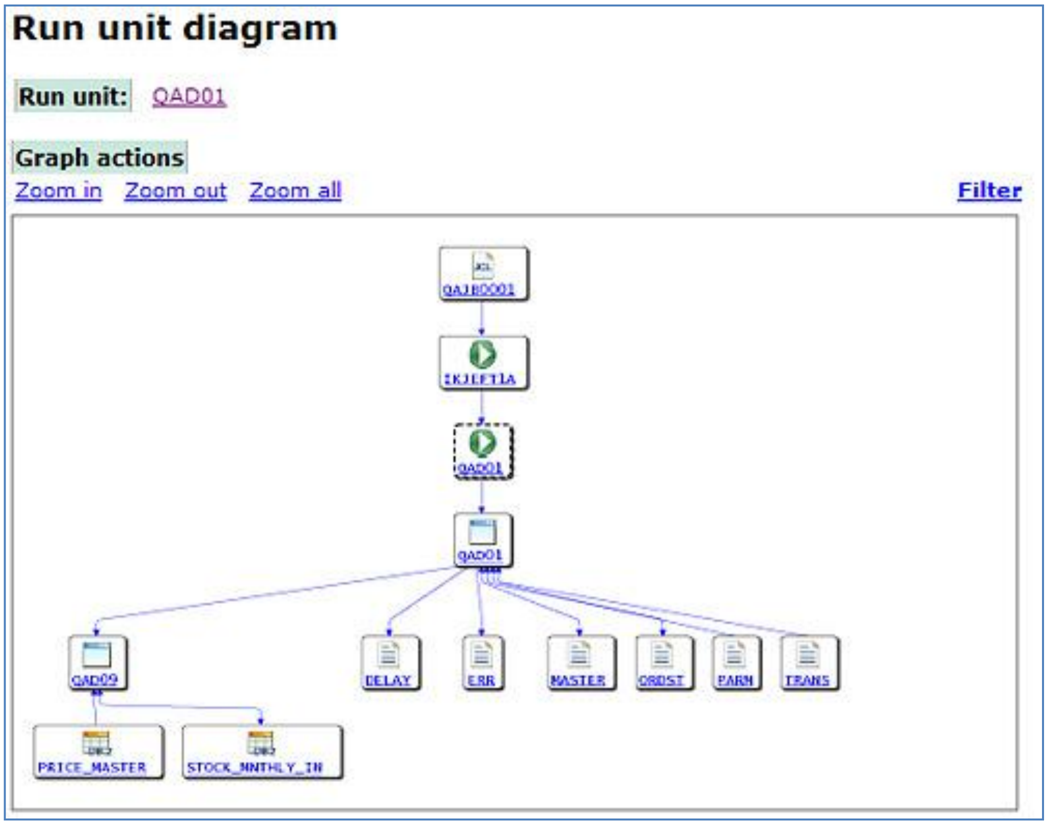

Figure 3. Dependencies from run unit diagram in IBM RAA.

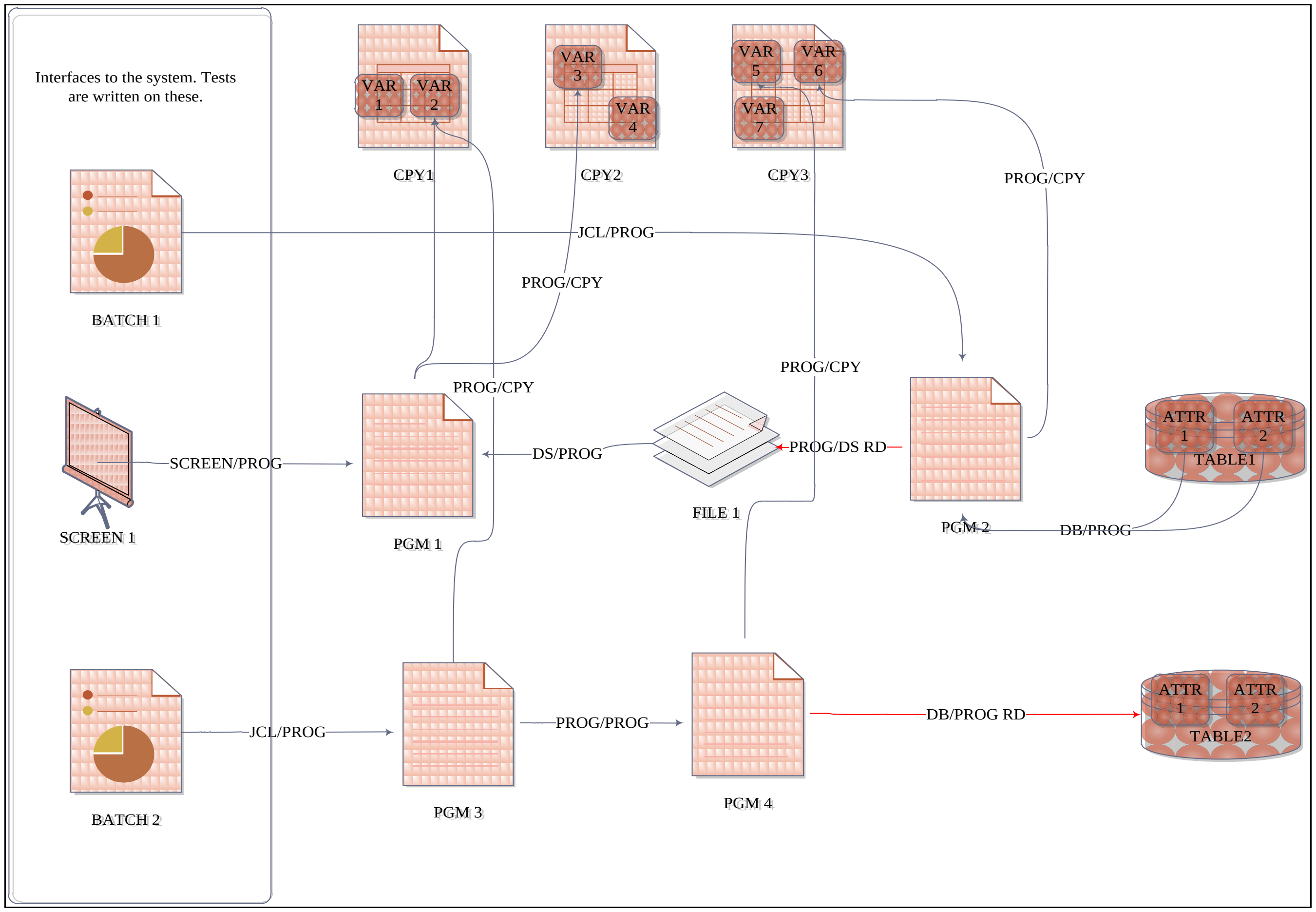

Figure 4. A typical mainframe system with all dependencies

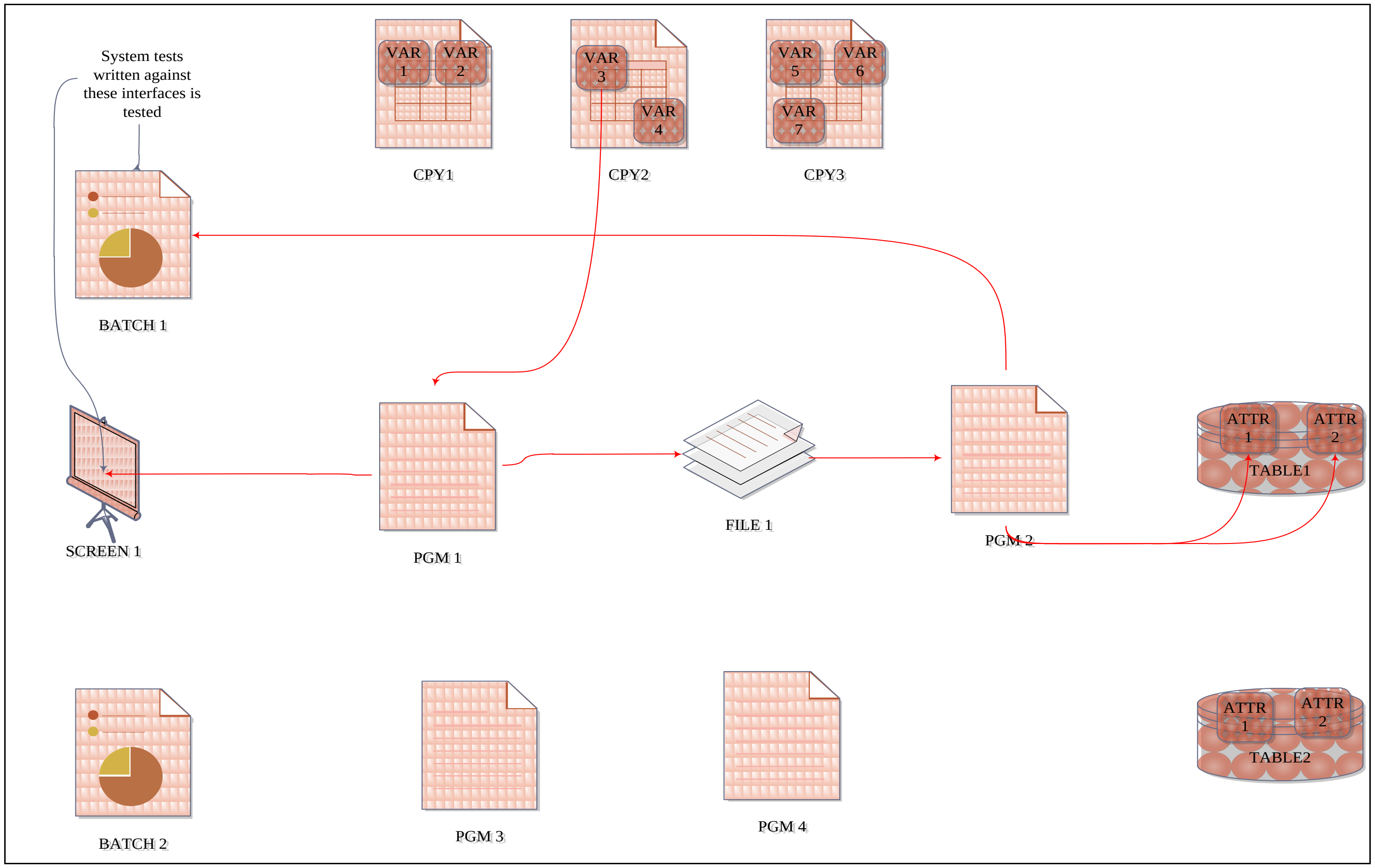

Figure 5. The impact of change in VAR3. Starting from the asset changed graph walk is done on the inverse of the graphs and the affected interfaces are filtered

Chapter 3: Filtering Interfaces and Tests

With the static analysis of all the assets in the system a system dependence graph G is created. Changes to the system are then analyzed to filter any impacts on the interfaces. This is done by graph traversal starting from the seed/s of change/s to the interfaces on the inverse (G^{-1}) of the graph G. The interfaces touched by the graph traversal will form the set of affected interfaces. The system tests associated with the interfaces are filtered as part of test selection (see Figure 5). These tests are run on the updated system to check for compliance in a test environment. Prior to the filtering, the test cases must be updated to reflect the change. Once all the tests pass, the graph G is updated according to the changes made to the assets.

Addition of code to existing programs can be dealt as modification of that asset, and the graph traversals can be started from the changed asset as a seed. However, new assets created (or new attributes for tables or new variables for copybooks), does not affect the system unless they are used. E.g. If a new program A is added, it will not affect the system unless it is called by some other programs, JCLs or PROCs. The assets where the new additions are used form the seeds of change for graph traversal and the affected interfaces are then filtered.

Like the firewall technique in [24], ID-RTS filters out test cases based on the modules that interact with the change. However, instead of filtering out integration tests on the first level modules that interact with the change, ID-RTS filters the system test cases at the entry points to the system that directly or transitively interact with the change. With mainframes, at least with the system in test, the unit or integration test cases are not properly defined as these test techniques were not widespread at the beginning of mainframe development. Moreover, programs cannot be tested standalone without a batch job submitting them. Today, mainframes are being tested against the designed system tests which mandate the behavior of the system. If tests and techniques enable fine grained tests (like unit or integration), ID-RTS can be extended to analyze impact on the assets that have tests available for and select test cases accordingly.

Chapter 4: Analysis

Harrold et al. in [5] have given analyzed metrics to gauge the effectiveness of any RTS techniques. They have defined a test t to be modification revealing if the output of the case differ in the original (P) and the modified system (P′). The metrics identified were Inclusiveness, Precision, Efficiency and Generality.

Inclusiveness

Inclusiveness measures the extent to which an RTS technique selects modification-revealing tests from the initial regression test suite T. Let us consider an initial test suite T containing n modification-revealing test cases. If an RTS technique M selects m of these test cases, the inclusiveness of the RTS technique M with respect to P, P′ and T is expressed as (m/n) ∗ 100.

A safe RTS technique selects all those test cases from the initial test suite that are modification-revealing. Therefore, an RTS technique is said to be safe, iff it is 100% inclusive.

Harrold et al. in [5] have also defined a test to be modification-traversing. A test t is modification-traversing if it executes all the modified and deleted code, irrespective of the output given. A set of modification-traversing tests is the superset of modification-revealing tests (see Figure 6). The ID-RTS approach filters out test cases that traverse more than the modified assets, giving 100% inclusiveness and safety.

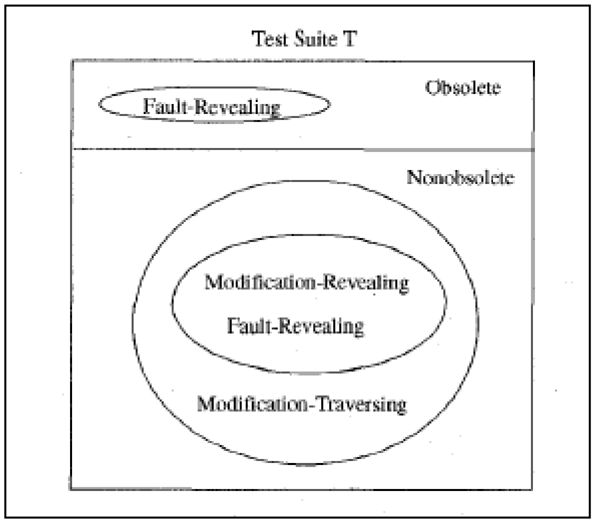

Figure 6. Modification-revealing and modification-traversing tests relation

The primary aim of ID-RTS is to be safe, as the approach plans to replace the existing retest-all techniques. The rationale behind any retest-all technique is to gain confidence on the compliance of the system behavior. ID-RTS works on this requirement and identifies the assets that are impacted by the change and includes all the tests associated with the impacted assets. Like the firewall technique [24], ID-RTS needs the system tests to be reliable. However, for mainframes, they are reliable because the same tests are used to guarantee compliance with the existing retest-all approach.

The nature of the retest all system tests however, is not designed to unit test any particular asset within the system. There are interfaces to a system in the form of online transactions and batch jobs that call other programs inside the system and use other assets, the tests that run after any change are run on the interfaces and poll for an output. ID-RTS is aimed at identifying the interfaces that are ‘infected’ by the change in any asset within the system, and testing all the test cases associated with the affected interfaces.

Precision

Precision measures the extent to which an RTS algorithm ignores test cases that are non-modification-revealing. Suppose T contains n tests that are non-modification-revealing for P and P′ and suppose M omits m of these tests. The precision of M relative to P, P′ and T is given by the expression $(m/n) * 100$ or 100% if $n = 0$.

As explained earlier, in any system test the test suite contains only the tests that run at the entry points (interfaces) of the system. In evaluating the precision of ID-RTS, there would be no unit tests in T. Even with that said, due to the coarse inter-procedural level filtering that ID-RTS employs, it is not precise. Also, Rothermel in [25], has shown that the savings acquired from fine grained intra-procedural techniques may not justify the costs associated. With the system under test, which is a typical mainframe system, the large number of programs would have even higher costs for using intra-procedural techniques.

Also, precision varies with the modularity of the system. If the system is designed such that there exists more number of small programs, copybooks and transactions, inter-procedural test selection can also be fine grained. As the impact is flooded from the seed, only programs that would indeed by affected by the change are filtered, eventually selecting only the modification-revealing interfaces. Coding best practices can be established to have modularity in the system, but the costs of changing the existing code base vs. the cost of testing extra test cases is highly debatable.

Efficiency

Efficiency measures the time and space requirements of the RTS algorithm. ID-RTS finds the impact on the interfaces of the system using graph traversals on the inverse of the system dependence graph G. For a strongly connected graph such as G, the time complexity for graph traversal is of the order of the number of edges O (E). As assets such as copybooks and database tables can have multiple nodes in the graph, graph traversal on the system dependence graph runs in pseudo-polynomial time O ((NP) ^{2}), where N is the average number of nodes per asset P. However not all types of assets are interconnected; DB entries, JCLs (excluding PROCs), files and copybooks (assuming there is no nesting) would not have any dependencies within their asset type and hence no edges amongst them in the graph exist. Let P denote programs and PROCs, V denote sum of all data variables in all copybooks (a record structure counts as a single variable), A denote all database table attributes, F denote files, J denote JCLs and screens, the complexity can be given as $O(P^2 + VP + AP + JP + FP)$ (see Figure 7). The space complexity is also of the same order.

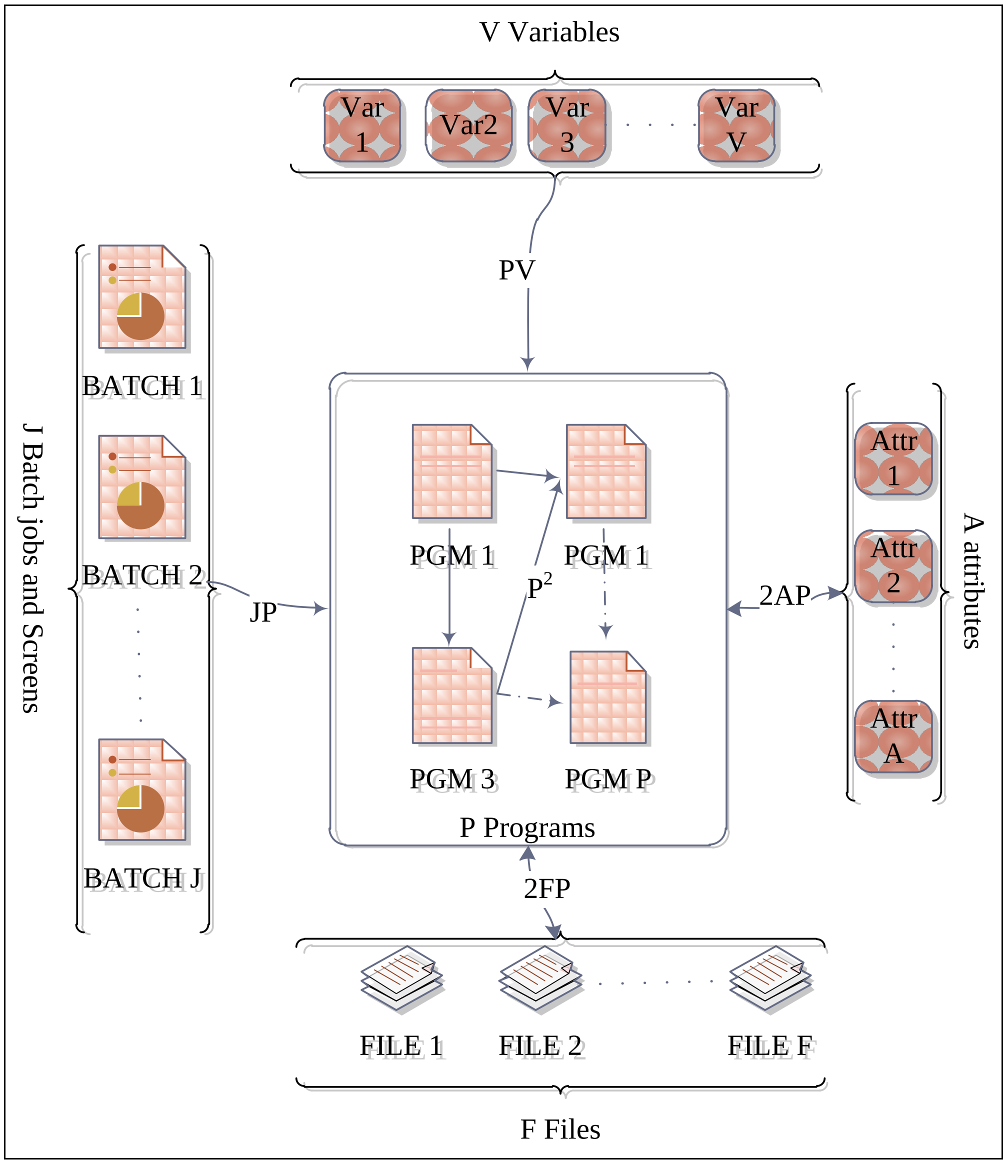

Figure 7. The analysis of complexity of impact analysis

Generality

Generality is the ability of the technique to work in various situations. ID-RTS was designed to work only in the mainframe environment and cannot be generically applied to other systems as is. However, it provides a basis for designing a framework for other business-oriented systems. Business-oriented systems similar to COBOL system have an integrated software solution that incorporates the key business functions of the organization. They consist of the frontend screens/transactions and batch jobs which act as the frontend, a series of programs to process the request and change the database tables or files in the backend and populate the result on the screen or return the code of the execution to the batch job.

Chapter 5: Experimental Setup

To calculate the savings from using ID-RTS, a real world mainframe application was analyzed for dependencies and impacts of changes. IBM's Rational Asset Analyzer (RAA), a static analysis tool available for mainframes was used. RAA statically analyzes all the assets imported into the RAA server and establishes dependencies among them as described earlier. From the dependencies established, it also analyzes impact of a change in source programs, data elements in copybooks, other files and DB2 DB.

The test environment for the application is a dedicated test environment with real world data replicated from the production system. The testing on mainframes has been expensive also because of the high costs of clock cycles used by testing processes on mainframes. These costs can be reduced by using virtual System z environments such as Rational Development and Test (RDnT) for System z. Apart from saving the clock cycles by running virtually on a desktop or server, it also adds to the system availability for mainframe application development and testing.

The system test cases for the application test the interfaces of the system. For batch jobs the output in DB tables or files is monitored for expected output. For online transactions the output is monitored on the screen itself, optionally the contents of the file or DB can be monitored. As only the interfaces are tested in systems tests, for the experiment, the impacted interfaces were filtered for each change, instead of filtering the actual test cases. This also assumes that tests are evenly distributed across the interfaces. Also, as CICS screens and transactions were not imported into RAA for the application under test, only batch jobs were considered. But this does not affect the generality of the research, for RAA analyzes impact of a change on all assets of the system including screens and transactions.

The nature of the applications under test is similar to the depiction in Figure 4. App A has around 6,707 assets in all containing 2,287 source programs, 1,553 JCL batch jobs and 1,823 copybooks. The rest form the control and data files. This excludes the CICS transactions and screens that the application might use. As this application does not use any database, we tested another application, App B, for impact of database changes. App B has 48,210 assets in all with 109 DB2 database tables and 5,393 JCL batch jobs.

Chapter 6: Results

We ran the impact analysis of last two production changes of each asset type, in order to gauge the efficiency of ID-RTS in real world scenarios. We then filtered out the affected batch jobs from the impact analysis report and calculated the mean impact by type as shown in Table 1 and Figure 8.

Asset Type	**Mean Impact by Type (%)**
JCLs	0.06
Database	0.93
Files	43.59
Copybooks	65.54
Source Programs	65.95

Table 1 Mean Impact by type (last 2 changes of each)

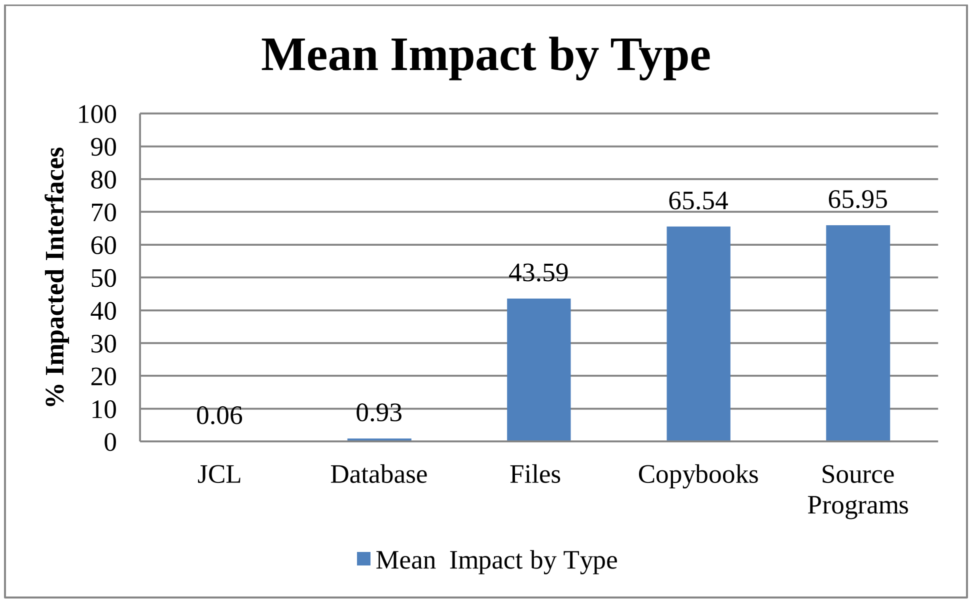

Figure 8. Mean Impact by Type as % of interfaces impacted

The order of the impacts amongst the asset types was found to be as expected, with the exception of copybooks and source programs. Contrary to our expectations, the impact of the copybook changes was found to be less than that of source programs, however with a close difference. This could be attributed to the nature of assets that were changed; the source programs changed were called by more assets than the programs that used the changed copybooks. This outcome is highly peculiar and moreover, we expected the difference in impacts of changes in copybooks and source programs to be minimal, which was exemplified.

Between files and databases, the outcome of the order was in compliance with the expectations. The order could vary from application to application. For applications under test, the DB2 database tables were introduced in App B much later than the usage of files in App A, accounting to files being used significantly more than the databases. The impact of change is proportional.

The time required for impact analysis varied by asset type, with copybooks and source programs taking the longest, approximately 2 hours on an average, files took 1 hour and 10 minutes and database tables took 10 minutes. JCL impact analysis is not supported in RAA as there are no dependencies on them. For the purpose of the experiment we assumed the impact of a JCL batch job change on interfaces as 1 to account for the same JCL changed. RAA's impact analysis gives a thorough report of all impacted assets, not just the interfaces. This analysis adds to the total time required for impact analysis. Tools that would solely report the impacted interfaces would have significant time savings.

To calculate the savings from ID-RTS, we created a three mix of asset changes (in %) in an iteration. Table 2 describes the three mixes along with the impacted interfaces; the savings from using ID-RTS is shown in the Figure 9. As seen from the results, the impacted interfaces and hence savings change with the combination changes. More programs and copybooks reduce the savings achieved, as these assets impacted the interfaces more than the rest of the assets.

Mixes in %	**Comb 1**	**Comb 2**	**Comb 3**
Source Programs	20	40	50
Files	20	10	5
Copybooks	20	30	30
JCLs/Screens	20	10	10
Database	20	10	5
Total Interfaces per change	100	100	100
Impacted interfaces per change	35.214	50.5	54.869
%Savings from ID-RTS	64.786	49.5	45.131

Table 2. Three asset change mixes and their impacts

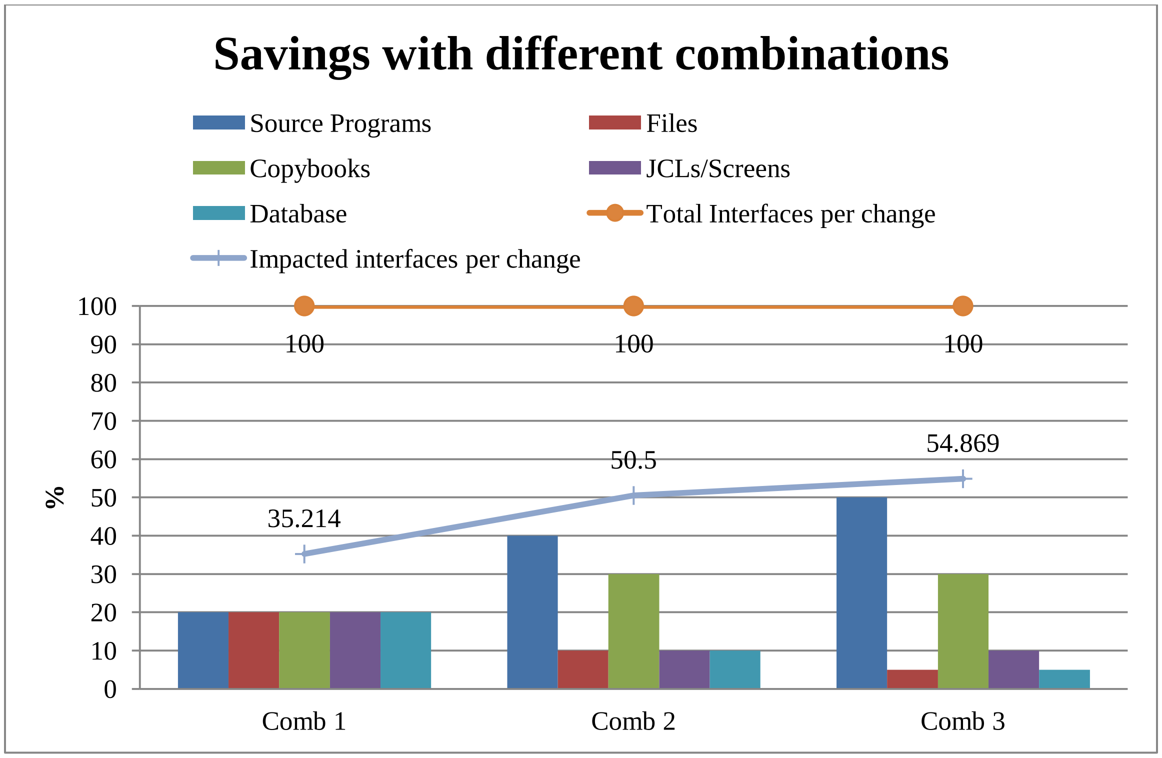

Figure 9. Three asset change combinations (from Table 2) and their impacts

To calculate the actual savings from ID-RTS, we first recorded changes for the period of March 2012 to February 2013 by asset type. The findings are tabulated in Table 3 (also Figure 10 and Figure 11). We then estimated impacts on App A by taking weighted mean of MIT with respect to the frequency of change. The results are shown in Table 4 and Figure 12. As, in all iterations programs were changed the most, the impacted interfaces were around 65% of the total 1,554 interfaces. As per ID-RTS, these are the interfaces to be tested while the retest-all technique tests all of them. For the entire period the average impacted interfaces per change were 1,023.3 (65.85% of the 1,554 interfaces). Thus the ID-RTS technique can save approximately **34%** of testing efforts.

This result can vary by the nature of dependencies within the application and the types of changes that are done. If for App A, there are more changes to DB, files and JCLs than copybooks and programs the savings would be proportionally more. Also, the impact might change in assets of the same type, depending on the nature of dependencies. E.g. Program A has more dependencies than program B, the impact of change of A would be proportionally more. This experiment was conducted to find out if the impact of actual production changes span to a subset of the system and calculate savings ID-RTS can give. Also, a learning phase can be planned, where the impacts of production changes are monitored to gauge the inter-dependency among the assets of the system. If all the changes impact significant percentage of interfaces, the retest-all technique can be employed for that system, saving the time required for impact analysis.

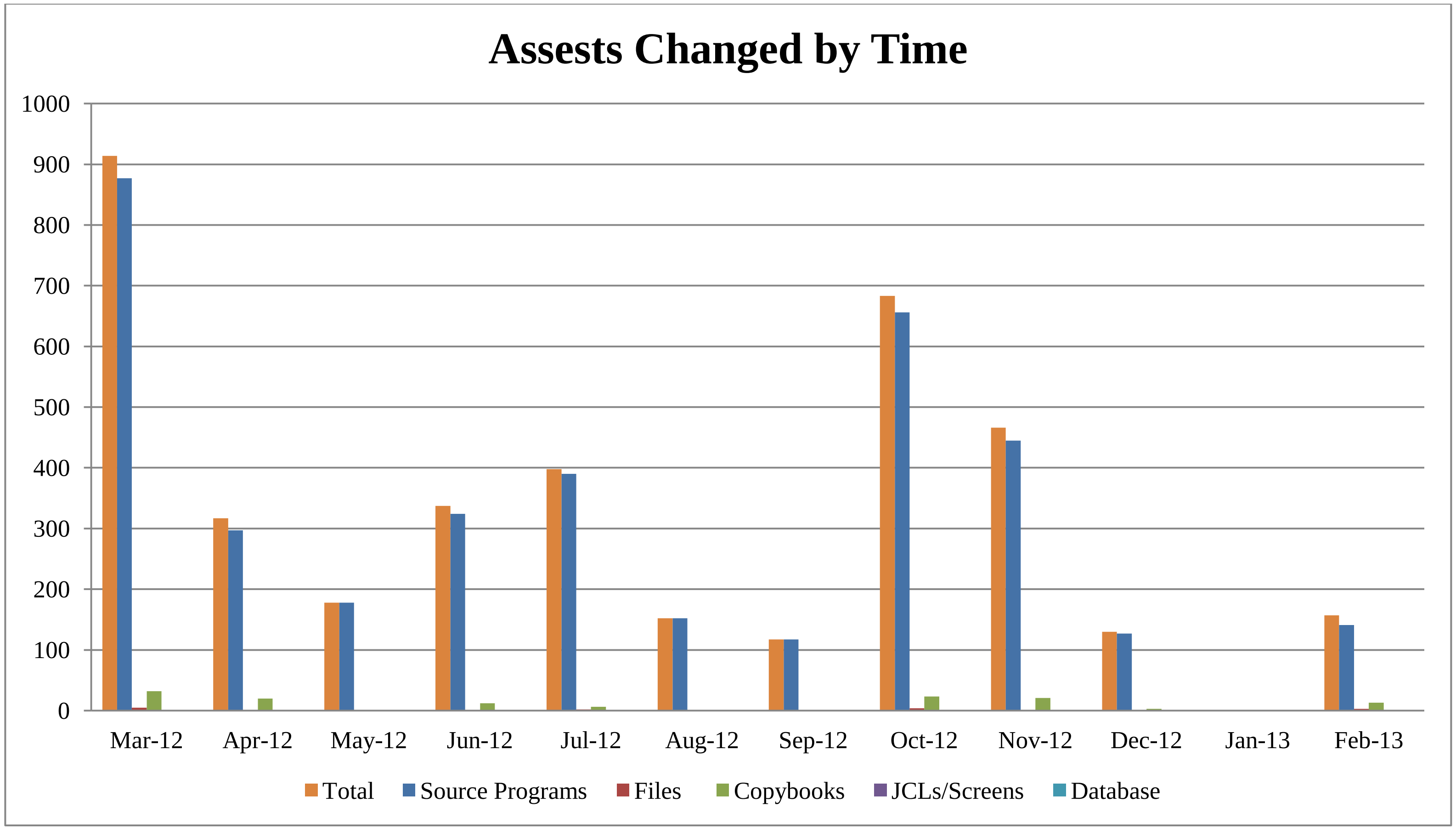

Figure 10. Assets changed over a period of one year

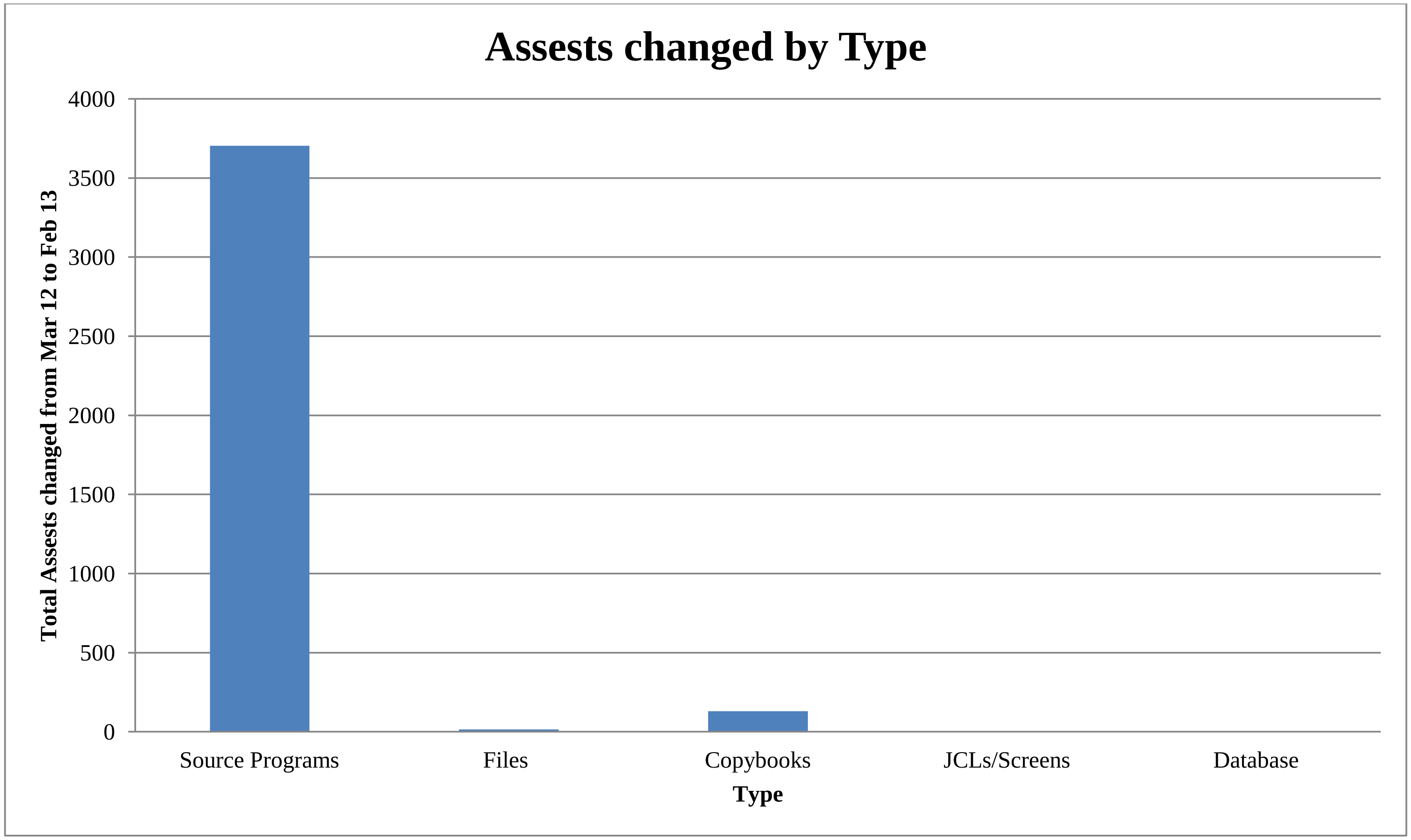

Figure 11. Assets changed by type over a period of one year

Asset Type\Month	Mar-12	Apr-12	May-12	Jun-12	Jul-12	Aug-12	Sep-12	Oct-12	Nov-12	Dec-12	Feb-13	Total
Source Programs	877	297	178	324	390	152	117	656	445	127	141	3704
Files	5	0	0	1	2	0	0	4	0	0	3	15
Copybooks	32	20	0	12	6	0	0	23	21	3	13	130
JCLs/Screens	0	0	0	0	0	0	0	0	0	0	0	0
Database	0	0	0	0	0	0	0	0	0	0	0	0
Total	914	317	178	337	398	152	117	683	466	130	157	3849

Table 3. Assets changed over a period of one year

Tested Interfaces per change	Mar-12	Apr-12	May-12	Jun-12	Jul-12	Aug-12	Sep-12	Oct-12	Nov-12	Dec-12	Feb-13	Total Average
ID-RTS	1022.7	1024.5	1024.9	1023.6	1023.0	1024.9	1024.9	1022.6	1024.6	1024.7	1017.7	1023.3
Retest-all	1554	1554	1554	1554	1554	1554	1554	1554	1554	1554	1554	1554

Table 4. Tested Interfaces by ID-RTS vs. retest all over a period of one year

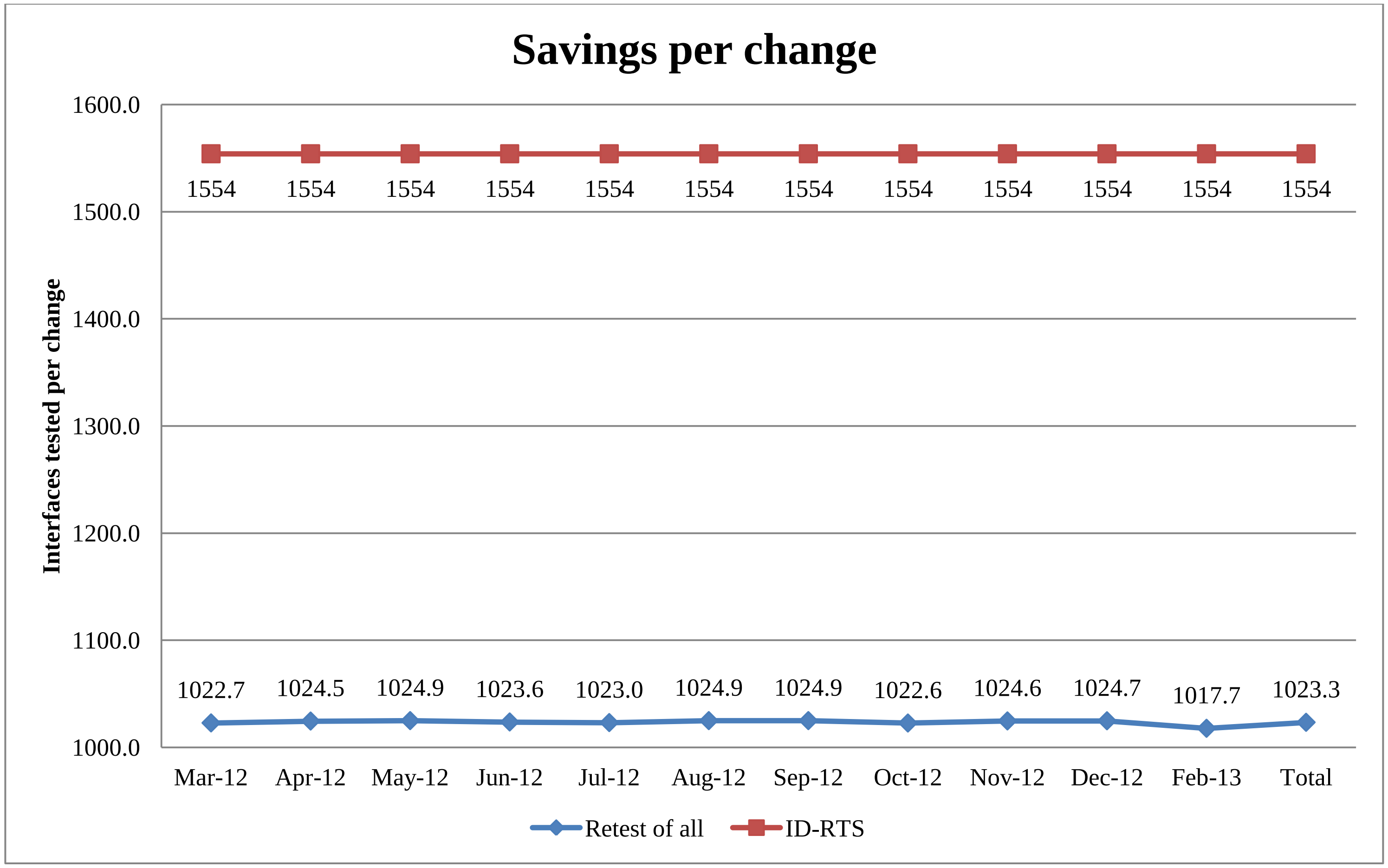

Figure 12. Interfaces tested by ID-RTS vs. retest-all over a period of one year

Chapter 7: Limitations

ID-RTS is a static code analysis based RTS technique. It cannot establish dynamic dependencies amongst assets. Thus, if a dynamic call is made by a source program to another source program using its name populated in a variable at run-time, the dependency amongst them cannot be established by static code analysis. However, such practices also affect the thorough analysis of program flow in the analysis tools. Hence, coding best practices must be established to prevent dynamic calls.

Also, the technique relies heavily on performance of the analysis tools. IBM's RAA as used in the experiment requires considerable amount of time to import assets into its DB, tokenize them and establish dependencies (E.g. The importing of application with 48,210 assets required 60 man hours of work). The time required is proportional to the number of assets in the system. However, the importing has to be done only once in the system lifetime; the changed assets can be specifically imported using the incremental import techniques during the system maintenance.

System tests for mainframes were designed to test the system with a black-box approach i.e. the tests are unaware of the code or the changes that were done. ID-RTS replaces system tests with a code-aware technique. The system testers have to be aware of the code changes done to the system, then analyze impacts of changes and run tests accordingly. However, as seen in the results, significant savings can be achieved by analyzing the changed code and its impact, which justifies the use of a code-aware testing technique.

Chapter 8: Conclusion and Future Work

Impact-Driven Regression Test Selection (ID-RTS) - a control flow and data dependence based, intra-procedural regression test selection technique designed for mainframes - can be used to replace the retest-all system test technique. With the exception of dynamic calls in COBOL programs, both data and control dependencies in a system can be drawn out in a system dependence graph using static code analysis.

The savings are achieved by executing only those tests from the system test case pool that test the impacted interfaces of the system. The impact analysis is done by graph traversal on the inverse of the system dependence graph, starting from the asset that was changed and filtering the affected interfaces. The impact of a change spans to a subset of the system, providing significant savings in the test cycle times. Modern analysis tools such as IBM's RAA can be used to establish dependencies among assets of the system and analyze impact of a change.

ID-RTS though not precise, is a safe RTS technique as it filters all test cases that walk through the modification of the system. The safety of the technique also depends on the reliability of the system test cases; and as these tests in the as-is retest-all technique are used to assure compliance, these test cases are reliable.

As future work, ID-RTS and the various studies on the object oriented RTS techniques can be integrated to design the framework for object oriented COBOL. Also, the current framework can be extended to include other business-oriented system such as SAP.

References

[1] R. Pressman. Software Engineering: A Practitioner’s Approach. McGraw-Hill, New York, 2002.

[2] H. Leung and L. White. Insights into regression testing. In Proceedings of the Conference on Software Maintenance, pages 60–69,

[3] Weiderman, Nelson H.; Bergey, John K.; Smith, Dennis B.; & Tilley, Scott R. Approaches to Legacy System Evolution (CMU/SEI-97-TR-014). Pittsburgh, Pa.: Software Engineering.

[4] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. IEEE Transactions on Software Engineering, 22(8):529–551, August 1996.

[5] G. Rothermel, R. H. Untch, and M. J. Harrold, ”Prioritizing test cases for regression testing,” In IEEE Trans. on Software Eng. vol. 27, No. 10, pp. 929–948, October 2001.

[6] M. J. Harrold, R. Gupta, and M. L. Soffa, ”A methodology for controlling the size of test suite,” In ACM Trans. on Software Eng. and Methodology (TOSEM), NY USA, pp. 270–285, 1993.

[7] M. Hennessy, and J. F. Power, ”An analysis of rule coverage as a criterion in generating minimal test suites for grammar based software,” In Proceedings of

the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE’05), Long Beach, CA, USA, pp. 104–113, November 2005.

[8] A. Kandel, P. Saraph, and M. Last, “Test cases generation and reduction by automated input-output analysis,” In Proceedings of 2003 IEEE International Conference on Systems, Man and Cybernetics (ICSMC’3), Washington, D.C., October, 2003.

[9] B. Vaysburg, L. H. Tahat, and B. Korel, “Dependence analysis in reduction of requirement based test suites,” In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’02), Roma Italy, pp. 107–111, 2002.

[10] J. A. Jones, and M. J. Harrold, ”Test-suite reduction and prioritization for modified condition/decision coverage,” In IEEE Trans. on Software Engineering (TSE’03), Vol. 29, No. 3, pp. 195–209, March 2003.

[11] D. Jeffrey, and N. Gupta, ”Test suite reduction with selective redundancy,” In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM’05), Budapest, Hungary, pp 549–558, September 2005.

[12] S. R. Khan, I. Rahman, S. R. Malik “The Impact of Test Case Reduction and Prioritization on Software Testing Effectiveness“ in International Conference on Emerging Technologies 2009.

[13] G. Rothermel and M. Harrold. Selecting tests and identifying test coverage requirements for modified software. In Proceedings of the International Symposium on Software Testing and Analysis, August 1994.

[14] Y. Chen, R. Probert, and D. Sims. Specification based regression test selection with risk analysis. In CASCON ’02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research, 2002.

[15] P. Chittimalli and M. Harrold. Regression test selection on system requirements. In ISEC ’08: Proceedings of the 1st conference on India software engineering conference, 2008.

[16] S. Biswas, R. Mall, M. Satpathy and S. Sukumaran. Regression Test Selection Techniques: A Survey. Informatica. 35(3):289-321, October 2011.

[17] M. Harrold and M. Soffa. An incremental approach o unit testing during maintenance. In Proceedings of the International Conference on Software Maintenance, pages 362–367, October 1988.

[18] A. Taha, S. Thebaut, and S. Liu. An approach tosoftware fault localization and revalidation based on incremental data flow analysis. In Proceedings of the 13th Annual International Computer Software and Applications Conference, pages 527–534, September 1989.

[19] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In Proceedings of the 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS’ 97), pages 3–21, May 1997.

[20] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, July 1987.

[21] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In Proceedings of the Conference on Software Maintenance, pages 282–290, November 1992.

[22] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology, 6(2):173–210, April 1997.

[23] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In IEEE International Conference on Software Maintenance, pages 348–357, 1993.

[24] H. Leung and L. White. A study of integration testing and software regression at the integration level. In Proceedings of the Conference on Software Maintenance, pages 290–300, November 1990.

[25] G. Rothermel, "Efficient, Effective Regression Testing Using Safe Test Selection Techniques," PhD dissertation, Clemson Univ., May 1996.

[26] Norman Wilde. Understanding Program Dependencies. SEI-CM. August 1990.

[27] J. Highsmith and M. Fowler. The Agile Manifesto. Software Development Magazine, 9(8), 29–30, 2001.

[28] Keren, A.; Hazzan, O.; Dubinsky, Y. Agile software testing in a large-scale project. In Software, IEEE, July-Aug. 2006.