Impact-driven Regression Test Selection for Mainframes

Abhishek Dharmapurikar, Benjamin J. R. Wierwille, Jayashree Ramanthan and Rajiv Ramnath

Ohio State University, Computer Science and Engineering, Columbus, Ohio, U.S.A.

Abstract. Software testing is particularly expensive in the case of legacy systems such as mainframes. Critical to many large enterprises, these systems are perpetually in maintenance where even small changes usually lead to an end-to-end regression test. This is called the "retest-all" approach and is done to ensure confidence in the functioning of the system, but this approach is impractical primarily due to resource needs and user stories generated within the agile system that require rapid changes to the system. This research is aimed at reducing the required regression testing and its costs associated with the system. The improvements are achieved by identifying only those tests needed by assets changed and others that are 'impacted'. The impact analysis leverages the availability of modern static code analysis tools and dedicated test environments for mainframes. By using our proposed impact technique on a real-world mainframe application, the test savings can be about 34%.

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. As for the mainframe systems that have been modernizing to keep up, reducing the testing costs would immediately benefit.

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

Dharmapurikar A., J. R. Wierwille B., Ramanthan J. and Ramnath R..
Impact-driven Regression Test Selection for Mainframes .
DOI: 10.5220/0004601300550066
In *Proceedings of the 1st International Workshop in Software Evolution and Modernization* (SEM-2013), pages 55-66
ISBN: 978-989-8565-66-2

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.

There have been many studies to reduce the cost associated with regression testing. Three techniques, test case reduction, test case prioritization and regression test selection are most prevalent. 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. Also these techniques have been proven to reduce the fault detection capacity of the suites [2]. 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. Our technique, Impact-Driven Regression Test Selection (ID-RTS) builds on this idea and aims at reducing test costs for mainframe systems and is proposed as a replacement for the retest-all regression tests. It is expected to be safe (i.e. the tests selected should reveal all the modifications done to the system), save costs and increase system availability.

The rest of the paper discusses RTS techniques and ID-RTS. Section 2 discusses the different regression test selection techniques. Section 3 of this paper analyzes the structures of the assets and the dependencies among them. Section 4 describes test case selection through impact analysis. Section 5 analyzes the efficiency of the RTS technique using standardized metrics. Section 6 describes an experiment carried out to gauge the savings from using this technique. Section 7 analyzes the results from the experiment and extrapolates savings for a year for an actual enterprise using real data for changes in that period.

2 Regression Test Selection Techniques

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′.*

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 [17], [18] are not safe due to their inability to detect the effect of program modifications that do not cause changes to the dataflow information. Also, they do not consider control dependencies among program elements for selecting test cases. Hence, these techniques are unsafe [16], [4].

Control flow techniques [21], [22] have been proven safe and the graph walk approach suggested in [22] is the most precise work for procedural languages [16] and most widely used control flow technique [27]. However they do not include non-code based components of the system such as DB and files.

Dependence based RTS techniques [19], [20] were proved to be unsafe as they omit tests that reveal deletions of components or code [4].

Differencing technique [19] is safe, but 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] are precise, but 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. 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 techniques which 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

1. Comprehensively representing the inter-asset dependencies in a dependence graph by static code analysis.
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.

3 Mainframe Asset 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). It originally consisted of source programs, copybooks, JCLs, PROC files and record oriented files. 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 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.*

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, in order to resolve the references. To establish ***source – copybook*** dependencies among the copybooks and programs the definition-usage model proposed by the dataflow methods [17], [18] is used. To minimize the impact of change in copybooks on programs that would not use all the variables inside the copybook, the dependencies are established at the data structure level, i.e. instead of the entire copybook, the data structures inside it would 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 1).

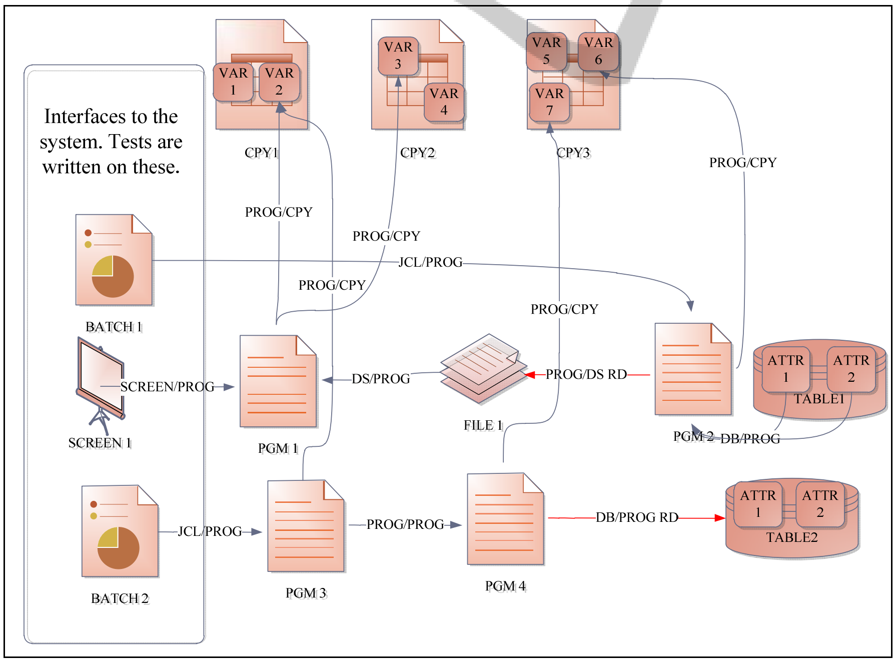

Fig. 1. The Dependencies amongst Various Assets in a Mainframe System. The Batch Jobs and Online Screens Form the Interfaces to the System. the System Tests Test These Assets.

Source – source dependencies exist when 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 1).

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.

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 1). 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 1). 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 Section 4.

Stored procedures can be treated as programs that have SQL statements to manipulate the DB, hence creating a source to DB dependency. Source to source dependencies would exist between the calling program and the stored procedures.

JCLs are used to run COBOL programs in batch mode, creating a ***JCL – source*** dependency. JCLs are dependent on a program if in any step they are executing that program (E.g. BATCH1 – PGM2 dependency in Figure 1). 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.

Screens are like programs, but can be executed online in a transaction. They can call other programs creating a ***screen – source*** dependency (E.g. SCREEN1 - PGM1 dependency in Figure 1).

Copybook - copybook dependencies exits when COPY statements are used within copybooks. 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.

To any mainframe system, batch jobs and online transactions on screens act as interfaces. These batch jobs and screens 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 1).

4 Filtering Interfaces and Tests

With the static code 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 2). 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.

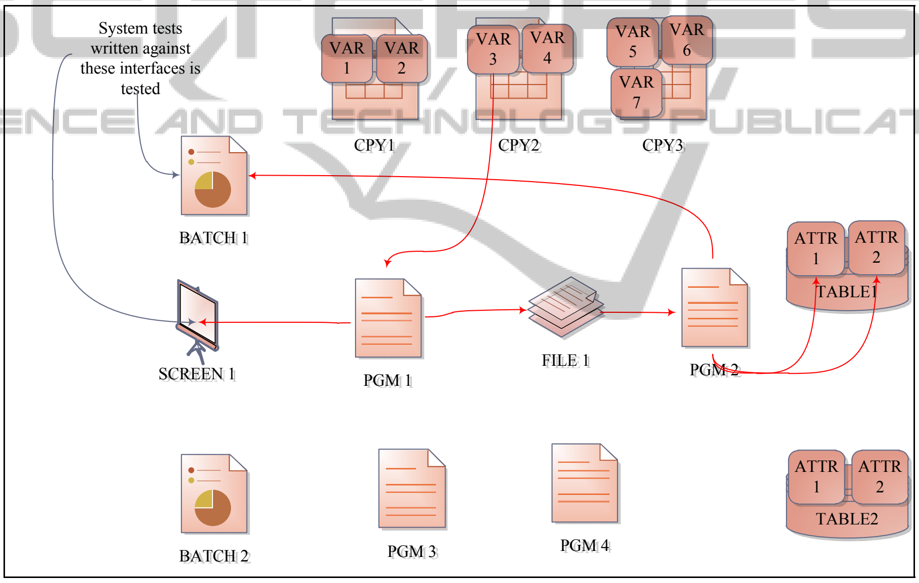

Fig. 2. A depiction of impact of change. The VAR3 data structure variables changed, whose impact is carried over the interfaces through graph traversal on the inverse graph G in Figure 1.

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.

5 Analysis

Harrold et al. in [5] have given analysis 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 differs in the original (P) and the modified system (P´). The metrics identified were *Inclusiveness, Precision, Efficiency* and *Generality*.

Inclusiveness measures the extent to which an RTS technique selects modification-revealing tests from the initial regression test suite T. 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. The ID-RTS approach filters out test cases that traverse more than the modified assets, giving 100% inclusiveness and safety. Like the firewall technique [24], ID-RTS needs the system tests to be reliable for it to be safe. And, for mainframes, they are reliable as the same tests are used to guarantee compliance with the existing re-test-all approach.

Precision measures the extent to which an RTS algorithm ignores test cases that are non-modification-revealing. Due to the coarse inter-procedural level filtering that ID-RTS employs, it is not precise. Rothermel in [25] has shown that the savings acquired from fine grained intra-procedural techniques may not justify the costs associated. For 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 are more number of small programs, copybooks and transactions, inter-procedural techniques can be more precise.

Efficiency measures the time and space requirements of the RTS algorithm. 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 $O(P^2 + VP + AP + JP + FP)$. The space complexity is also of the same order.

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 languages.

6 Experiment Setup

To calculate the savings from using ID-RTS, we conducted a small experiment to analyze a real world mainframe application 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. As a contribution, this is one of the few studies to have been actually designed and tested in an enterprise environment [27].

As only the interfaces are tested in systems tests for an application, 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 1. App A has around 6,707 assets in all containing 2,287 source programs, 1,554 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.

To calculate the savings from implementing ID-RTS we have taken a two step approach,

1) We identify the impact of last 2 changes in each asset type in set A (namely JCLs, programs, copybooks, files and DB tables) on the interfaces of the system. From the data collected, the mean impact by asset type (MIT) is calculated as % of the 1,554 (5,393 for App B) interfaces affected. As the impact of an asset change depends on the number of other assets dependent (directly or transitively) on it, we expect the number of impacted interfaces per asset type to be in the decreasing order for copybooks, programs, files, DB and JCLs for this particular system.
2) To calculate the actual savings, we first calculate the frequency of occurrence of changes by asset type (FT) as % in an agile iteration and then extrapolate impacts using the weighted mean of MIT with respect to FT for each asset type in A. Weighted mean impacted interfaces (WMI) is calculated as

$$WMI = \sum_{\in A} MIT \times FT$$

This would give us the % interfaces we need to test for each change in an iteration. **Savings in % would be 100 – WMI**.

7 Results

We ran the impact analysis of last 2 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. As we found that the other RTS techniques as described in section 2 were infeasible or not safe for mainframes, we compare ID-RTS with the existing retest-all technique.

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.

Table 1. Mean Impact by Type for last 2 changes.

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

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 adds to the total time required for impact analysis. Tools that would solely report the impacted interfaces would have significant time savings.

To calculate 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 2. We then extrapolated impacts on App A by taking weighted mean of MIT with respect to the frequency of change. The results are shown in ID-RTS column of table 2. 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 for each change 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 estimate savings for ID-RTS. Hence we also limited the test to last 2 changes to each asset type. 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.

Table 2. The changes done to the App A by type for the period of March 12 - February 2013. Column ID-RTS gives the extrapolated impacted interfaces using data from Table 1 and Re-test-all gives the total interfaces in App A to be tested with that technique.

Period	Source Pro-grams	Files	Copy-books	JCLs/ Screens	DB	To-tal	ID-RTS	Re-test -all
Mar-12	877	5	32	0	0	914	1022.7	1554
Apr-12	297	0	20	0	0	317	1024.5	1554
May-12	178	0	0	0	0	178	1024.9	1554
Jun-12	324	1	12	0	0	337	1023.6	1554
Jul-12	390	2	6	0	0	398	1023	1554
Aug-12	152	0	0	0	0	152	1024.9	1554
Sep-12	117	0	0	0	0	117	1024.9	1554
Oct-12	656	4	23	0	0	683	1022.6	1554
Nov-12	445	0	21	0	0	466	1024.6	1554
Dec-12	127	0	3	0	0	130	1024.7	1554
Feb-13	141	3	13	0	0	157	1017.7	1554
Total	3704	15	130	0	0	3849	1023.3	1554

8 Conclusion and Future Work

The retest-all system test technique which tests all components of the system for regression test can be replaced by our proposed Impact-Driven RTS. Modern analysis tools such as IBM's RAA can be used to draw dependencies among assets of the system and analyze impact of a change. The impact of a change spans to a subset of the system, providing significant savings in the test cycle times which reduces associated costs and increases system availability.

As future work, 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 systems such as SAP-ABAP. Similar to mainframes these systems use batch jobs, online transactions, DBs and files within a single system boundary.

References

1. R. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, 2002.
2. Leung H., White, L.: Insights into regression testing. In Proceedings of the Conference on Software Maintenance, pages 60–69, 1989.
3. Weiderman, Nelson H., Bergey, John K., Smith, Dennis B., & Tilley, Scott R.: Approaches to Legacy System Evolution. In (CMU/SEI-97-TR-014) Pittsburgh, Pa. Software Engineering Institute, Carnegie Mellon University, 1997.
4. G. Rothermel and M. Harrold. Analyzing regression test selection techniques. In IEEE Transactions on Software Engineering, pages 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, pages. 929–948, Oct. 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, pages. 270–285, 1993.
7. M. Hennessy, 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, pages 104–113, November 2005.
8. 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., pages 768-773 vol.1 October, 2003.
9. 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, pages 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, pages 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, pages 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, pages 416- 421, October 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, pages 169-184 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, page 1, 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, pages 87-96, February 2008.
16. S. Biswas, R. Mall, M. Satpathy and S. Sukumaran. Regression Test Selection Techniques: A Survey. In 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. 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, pages 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. In SEI-CM. August 1990.
27. S. Yoo, M. Harman: Regression testing minimization, selection and prioritization: a survey. In Journal of Software Testing, Verification and Reliability, Volume 22, Issue 2, pages 67–120, March 2012.