A Dynamic Approach to Visualize the Impacts for Support Selective Testing on Regression Testing

Omid Pourgalehdari
School of Electrical and Electronic Engineering,
University Sains Malaysia, Engineering Campus 14300 NibongTebal, SeberangPerai Selatan, Pulau Pinang, Malaysia

ABSTRACT
Keeping up with the advancement in hardware technology, the size and complexity of software systems are increasing at a rapid rate, thus, making them difficult to maintain, expand, and evolve. To alleviate such difficulties, change impact analysis (CIA) and its implementations has been the subject of research for several years. Generally, CIA facilitates regression testing. Specifically, CIA helps to estimate the potential consequences of a software change, including the affected module(s) and their data dependencies, re-testing needs, as well as the required resource planning. Historically, many CIA implementations use static analysis and traditional text-based impact reporting. Although useful, static based CIA implementations often cited as time- and effort-intensive (e.g. requiring extensive documentation/design search). Dynamic slicing is an option to address the aforementioned issues. However, the volume of analyzable data potentially impedes understanding. Visualization can be a good leverage for improving analyzability and understanding of impact analysis from dynamic slicing.In line with such a prospect, this paper offers a dynamic approach to visualize the impacts for support selective testing on regression testing.

Keywords: dynamic change impact analysis, regression testing

1. INTRODUCTION
Gradually, software are replacing most other products (e.g., mechanical products) whenever possible – due to its customizability and changeability[1]. Nevertheless, in order to remain in use, software needs to evolve in line with technological advancement and user needs. Over the years, software systems are grown extremely in terms of size and functionality. It is now common to have commercial software with more than a million lines of codes. Such a significant growth has a strong influence as far as evolution is concerned. Evolution is an expensive task [2, 3], costing an average of two to four times the development costs. In the most part, evolution costs can be associated with regression testing.

In order to reduce regression testing costs and avoid unwanted side effects (i.e., or ripple effects) resulting from evolution, there is a need to estimate the impact of change[4]. Change impact analysis (CIA) and its implementation have often been sought for to estimate the change including the affected module(s) and their data dependencies, re-testing needs, as well as the required resource planning [5].
Study on existing CIAsis dedicating that most are using static tracing to address the impacts [6]. Orso et al [7] are indicated that obtaining the data via static tracing is reliable. However, concerning implementation, current static tracing practices have some specific problems/requirements (e.g., useless trace artifacts, unavoidable failures, too many up front activities and the necessity of comprehensive documentation). These problems/requirements are among of some reasons that making the static traceability intensive.

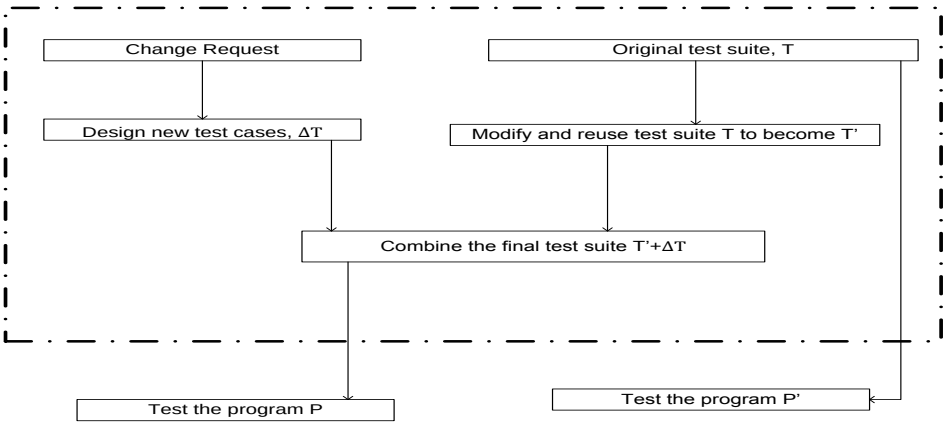
Agrawal & Horgan [8], Korel & J. Rilling [9] Ryder and Tip [10] and Chao et al. [11], believed, unlike static analysis, the dynamic analysis offers a reasonable price, size and maintainer's effort. More often, in dynamic analysis requires placing a track to all functions/modules of a system for trace its execution. The tracking makes the CIA able to; not only address the dynamic impacts in terms of (i.e., loading/unloading of dynamic impacts) but also has good advantages to support the static dependencies. They are among of some reasons that make dynamic analysis as an attractive option. However, in dynamic tracing increasing the number of analysable data still, make the analysis stage complex and impede the understanding. Enhancing and improvement the reporting can be a key success for address the dynamic traceability issues (i.e., in terms of understanding and complexity).

To improve the reporting the impacts, using visualization can be a good option in order to address dynamic tracing as well as understanding issues. Visualization potentially can provide multiple views for item's interest from different perspectives. For example, system interaction mapping can focus on call sites, depth of call, and dependency analysis.

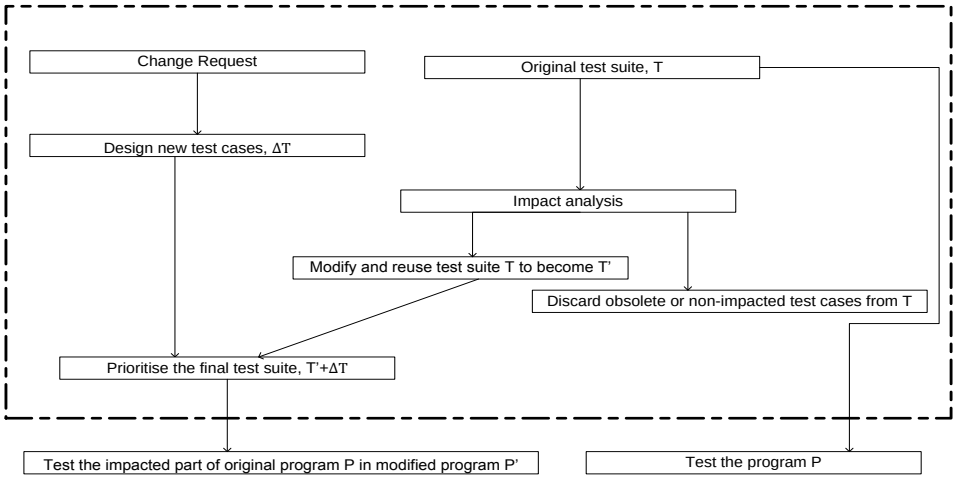
In light of the stated problem statement, (i.e., in terms of static traceability, dynamic traceability as well as reporting stage), this study investigates dynamic approach to visualize the impacts for support selective testing on regression testing.

This paper organized as follows. Section 2 provides a problem definition model, emphasizing the role of CIA for regression testing to address issues related to reworking. Subsequently, section 3 discusses on some related work. Next, in section 4, the case studies and the implemented prototype (J-via) will be defined. Subsequently, in section 5, some obtain results from developed prototype (J-via) for support the approach will be demonstrated. Finally, in section 6, an outlines for conclusion will be given.

2. PROBLEM DEFINITION MODEL
Regression testing can be conducted using two approaches, namely, retest all and selective retesting. Figures 1(a) and 1(b) illustrate the use of both approaches [12].



(a)



(b)

Figure 1: (a) Retest All Approach and (b) Selective Retest Approach

In this model, there is a program P and a set of test cases to validate P, called T. If and only if P' be the modified version of P, then the regression testing aims validate P'. To test P', new test cases derived from the client's new change request, which are calledΔ T, must be developed.

InFigure 1 (a), some modules may need to be changed in some situations while considering T andΔ T. Retesting all approaches can be costly. Additionally, some test cases in T may be outdated because of changes made in P. Therefore, some of the test cases in T may need to be updated.

In Figure 1(b), using the same notation, the impacted modules or functions can be classified into known and unknown classes, which will be discussed in the following subsections.

2.1 Known Impacted Function/Modules

If the impacted function/modules have been identified, Finally, T' can be selected and modified as a subset of T. The chosen test cases for T' are selected from T if and only if they affect P'. ΔT can also be developed based on changes in the client's requests.

The final test suite is (T'+ΔT). In some cases, the execution of the test case within the final test collection of (T'+ΔT) should be prioritized to improve the chances of identifying faults.

2.2 Unknown Impacted Function/Modules

The above discussion relied on known impacted functions or modules. In most cases, however, the designer or the customer does not consider the impacts on software systems or their dependencies. Unknown impacts are a typical concern for software systems, including commercial off-the-shelf (COTS) systems. The COTS part of any software system typically includes functionalities that are not included in the system requirements. Pre-existing functionalities are important because they may have interacted with or are affected by the non-COTS aspects of the software system.

To identify the potential consequences of software evolution, as well as the impacted modules or functions, CIA is necessary [5]. Dynamic CIAs tracks the functions or modules of a system to trace their execution. Tracking allows the CIA to address the loading or unloading of dynamic impacts. The advantages of CIA lie in its capability to support static dependencies.

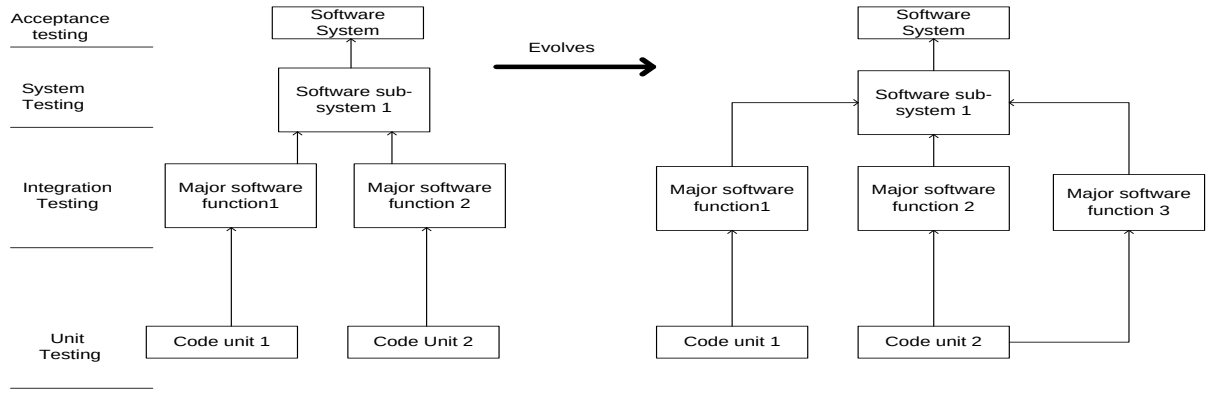


Figure 2: Example for Testing Levels

Figure 2illustrates the testing levels, namely, acceptance, system, integration, and unit testing. Any changes in the system make regression testing essential for code unit 2 at the unit testing level and for major function 2 and major software function 3at the integration level. Given that major functions2 and 3 affect software sub-system 1 and may also affect the overall software system, another round of regression testing at the system level must be performed. Based on the identified dependencies, the test cases that should be used must be determined.

In the earlier example, in terms of known impacted modules/functions (see Figure 2), T' can be selected as a subset of T. T' refers to the test cases that were impacted in P'. The test cases, which don't have any impact on the changed module, can be discarded. ΔT can also be developed based on the client's change requests. Recently tested suites have to be prioritized in this case. The following subsection focuses on the scope of review for this study.

3. RELATED WORKS

Over the past few years, extensive research on CIA has been conducted, and research prototypes have been developed. However, previous studies focused on different aspects of developed CIAs. Thus, different definitions of CIA have been given. For example, Pfleeger[4] defined CIA from a risk management perspective, considering it an "evolution of many risks associated with change, including estimates of the effect on resources, effort, and schedules."Turver and Munro [5] approached CIA as requirement engineers and stated that CIA is the "assessment of change to a source code of a module on other modules of a system to measure complexity as well as change scope determining." They are stating that the impact has to be estimated first before actually implementing any changes to preven such side effects as ripple effects. Regarding the given problem definition model, Zamli et al. [12] discussed CIA from a testing perspective, stating that following the consequence of change is a good option to highlight affected parts to reduce regression testing costs and to prevent of rework. The change impact analysis (CIA) can be done by dependency analysis or traceability analysis. However, this study is focused on dependency analysis and some related works will be given as follows.

3.1 Dependency Analysis

Dependency analysis focuses on impact information captured from the source code. Dependency analysis can be categorized onto two groups of static dependency analysis and dynamic dependency analysis [13].

3.1.1 Static Dependency Analysis

Static dependency analysis involves obtaining data from the source code. The data are then analyzed to assess the impact of a change. This method is categorized into four groups, namely, data dependency, slicing, call graph, and retrieve information, which will be subsequently discussed.

CIA Studies focusing on data dependency analysis have been conducted by M. Lee et al. [14], who identified three different object-oriented dependency graphs that can compute the transitive closure to identify impacted elements. They also proposed a concept based on intra-method data dependency. According to this concept, the impacts on entities located in the method bodies are calculated by the graphs. Change dependencies between methods are calculated by inter-method data dependency graphs, whereas a change impact at the system level is calculated by object-oriented system dependency graphs. The researchers also distinguished four different types of impacts between two related entities. The four impacts are contaminated, which refers to both elements being impacted; clean, which indicates no impact; semi-contaminated, in which the target is not impacted by the source, but with the source having been potentially impacted; and semi-clean, wherein the source is not impacted, but changes exist in the target.

Weights are assigned to the relationships among entities depending on the types of impact relations they have. The total change impact weight is obtained by computing the sum of the weights that have been assigned to the relations between two entities. To calculate the impacts, the total change impact weight is assigned to all three graphs.

JTracker is a CIA approaches that supports regression testing [15] and addresses the impact sets by printing the dependency graphs. JTracker then analyzes the program and builds a dependency database, considering such relations as inheritance, aggregation, and other relations between classes, as dependencies. JTracker marks the classes that the programmer needs to inspect. This particular CIA reuses the results previously gathered by impact calculations and then updates them depending on any of the changes to the program. This process is accomplished by creating a dependency graph of the program based on input-output mapping. The graph is then enhanced using static data flow information from the

method bodies. The results of each analysis process are stored, which allows one to conduct an incremental search for impacted entities, thereby reducing costs.

Zalewski and Schupp[16] proposed conceptual change impact analysis (CCIA), which is based on the principle of pipes and filters. CCIA assesses the impacts of library changes. The first step in CCIA is to locate changes that impact conceptual specifications. Optional filters are then applied to refine the output for the detection of specific kinds of impacts. Two filter algorithms are given, one to detect the impact of a change on the degree of generality and one to detect the compatibility with different versions. In the second step, the change is implemented, and the differences between the original and the modified program are identified. This step is followed by the construction of a dependency graph, with nodes being annotated with information obtained from the differencing process. The last step involves a depth-first search, which is performed to propagate the impacts of a change through the graph. Additional node information is used to reduce change propagation. The researchers also created two algorithms for use in searching for specific kinds of impacts, namely, constraint change, which verifies if the requirements for algorithm parameters have changed, and concept compatibility, which verifies the compatibility of a concept between versions.

Petrenko and Rajlich[17] developed JRipples, which uses static dependency analysis and enhances impact analysis to handle the variable granularity of the analyzed software artifacts. A dependency graph of the program is also built to mark the annotated nodes (e.g., changed, propagated, inspected, or blank). Using this method, the programmer determines whether the impact propagates at a coarse or at a fine level of granularity. All the child nodes of an impacted element will be marked. If the granularity is defined by the programmer, the parent nodes will be marked. If granularity does not exist, entire code fragments will be selected by the programmer. Subsequently, the all-encompassing entities will be marked and used to propagate the impact.

Other source code entities may be affected by changed methods that call them either directly or indirectly. Assessing the impact of a method change is best achieved by analyzing the call behavior of a system. Generally, call graph analysis involves a statistical analysis of the source code. The method calls are then extracted and stored in a graph or matrix, which is used to assess the propagation of an assumed change.

An approach based on call graphs was developed by Ryder and Tip[10]. The affected tests are checked, and the ones that must be re-executed are identified using this approach. This approach also determines the change that caused the test to fail. A tool called Chianti was later developed based on this approach [18].

An approach called CCGImpact, which is implemented in PCIA, was proposed by Badri et al. [19], who sought to improve the call graph's accuracy in predicting changes and to keep costs low. Thus, the static call graphs were combined with static control flow information, thereby addressing the lack of precision of traditional call graph-based approaches. This method records all control flows between method calls to improve the call graph with information on the call order. Infeasible paths are eliminated by removing excluded calls. To create control call graphs (CCG), the source code is analyzed without considering the statements and instructions that do not invoke a method call. Sequence information can enhance CCGs by generating compacted sequences that can be used in pruning infeasible paths. The process ultimately obtains a program's behavioral profile.

Program slicing was proposed by Weiser [20]to support dependency analysis. This method captures the key portions of a program and then removes parts that are unimportant in evaluating specific variables at a certain location.

Binkley and Harman [21] introduced the concepts of dependence clusters and dependence pollution based on slicing and proposed a visualization method to locate the clusters. The size of computed slices is significant here. To perform this method, slices of the same size are createdfor each variable. The variables comprise a cluster, and any change in a variable is replicated to the other variables within the same cluster. Further studies found that 80% of all studied programs contained clusters, 10% of which are unchanged.

GRACE, is a CIA, which is a program dependency graph (PDG) based on Visual Basic, was developed by Korpi and Koskinen[22]. GRACE is capable of supporting impact analysis for Visual Basic. The program uses static forward slicing to capture the possibly affected parts of a program. GRACE consists of the parser, which translates Visual Basic programs into abstracts yntax trees (AST), and the PDG generator, which converts the ASTs into graphs. GRACE also combines single graphs with system graphs. To compute the impacted code entities, GRACE performs a straightforward reachability analysis on the PDG.

One challenge in static slicing is determining how it can be enhanced to reduce the set of proposed impacts. [23] used probabilistic algorithms to address this issue, basing their work on the three observations. First, the probability of being affected by a particular change is not the same for all statements. Second, the likelihood of being affected is low for some data dependencies. Finally, changes with a higher probability are propagated by data dependencies rather than by control dependencies.

Impact analysis can be supported when evolutionary outlines are uncovered from the source code [24]. To accomplish this, the information is retrieved, allowing developers to track frequent change patterns and pinpoint which code entities are involved.

The patterns on a class level, which will be used to retrieve the information, were identified by Vaucher et al.[25]. The method they used to calculate the level of change for each class evolution involved the retrieval of information on the implementation and functional changes from the source code. To recover implementation changes, the numbers of the added, removed, and modified instructions of methods are counted. The total change of a class is then determined by computing the sum of all relative and functional changes. Upon acquiring all the information, classes are clustered according to their change behavior. Furthermore, similar groups of class clusters (e.g., class patterns, code stabilizer, punctual changes, and common concerns) are identified using dynamic time warping.

The class patterns are then grouped according to which ones change simultaneously. Code stabilizations are the new classes that must undergo modifications before attaining stability. Punctual changes that provide data on the classes are grouped according to the changes in specific versions, and classes implementing the same changes should be identified.

A new set of coupling measures for classes based on information retrieval techniques was proposed by Poshyvanyk

et al. [26]. Their goal was to overcome the limitations of static coupling measures. They proposed that different dimensions must be covered by a reliable coupling measure in impact analysis. To detect similarities among code entities, overlapping identifiers, such as names and comments, are used in a method called latent semantic indexing (LSI). Through LSI, a document matrix that captures the distribution of words in methods is built. Each document is denoted as a vector in the LSI subspace. After constructing the matrix, the cosine among vectors is computed to measure the conceptual coupling between methods. The classes are then ranked using coupling measures for impact analysis.

3.1.2 Dynamic Dependency Analysis (DDA)

The numerous dependencies in program executions can be addressed by DDA. DDAobserves the accesses performed by a program to detect all data dependencies**.** *Dynamic slicing* and *control dependency* are proposed methods for dynamic data dependency analysis[11].

Dynamic slicing was introduced by Korel and Laski [27]. Computing dynamic slices can be achieved using an iterative algorithm based on data flow equations. Unlike static slicing, dynamic slicing identifies only the statements that affect on a particular execution trace.

Law and Rothermel[28]developed PathImpact and EvolveImpact, which are dynamic impact algorithms that are based on dynamic slicing. PathImpact collects "method entry" and "method exit" events as execution traces. However, this process could result in duplicate entries because the methods can be called more than once. Traces were compressed to directed acyclic graphs (DAGs) using the SEQUITUR compression algorithm to address this issue. Nevertheless, PathImpact encountered other problems, such as the challenge to rebuild the entire DAG for larger programs. EvolveImpact was proposed to meet this need. To update the DAG incrementally, EvolveImpact observes any changes to the test suites and system components. A unique identifier that appears at the beginning of each trace and is supplemented by special ending symbols marks each test case. If any changed method is present in the DAG or if the DAG must be refreshed, these keys and ending symbols are used to remove the traces. To handle the new symbols and keys, Law and Rothermel developed a modified version of the SEQUITUR compression algorithm called ModSEQUITUR.

A. Orso et al.[1] developed another approach for dynamic slicing, gathering field data from real users under real-world conditions to be used in evaluating software and in performing impact analysis is necessary for dynamic slicing. Remote analysis and field data gathering can be accomplished by the Gamma approach. In this approach, developers use their programs tocollect dynamic data, such as execution traces. The users then execute the programs and then send the execution data back ("coverage data at block and method levels"). This step is followed by the computation of the dynamic slices based on the execution data of entities traversing a changed entity. They compared this approach with static slicing and call graphs and found that real field data differ from synthetically computed data.

Execute-After (EA)sequences were later introduced by Apiwattanapong et al. [29]. EA sequences attempted to address the performance and precision limitations of Path Impact and Coverage Impact and achieved this goal by simplifying the recorded traces. Only the first and last events of each method were recorded. Each EA sequence had timestamps to determine which methods were impacted and also to see if the possibly impacted methods have timestamps that are equal to or higher than the first timestamp of the changed method.

In 2006, Huang and Song proposed a dynamic slicing technique that sought to address the cost issues and time limitations of both the EA sequence and PathImpact. This technique focused on collecting method return into events. This technique utilized an algorithm that retrieves the execution instruction of methods and then associates method events with the methods executed after the event. The recorded information is later applied to reduce redundancy. Methods are only listedin the impact set if and only if they are executed after a changed method was executed.

Online impact analysis is another technique that facilitates dynamicslicing. This method was introduced by [30]. In this technique, execution traces are gathered during runtime, using a complier to add callbacks to each function. After completing the data collection, the call stack was investigated to determine if a function has been affected by any changes in the other functions. All the direct and indirect functions are impacted if they were called by a changed function.

Research by Breech et al. [31]combined the precision of static techniques and the analysis speed of dynamic techniques. In their method, they built an influence graph of the program because changes can only ripple through return values, parameters, and global variables. The graph includes methods and their relations, which are represented by nodes and edges, respectively. Dynamic slicing is then performed, merging the results with information from the influence graph. Only the impacted methods that are located in the influence graph with the same type of relation,as inferred by the dynamic slices, are counted.

Mohammad [32] attempted to improve understanding of CIAs. Program slicing and traceability links assist in obtaining the impacts within a system. The CATIA tool developed by [33] served as the basis for her work. To support impact analysis, her work involved visualizing program dependencies and traceability links, which are used to track the changes across different entities of the program. A graph view for impacted elements was used by the author to enhance the tool.

According to [34],the executed statements have a *control dependency*. [35]later identified the control flow analysis as calling dependencies, logical decisions, and complexity of structure.

Dynamic function coupling (DFC) was introduced by (Besz´edes, Gergely et al. 2007). This method involves studying the relations of two functions to determine whether one affects the other. To find these relations, the distance between the call levels is computed. The approach uses forward and backward EA relations. The DFC values are derived from a dynamic call tree, which includes method entry and method exit proceedings. Function pairs that were selected according to the DFC measure are considered to be impacted by a change in one of the functions.

To compute the impact of a method change, Gupta et al. [36]proposed a new dynamic algorithm to trace the impact of a change on other program variables. Tracing impacts relies on the dependencies among program entities. The type, usage, and scope of the data dependency help in distinguishing among dependencies. A control flow graph (CFG) of the program, which consists of nodes and edges, is built. Given

that different types of dependencies exist between the nodes, the directly and indirectly impacted entities can be identified. Direct impacts can be computed when all the usage traces of the changed node are collected, whereas indirect impacts can be computed when the collected direct sets are analyzed and inspected according to their dependency.

Gupta et al. expanded their work one year later, classifying changes into functional, logical, structural, and behavioral to improve impact detection. Functional changes are the statements that affect functions, whereas logical changes refer to control-flow changes. Structural changes focus mainly on adding or deleting code entities, and behavioral changes focus on changes in execution order, such as changes in program entry and exit. In the proposed algorithm, the original and modified versions of the program are analyzed to verify whether any difference exists. Such differences are then stored in a database. Functional impacts are calculated using a classification algorithm. Furthermore, to calculate all logical changes, the statements in the CFG are analyzed. The changes that are dependent on the statements are printed out and added to the impact set. Finally, the structural changes are determined when all added or removed statements of the original code have been classified. To calculate the behavioral changes, the statements that cause the behavioral changes in the original program are added to the impact set.

4.J-via AND CASE STUDIES

Regarding given related works, dynamic change impact analysis (CIA) is a great leverage to identify the potential consequences of software evolution, as well as the impacted modules or functions. In order to do so, a prototype named J-via is implemented. The implemented prototype is in java language and is able to dynamically record all trace trails on runtime from statement level, and finally visualize the logged files. Structure for J-via is given into Figure 3.

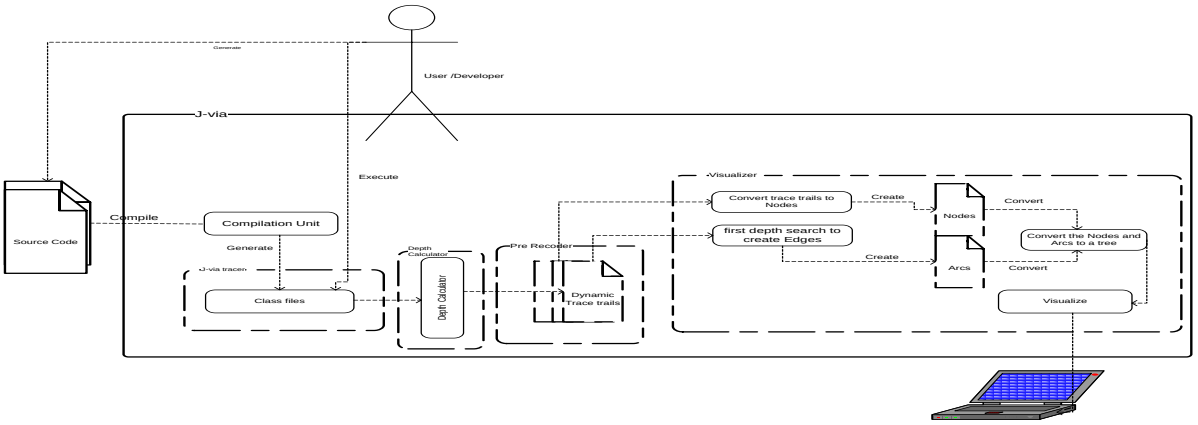


Figure 3: J-via Structure

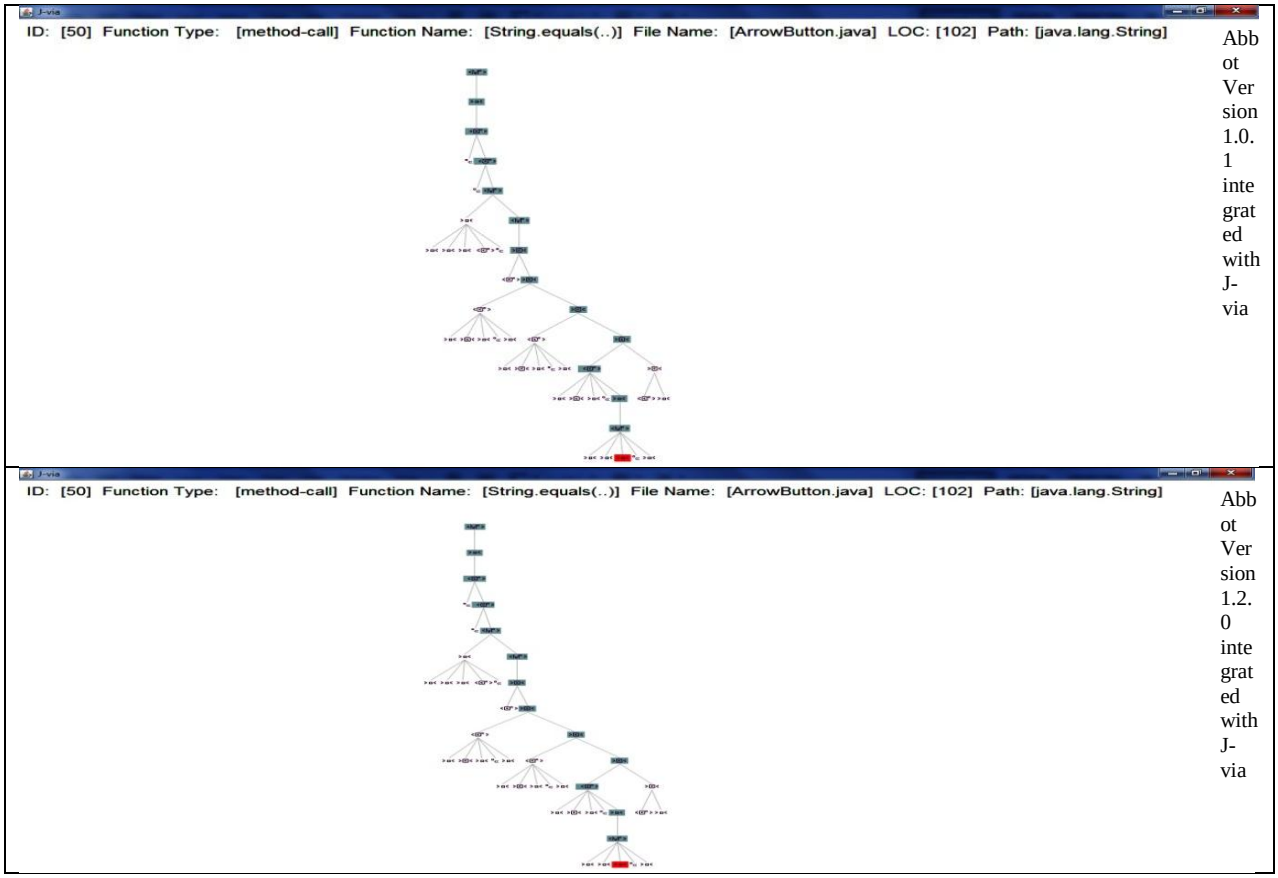
The recorded trace trails among an execution are categorized into six groups (e.g., method call, method execution, constructor call, constructor execution, field get, and field set. (Note: since the focus for this study is following the consequence of change. Therefore, from defining of syntax and semantic for each function is waived).

4.1 case studies

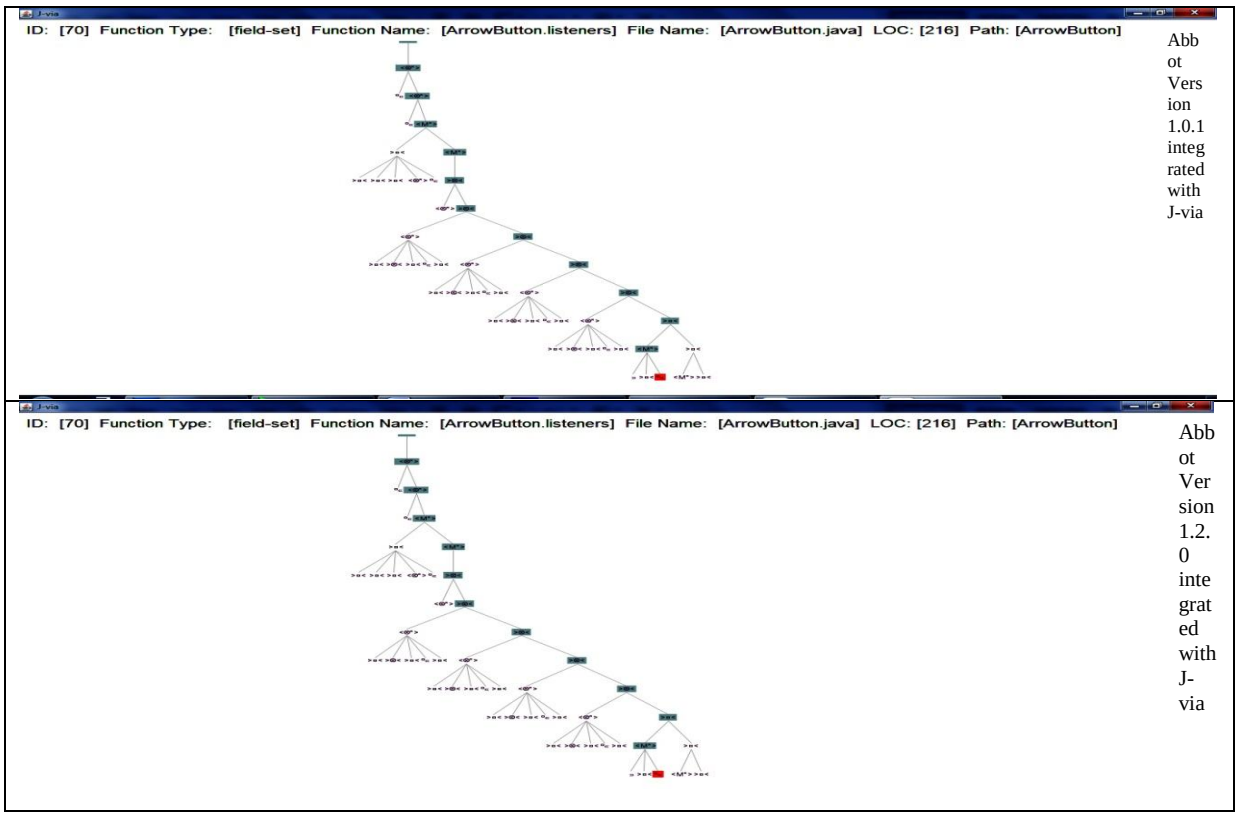
From section 2, success in *selective regression testing* requires knowing the potential consequences of software evolution, as well as the impacted modules/ functions. In order to do so, results from running two versions of abbot (i.e., abbot 1.0.1. and abbot 1.2.0) integrated with J-via were studied. Abbot versions(1.0.1) and (1.2.0) are pure Java applications, each consisting of more than 18 classes, and nine packages. Abbot is originally designed for GUI–component testing in java. In order to do following the consequence of changes, both versions of abbot were examined using similar test cases (i.e,, ArrowButtonTest.java). J-via recorded all the interactions and reported all the consequences of the changes in both versions, as well as the impacted modules/functions.

5. RESULTS

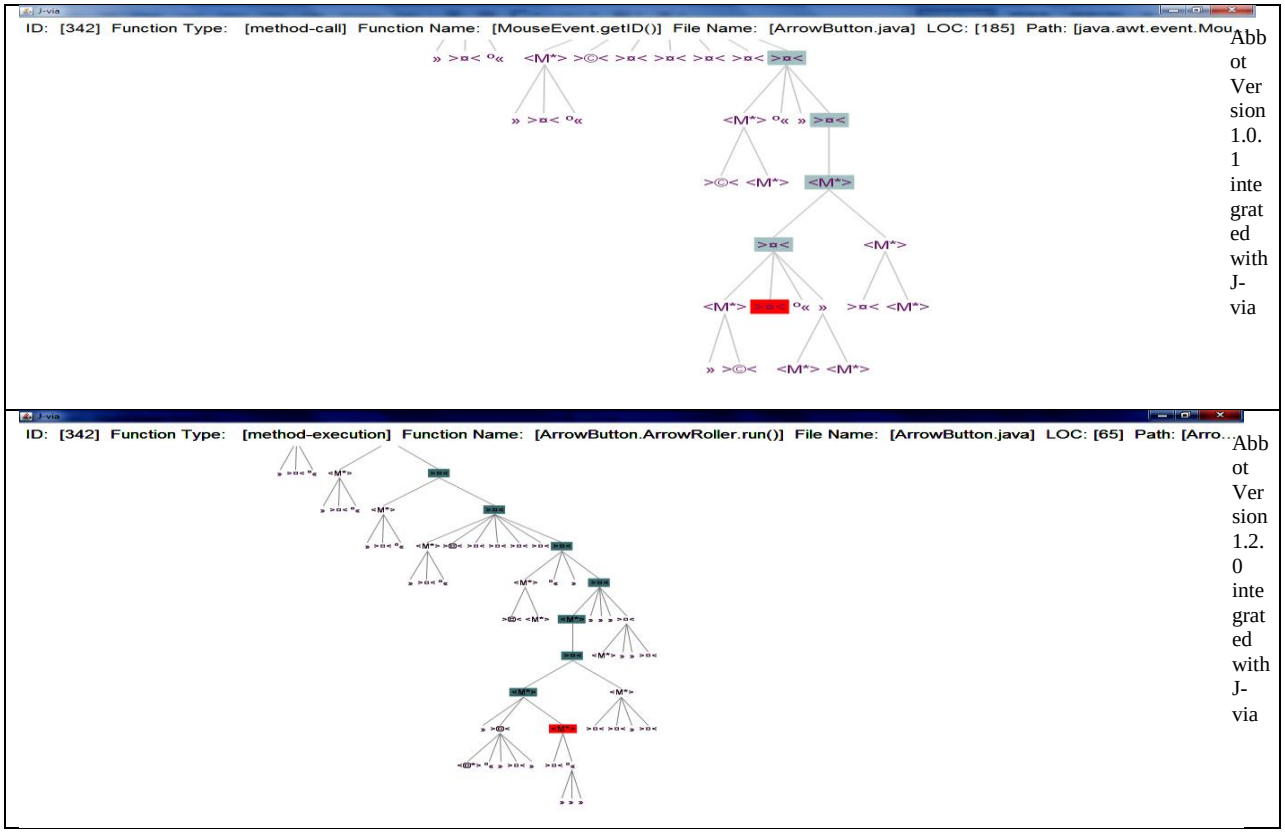
The results were obtained from running both versions of abbot integrated with J-via in Eclipse (3.2) environment using Windows 7 Ultimate (32 bit) operating system on an AMD processor (1.3 GHz). The summary of the results from running both versions of abbot with same test case (e.g., ArrowButtonTest.java) are given in below figures.


ID: [50] Function Type: [method-call] Function Name: [String.equals(..)] File Name: [ArrowButton.java] LOC: [102] Path: [java.lang.String]
Abbot Version 1.0.1 integrated with J-via
ID: [50] Function Type: [method-call] Function Name: [String.equals(..)] File Name: [ArrowButton.java] LOC: [102] Path: [java.lang.String]
Abbot Version 1.2.0 integrated with J-via

(a)


ID: [70] Function Type: [field-set] Function Name: [ArrowButton.listeners] File Name: [ArrowButton.java] LOC: [216] Path: [ArrowButton]
Abbot Version 1.0.1 integrated with J-via
ID: [70] Function Type: [field-set] Function Name: [ArrowButton.listeners] File Name: [ArrowButton.java] LOC: [216] Path: [ArrowButton]
Abbot Version 1.2.0 integrated with J-via

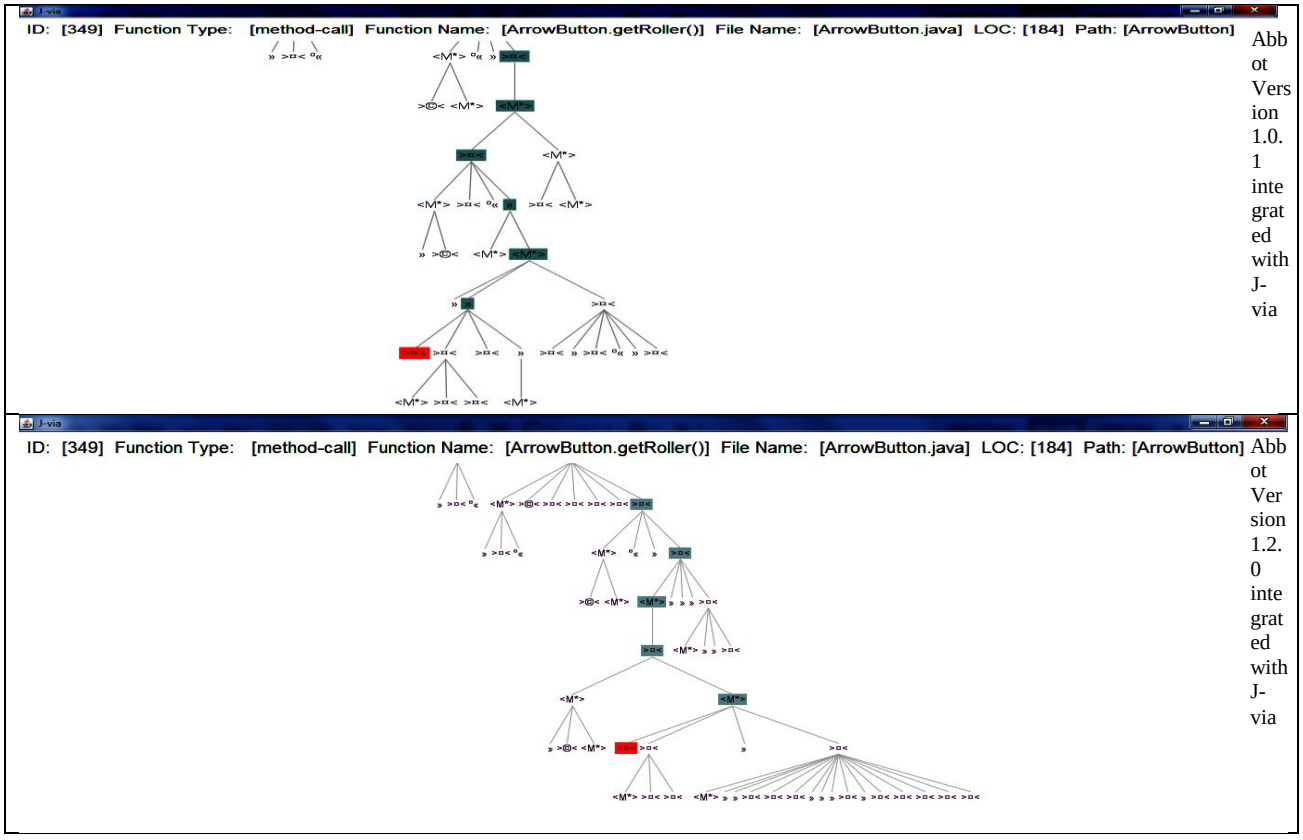
(b)



Abbot Version 1.0.1 integrated with J-via

Abbot Version 1.2.0 integrated with J-via

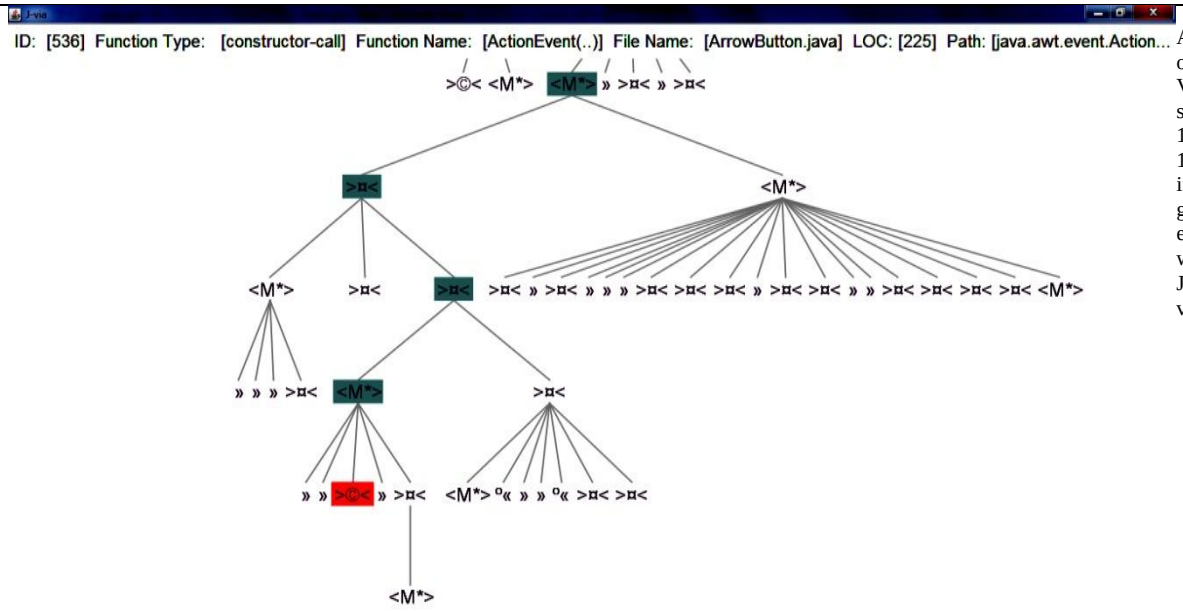
(c)



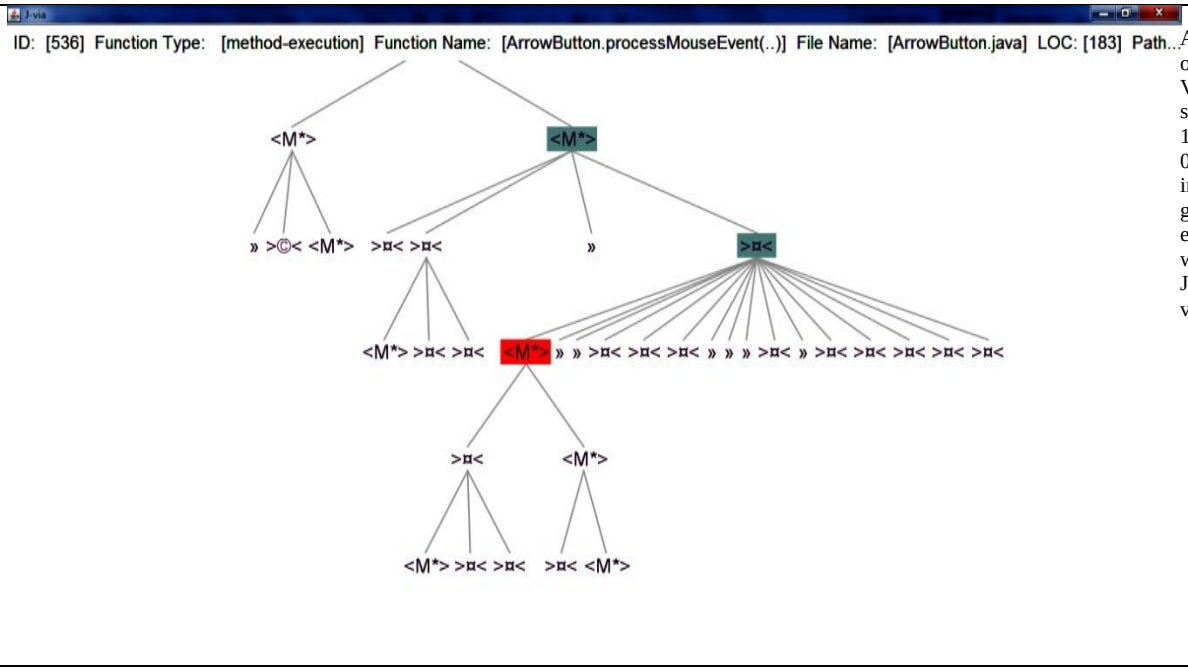
Abbot Version 1.0.1 integrated with J-via

Abbot Version 1.2.0 integrated with J-via

(d)

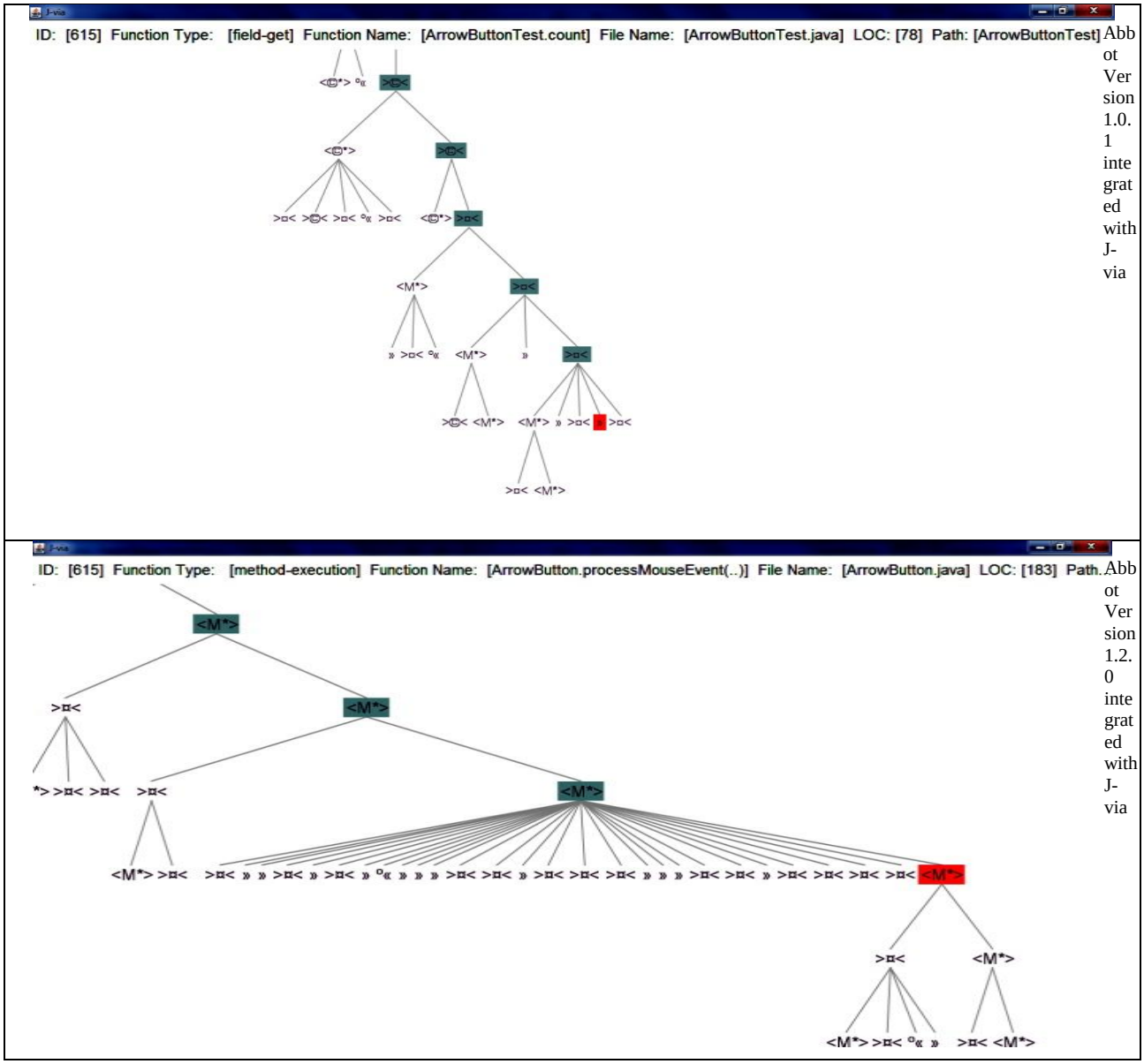


Abbot Version 1.0.1 integrated with J-via



Abbot Version 1.2.0 integrated with J-via

(e)



(f)

Figure 4: Consequential Images for affected parts by change on Run the ArrowButtonTest.java Integrated with J-via on Both Versions of Abbot; (a) Comparing Nodes With Number 50; (b) Comparing Node with Number 70; (c) Comparing Node Number 342; (d) Comparing Nodes with Number 349; (e) Comparing Nodes with Number 536; (f) Comparing Nodes with Number 615

The above given figures are some random chosen modules out of 616 interactions of Abbot 1.0.1 and 1313 recorded interactions from Abbot 1.2.0. Figure 4(a) shows the comparison of nodes with number 50 in both versions of abbot on run ArrowButtonTest.java, given figure indicates that the desire function (e.g., String. equals()) didn't affect by change. Furthermore, along with execution Figure 4(b) specifies that any function didn't affect by change. However, unlike previous nodes, the node with number 342 is different (see Figure 4(c)). The left picture shows that in Abbot 1.0.1 the node 342 is a "method-cal" named "MousEvent.getID" while right figure is indicating that the node 342 in Abbot 1.2.0 is a "method-execution" named "ArrowButton.ArrowRoller.run()".Figure 4(d) shows that the path of execution is back to normal and nodes with number 349 in both Abbot versions are not affected by change. Right and left figures of Figure 4(d) show that the node number 349 in both versions of Abbot (i.e., 1.0.1, and 1.2.0) are "mthod-call" named "ArrowButton.getRoller()". Furthermore, Figure 4(e)(f), are showing another parts in both versions of abbot that are affected by change. Left figure from Figure 4(e) shows the node number 536 from Abbot 1.0.1 is a "constructor-call" named "ActionEvent()", while right figure indicating that node number 536 in Abbot 1.2.0 is "method-

execution" named "ArrowButton.ProcessMouseEvent()". In end, comparison of Figures 4(a) to 4(f) shows the ability of following the consequence of change visually in J-via. This property of J-via can be considered a good option to address the issue of selective retest in regression testing. Furthermore, the summarized results for affected classes, and number of interactions in both versions of abbot on run ArrowButtonTest.java is given into Table 1.

Table 1. The Summarized J-via Results for Running*ArrowButtonTest.java* in Both Versions of Abbot

Abbot Version	**1.0.1**	**1.2.0**
Affected Classes	ArrowButtonTest.java ArrowButton.java abbot.tester.ComponentTester	ArrowButton.java ArrowButtonTest.java abbot.tester.ComponentTester ArrowButton$ArrowRoller ArrowButtonTest$1
Number of Test Cases	2	2
Number of Interactions	616	1133

Table 1 illustrates the summarized results of J-via from running ArrowButtonTester.java into both version of abbot (e.g., 1.0.1, and 1.2.0). In given table the results for affected classes are given. These results can be used in selective testing to show which parts needed to be re-test.

6. CONCLUSION AND FUTURE WORK

This research was aimed at define a new approach for dynamic change impact analysis (CIA) and visualization support for selective regression testing. As regards the argued problem definition in section 2.0, the identification of the consequences of change to find the impacted modules/functions provides good leverage for supporting the selective-retest approach over the retest-all approach to address some issues (i.e., rework and expensiveness). In order to do so, a prototype named J-via to support for selective regression testing by browsing among the consequences of changes to identify the affected functions/modules is developed and evaluated.

6.1 Future Work

J-via was shown to have the capacity to support selective re-testing, which is a type of regression testing. In the given example for regression testing, each version of the program must be separately run before comparisons of the obtained results are made. However, the implementation of an additional functionality for J-via can help in rendering a differentiation of two versions of a program and in returning the impacted parts for same execution path automatically.

7. REFERENCES

[1] A. Orso, T. Apiwattanapong, et al. (2003). 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, Helsinki, Finland.

[2] April, A., J. H. Hayes, et al. 2005. "Software Maintenance Maturity Model (SMmm): The Software Maintenance Process Model: Research Articles." Journal of Software Maintenance. Evolution 17(3): 197-223.

[3] F. Cuadrado, et al. A 2008.Case Study on Software Evolution towards Service-Oriented Architecture.in Proceedings of 22nd International Conference on Advanced Information Networking and Applications - Workshop.

[4] Pfleeger, S.L., 1998. Software Engineering: Theory and Practice. : Prentice-Hall, Inc. 576.

[5] Turver, R.J. and M. Munro, 1994. An Early Impact Analysis Technique for Software Maintenance. Journal of Software Maintenance: Research and Practice, 6(1): p. 35-52.

[6] Jashki, M.-A., R. Zafarani, and E. Bagheri, 2008. Towards a more efficient static software change impact analysis method, in Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and Engineering. ACM: Atlanta, Georgia. p. 84-90.

[7] Orso, A., et al. 2004. An Empirical Comparison of Dynamic Impact Analysis Algorithms.in Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK.

[8] Agrawal, H. and J.R. Horgan. 1990. Dynamic Program Slicing. in Proceedings of the ACM SIGPLAN Conference on Programing Language Design and Implementation. White Plains, New York, USA: ACM.

[9] B. Korel and J. Rilling. 1997. Dynamic Program Slicing in Understanding of Program Execution. in Proceedings of 5th International Workshop on Programing. Computer Dearborn, MI , USA.

[10] Ryder, B.G. and F. Tip. 2001. Change Impact Analysis for Object-Oriented Programs. in Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Programing Analysis for Software Tools and Engineering Snowbird, Utah, USA: ACM.

[11] Chao, L., et al. 2007. Indexing Noncrashing Failures: A Dynamic Program Slicing based Approach.in Proceedings of IEEE International Conference on Software Maintenance., Paris, France.

[12] Zamli, K.Z., et al., 2009. Selective Retest, in Software Testing.Open University Malaysia (OUM). p. 90-91

[13] Jitender Kumar, Chhabra, and K. K. Aggarwal, 2006.Measurement of Intra-Class \& Inter-Class Weakness for Object-Oriented Software.in Proceedings

of the Third International Conference on Information Technology: New Generations.

[14] M. Lee, A. J. Offutt, and R. T. Alexander. 2000. Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software. in Proceedings of the 34th International Conference on Technology of O-O Language and Systems. Santa Barbara, CA, USA.

[15] S. Gwizdala, Y. Jiang, and V. Rajlich.2003.Jtracker - a Tool for Change Propagation in Java.in Proceedings of the Seventh European Conference on Software Maintenance and Reengineering.

[16] Zalewski, M. and S. Schupp. 2006. Change Impact Analysis for Generic Libraries,".in Proceedings of the 22nd IEEE International Conference on Software Maintenance Philadelphia, Pennsylvania.

[17] M. Petrenko and V. Rajlich.2009. Variable Granularity for Improving Precision of Impact Analysis,".in Proceedings of the IEEE 17th International Conference on Program Comprehension. . Vancouver, Canada.

[18] X. Ren , et al. Chianti: 2005. A Change Impact Analysis Tool for Java Programs.in Proceedings of the 27th Int. Conference on Software Engineering St. Louis, MO, USA: ACM.

[19] Badri, L., D. St-Yves, and M. Badri. 2005. Supporting Predictive Change Impact Analysis: a Control Call Graph based Technique. in Proceedings of 12th Asia-Pacific Conference on Software Engineering APSEC '05.

[20] Weiser, M. 1981. Program Slicing.in Proceedings of the 5th Int. Conference on Software Engineering,. San Diego, California, USA: IEEE Press.

[21] D. Binkley and M. Harman. 2005. Locating Dependence Clusters and Dependence Pollution. in Proceedings of the 21st International Conference on Software Maintenance . Budapest, Hungary.

[22] J. Korpi and J. Koskinen, 2007. Supporting Impact Analysis by Program Dependence Graph Based Forward Slicing. in Advances and Innovations in Sys. Computing Sciences and Sofware. Engineering, p. 197-202.

[23] R. Santelices and M. J. Harrold, 2010.Probabilistic Slicing for Predictive Impact Analysis. Georgia Tech Center for Experimental Research in Communication System.

[24] D. Binkley and D. Lawrie, 2010. Information Retrieval Applications in Software Maintenance and Evolution.in Encyclopedia of Software Engineering.

[25] S. Vaucher, H. Sahraoui, and J. Vaucher. 2008. Discovering new Change Patterns in Object-Oriented Systems. in Proceedings of the 15th Working Conference on Reverse Engineering Washington D.C.

[26] D. Poshyvanyk, et al., 2009. Using Information Retrieval Based Coupling Measures for Impact Analysis. Empirical Software Engineering vol14: p. 5-32.

[27] Korel, B. and J. Laski, 1988. Dynamic Program Slicing.Information Process. Letter, 29(3): p. 155-163.

[28] J. Law and G. Rothermel. 2003. Whole Program Path-Based Dynamic Impact Analysis. in Proceedings of the International Conference on Software Engineering.

[29] T. Apiwattanapong, A. Orso, and M. J. Harrold. 2005. Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences in Proceedings of the International Conference on Software EngineeringSant. Louis.

[30] B. Breech, et al. Online Impact Analysis via Dynamic Compilation Technology,".in Proceedings of the 20th IEEE International Conference of Software. Maintenance.

[31] B. Breech, M. Tegtmeyer, and L. Pollock. 2006. Integrating Influence Mechanisms into Impact Analysis for Increased Precision. in Proceedings of of the 22nd IEEE International Conference on Software Maintenance.

[32] R. N. Mohamad, 2010. A Change Impact Analysis Approach Using Visualization Method, in Computer Science and Information Systems,.Malaysia University.

[33] S. Ibrahim, et al., 2005. Implementing a Document-based Requirements Traceability: A Case Study, in IASTED International Conference on Software Engineering.

[34] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing using Dependence Graphs. in Proceedings of ACM SIGPLAN Conference on Programing Language Design and Implementation. Atlanta, Georgia, USA: ACM.

[35] Wilde, N. and R. Huitt, 1991. Maintenance support for object oriented programs. IEEE Transactions on Software Engineering, p. 162-170.

[36] Gupta, Y. Singh, and S. Chauhan, 2010. *A Dynamic Approach to Estimate Change Impact Using Type of Change Propagation.*Journal of Information. Processing System.,**6**: p. 597-608.