Impact Analysis via Reachability and Alias Analysis

Wen Chen, Alan Wassyng, Tom Maibaum

Impact Analysis via Reachability and Alias Analysis

Wen Chen, Alan Wassyng, and Tom Maibaum

McMaster Centre for Software Certification,
McMaster University, Hamilton, Ontario, Canada
{chenw36,wassyng}@mcmaster.ca
{tom}@maibaum.org

Abstract. This work is concerned with localizing and analyzing the potential impact of changes to large-scale enterprise systems, and, in particular, how to incorporate *reachability* analysis and *aliasing/pointer* analysis to minimise *false-positives* and eliminate *false-negatives*. It is a continuation of our previous work, which included static analysis [1] and dynamic analysis [2] of changes to systems containing hundreds of thousands of classes and millions of methods. This current work adds: reachability analysis that examines the program to see "whether a given path in a program representation corresponds to a possible execution path", such that *infeasible* paths of mis-matched calls and returns can be filtered out from the estimated impact set; and alias analysis to identify paths that are *feasible* but cannot be affected. Using our approach, organizations can focus on a much smaller, relevant subset of the test suite instead of performing their entire suite of tests without any idea as to whether any test is necessary. Also, in the future, we hope to be able to help testers to augment the test suite with new tests that cover the impacted methods/paths not already subjected to testing. We include a case study that illustrates the savings that can be attained.

Key words: Large-scale Enterprise Systems: Impact Analysis: Reachability Analysis: Alias Analysis: Static Analysis: Dynamic Analysis: Instrumentation: Regression Testing

1 Introduction

The target system in this study is *large-scale enterprise systems*. Large-scale enterprise systems are commercial software packages that enable organizations to integrate various applications, replacing hard-to-maintain interfaces, eliminating redundant data entries, etc., to accommodate business growth. One of the largest enterprise vendors SAP, had 2012 revenues of 16.22 billion Euros [3]. Enterprise systems are clearly a common phenomenon in the IT marketplace with fast growing needs. However, implementing enterprise systems may lead to high costs for software maintenance and testing, since corrective changes and enhancements are made on a frequent basis. One type of software change, such as vendor *patches*, typically have to be applied as they are required to upgrade the system in order to fix defects, and to introduce new features.

Enterprise systems are complex, critical and costly. For instance, Oracle Corporation's *E-Business Suite* [4] has over 230 thousand classes, and 4.6 million functions. Despite problems due to their inherent complexity, enterprise systems play a critical role in many organizations. They are used to implement actual business processes, information flows, reporting, data analytics, etc. It is estimated that "Large companies can also spend $50 million to $100 million on software upgrades. Full implementation of all modules can take years" [5]. As a consequence of these characteristics, these systems can also often be classified as *legacy systems* and are poorly understood and difficult to maintain.

Impact analysis is the key in analyzing software changes or potential changes and in identifying the software objects the changes might affect [6]. Organizations need a change impact analysis tool to identify the impacts of a change after or even before a making a change. If the impacts can be obtained even before applying the change, it enables the organization to make test plans or to run tests in advance, saving the lag between system deployment and release.

Conventional impact analysis includes static approaches, dynamic approaches or a hybrid of the two. Static approaches identify the impact set – the subset of elements in the program that may be affected by the changes made to the system – by analyzing relevant source code or compiled code. Dynamic approaches collect information about execution data for a specific set of program executions, such as executions in the field, executions based on an operational profile, or executions of test suites.

2 Research Motivation

Our original work in this domain [1] showed that we could obtain a set of static impacts which are safe and more precise than conventional `vanilla` static approaches, while another more recent work [2] combined the static approach with dynamic instrumentation (aspect-based), and is able to identify real impacts at run-time to further improve the precision. However, in spite of the success of this recent approach, the case studies suggested that there still might be a good number of false-positives present in the estimated impact set. That analysis found out that only a tiny portion of the system (0.26% of all top functions/APIs) were affected at run-time. Even though those top functions were executed over 150 thousand times, one could not conclude that the rest of the static impacts were safe to discard. Consequently, testers may still need to rerun many of the regression tests.

While seeking further analysis to remove more false-positives, we realized that *Reachability Analysis* can be used to determine whether, within a graph G, a node s can reach another node t, *i.e.*, whether the path $s \rightsquigarrow t$ is feasible, and so seemed a promising tool in our search for reducing false-positives. We also identified *Alias Analysis* as a potential tool to further remove false-positives by identifying changed and aliased variables and methods that can access them.

3 Related Work

In graph theory, reachability refers to the ability to get from one vertex to another within a graph by traversing edges of the graph. Algorithms for determining reachability fall into two categories: those that require preprocessing and those that do not [7]. Algorithms like *breadth-first search*, in which reachability of one node from another node can be determined directly without the use of complex data structures, are in the first category. While algorithms like *Floyd-Warshall*, *Thorup's algorithm* and *CFL-reachability* fall into the second category, where more sophisticated methods and/or complex data structures are required.

Reps *et al.*[8] showed how a number of program analysis problems can be solved by transforming them to *graph-reachability* problems. The purpose of program analysis is to ascertain information about a program without actually running the program. In his work, program-analysis problems can be transformed to *context-free-language reachability problems* ("CFL-reachability problems").

Many compiler analyses and optimizations require information about the behaviour of pointers in order to be effective. *Pointer analysis* is a technique for statically determining the possible runtime values of a pointer [9]. Aliasing occurs when two distinct names (data access paths) denote the same run-time location. This analysis has been studied extensively over the last decade. Alias information is central to determining what memory locations are modified or referenced. Ondrej introduces a flexible framework `SPARK` for experimenting with points-to analyses for Java [10]. `SPARK` is intended to be a universal framework within which different points-to analyses can be easily implemented and compared in a common context. We believe that aliasing analysis is useful in hybrid impact analysis, and that we can use it to identify aliased objects in the static dependency graph to remove false-positives from the impact set.

4 Reachability Analysis

Ordinary (flat) graph reachability analysis does not take into account the fact that, in practice, many apparently reachable paths can be infeasible because of mis-matched calls and returns, and this information can only be obtained by considering control flows and/or data flows of the program. Reps [8] introduced the *Context-Free-Language Reachability Problem* as:

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

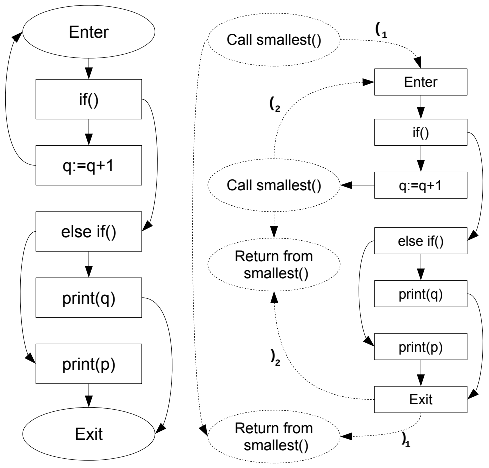
Then an ordinary graph reachability problem can be transformed into a CFL-reachability problem by labelling each edge with a symbol e and letting L be the regular language e^*. The reason that we introduce CFL-reachability analysis here is that it can help us answer the *undecidable* question: **"Does a given path in a program representation correspond to a possible execution path?"**. The idea is that we can define a context-free language L to represent feasible

```
Smallest(int p, int q){
\*precondition p>1 && q=2*\

if (p mod q > 0 && q < sqrt p)
then
    q := q+1;
    smallest(p, q);

else if (p mod q = 0)
then
    print(q, is factor)

else
    print(p, is prime);
}
```



(a) Program Smallest (b) Control-flow Graph and Supergraph

Fig. 1: Program smallest and its graphs. Dashed nodes and arrows correspond to extra nodes and edges while expanding from G to G^*.

paths and then determine if a given string ω is recognizable in L, i.e., is $\omega \in L$? Our assumption is that paths that can possibly be feasible execution paths are those in which "returns" are matched with corresponding "calls". These paths are called *realizable* paths.

A *Supergraph* G^* [8] was defined to deal with *realizable* paths. A supergraph consists of a collection of control-flow graphs – one for each procedure. Each flowgraph has a unique *start* node and a unique *exit* node. The other nodes of the flowgraph represent *statements* and *predicates* of the program in the usual way, except that each procedure call in the program is represented in G^* by two nodes, a *call node* and a *return-site* node. In addition to the ordinary intraprocedural edges that connect the nodes of the individual control-flow graphs, for each procedure call G^* contains three more edges: an intraprocedural *call-to-return-site* edge; an interprocedural *call-to-start* edge; and an interprocedural *exit-to-return-site* edge.

Suppose we have a simple recursive program `smallest` (Figure 1a) to find the smallest prime factor of a positive integer number. In Figure 1b, the graph on the left is the regular control-flow graph of program `smallest`, and the one on the right is the extended supergraph.

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

$$
\begin{aligned}
\mathsf{matched} \rightarrow & \mathsf{matched\ matched} \\
& \mid (_i\mathsf{matched}\ )_i\textbf{for } 1 \leq \mathsf{i} \leq \mathsf{N} \\
& \mid \mathsf{e} \\
& \mid \varepsilon
\end{aligned}
$$

From the supergraph, we can identify paths, for example:

- "Call Smallest → Enter → if() → else if() → print(q) → Exit → Return from Smallest", which has word "$(_1eeee)_1$", is a feasible path since the call-to-start edge "$(_1$" is matched by a correct exit-to-return-site edge "$)_1$".
- however, for the same path that exits to the inside return-site node ("$(_1eeee)_2$"), we consider it infeasible – "$(_1$" was mistakenly matched by "$)_2$".

5 Alias Analysis

Alias analysis, *pointer* analysis, *points-to* analysis, *pointer alias* analysis etc., are often used interchangeably to denote an analysis that attempts to analyze pointers and aliases, such as run-time values of a pointer, or an aliased pair of names that point to the same run-time location due to the use of pointers or references. Typically, results of alias analysis are sets of aliased variables, say, $aliased(x)$. If $l \notin aliased(x)$ for abstract location l and variable x in the program P, then x can never alias to variables represented by l in some execution of P.

Suppose we have a program `aliasingTest` to test if the three types of variables in Java can be aliased: *class variable (static field)*, *instance variable* and *local variable*. All the three types of variables (integer arrays in this example) are first initialized (Line[9]-Line[14]) with integer 1 at the first index. Then we create aliased variables to each of the three (Line[16]-Line[18]). Instead of manipulating the original variables, we run functions on the aliased ones (Line[20]-Line[23]).

After the invocations of the first three functions (either static or non-static), the original variables were actually changed (with first element altered to integer 11), even though the functions only manipulated the aliased copies. Our observation is that, if along a path in the access dependency graph of a program one can obtain the aliasing information for each method, dependencies among methods can be identified more precisely. In particular, we follow these steps to achieve more precise dependencies:

1. A *flow-insensitive and context-insensitive alias analysis* to compute a single and valid solution to the whole program.
2. We examine the pairs of aliased variables (static field, instance field, and local variable) throughout the program and obtain a mapping from each method f to variables var_f and aliased variables $aliased(var_f)$ it can access, i.e., $f \rightarrow \{var_f, aliased(var_f)\}$.

3. We examine paths in the estimated impact set, for any other changed function g with mapped variables and aliased variables $\{var_g, aliased(var_g)\}$ that can be reached by f, if and only if there exists any intersection of $\{var_f, aliased(var_f)\}$ and $\{var_g, aliased(var_g)\}$, we say f can be affected by g.

Therefore, in `aliasingTest`, a dependency edge between function *main* and function *alterEmpty* should be removed since there is no aliased variables within *alterEmpty* that was used in *main*.

Listing 1: Program aliasingTest

```
package aliasingTest;
  public class aliasingTest {
      static int[] staticArray;
      int[] instanceArray;
      public aliasingTest(){
          instanceArray = new int[6];}
      public static void main(String args
            []){
          //initializations of arrays
          staticArray = new int[5];
          int[] localArray = new int[3];
          aliasingTest at = new
                aliasingTest();
          localArray[0] = 1;
          staticArray[0] = 1;
          at.instanceArray[0]= 1;
          //aliasing to arrays
          int[] aliasOflocalArray =
                localArray;
          int[] aliasOfstaticArray =
                staticArray;
          int[] aliasOfinstanceArray = at.
                instanceArray;
```

Listing 2: aliasingTest cont.

```

          //run functions that can be
                invoked within main()
          alterArrayLocal(aliasOflocalArray
                );
          alterArrayStatic(
                aliasOfstaticArray);
          at.alterArrayInstance(
                aliasOfinstanceArray);
          alterEmpty();
      }
      static void alterArrayLocal(int[]
            array){
          array[0] = 11;}
      static void alterArrayStatic(int[]
            array){
          array[0] = 11;}
      void alterArrayInstance(int[] array){
          array[0]=11;}
      static void alterEmpty(){
          System.out.println("No job is
                doing here.");}
  }
```

6 Impact Analysis Overall

We extended our previous approachs in [1] and [2] by reachability analysis and alias analysis, to form the new process depicted in Figure 2. Note that, the set of potential false-positives was obtained by subtracting the dynamic impact set from the static impact set. The reachability analysis works on this set to find infeasible paths. The alias analysis continues cutting out false-positives from the reduced set by identifying functions that are not able to access the aliased variables of a changed function, if they are not themselves directly changed.

7 Case Study

In the study, our goal was to investigate whether this new, extended approach can meet our goal, which is to safely remove false-positives from the change impact set. We followed the same order, variables, measures, experiment setup etc., as we did in the experiments in [1] and [2].

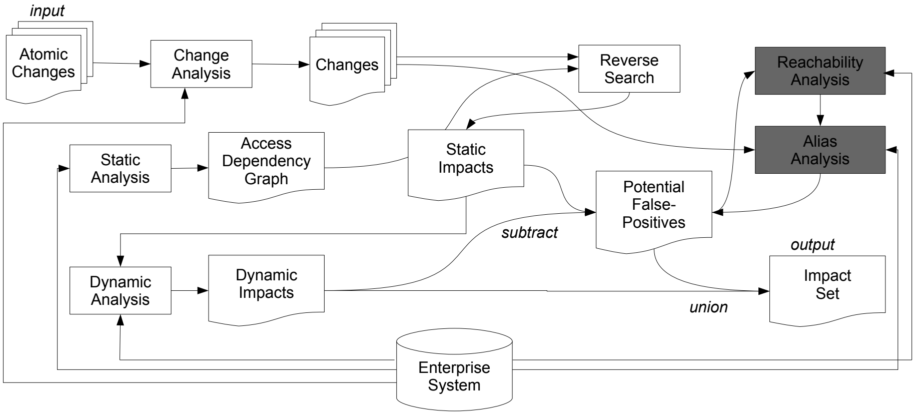


Fig. 2: System Flow of the Complete Approach

Application	Database	Classes	Entities	LOC
11.5.10.2 (11i)	10.2.0.2.0 (10g)	195,999	3,157,947	8.7 Million

Table 1: Oracle E-Business Suite Release 11i and Some Facts

There is only one independent variable in this case study: the extended impact analysis tool. Dependent variables in this study include *precision* and *time overhead*. For the measurement of precision, we used the one in Equation 1, where I represents the number of estimated impacts (functions and fields), and M represents the total number of entities in the program.

$$Precision = \frac{|I|}{|M|} \tag{1}$$

7.1 Experimental Setup

The experiment was set up on a desktop server with a Quad core 3.2GHz CPU, 32G RAM and operating system Red Hat Enterprise Linux Server release 5.10 (Tikanga) 64 bit. We used one release of Oracle E-Business Suite (Table 1) as the object of the analysis, and for the source of atomic changes we used one vendor patch (patch # 5565583, 212MB) that can be obtained either from Oracle E-Business Suite *Patch Wizard* or manually download from Oracle Metalink.

7.2 Experiment Design

We had already collected results from static analysis and dynamic analysis in the impact analysis process from [1] and [2]. Thus, for this experiment we focused only on the improvements made through reachability analysis and alias analysis.

Function	Top Function	Static Impacts	Dynamic Impacts	Potential FPs
3,157,947	1,673,132	699,534	4,806	694,728

Table 2: Instrumentation Result on Patch # 5565583

Static	Dynamic	Rmd By Reachability	Rmd By Alias	Final Impacts
699,534	4,806	61,125	86,374	547,229

Table 3: Final Impacts of Patch # 5565583

Then CFL-reachability analysis was implemented via Wala [11] to cut down false-positives. We ran the Tabulation algorithm [12] implemented by Wala on the set of "potential false-positives". The alias analysis takes the processed "potential false-positives" from the above reachability analysis as the input, and calculates aliasing information, such that methods that have no accesses to those aliased and changed variables in the system are not considered as affected. Another input is the set of changes resulting from the patch analysis. What we need is to find the methods on a particular path that access those changed variables and also variables that are aliased with them. For the sake of safety, we assume here that, if a function is changed, then potentially all of its accessible variables can be changed.

7.3 Results and Analysis

The system used in our case study contains 195,999 classes. We determined that there are 3,157,947 entities (both functions and fields) in the system. The process of building the access dependency graph added over 18.4 million dependencies and took over 9.5 hours to complete. By patch analysis, we found 16,787 direct database changes, and 25,613 direct library changes (classes) for patch #5565583. The static analysis identified 8,154 direct changed functions for this patch, which led to 699,534 affected functions (22% of the total functions), and 160,800 affected top functions (9.6% of the total top functions) in the system. The computed impacts for the patch after static analysis and dynamic analysis are shown in Table 2.

Thus we had 694,728 "potential false-positives" to work on in the reachability and alias analysis. Both CFL-reachability analysis and alias analysis were implemented via Wala. We ran Wala on the enclosing classes of each function in those "potential false-positives", and then mapped identified feasible statements to functions in the system. Then those functions with the direct changes (42,400) were given to Wala's alias analysis framework to find aliased variables for each changed function. In the end, we found many of the functions within the "potential false-positives" were not present in feasible paths (611,253) or able to access any aliased variables (863,374) of changed functions. We therefore removed 6,865,697 (37.3 %) dependencies from the original dependency graph. We summarized the results in Table 3.

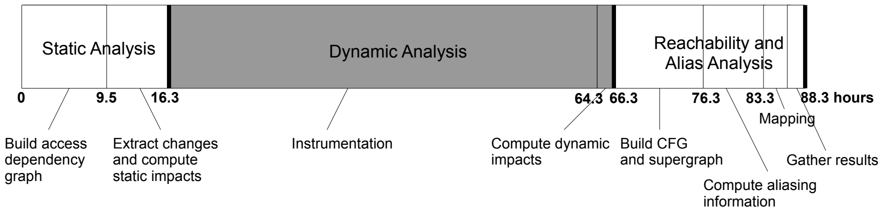


Fig. 3: Execution Time for Patch #5565583

As we can see from Table 3, we achieved a precision of 3.8% at the end of the static analysis and then improved it to 2.98% at the end of the complete approach. The dynamic analysis identified that only 4,806 functions to be executed, which left a large portion (99%) of the static impacts as potential false-positives. The reachability analysis and alias analysis reduced the false positives by 21.8% . At the current stage, our case study does not include a user's application built on Oracle's E-Business Suite, so the impacted entities are confined to E-Business Suite. That explains why, even with a reasonably large number of real executions, the dynamic impacts are associated with just a tiny part of the system.

The entire process requires considerable time to complete (see Figure 3). Considering the sizes of the system and patch, it is still more manageable than rerunning everything in the regression suite. More crucially, it provides testers more confidence as to which parts in the system are affected. The most time-consuming task is the instrumentation, which occupies around 56.8% of the total execution time. As with the static dependency graph, the instrumentation forms a substantial corporate asset for future analysis, and can be easily and quickly updated as needed.

8 Conclusion and Future Work

In this work, we have incorporated CFL-reachability analysis and alias analysis in identifying software impacts. As far as we can ascertain, these two techniques have not been used in this way to cut down on the false-positives in preceding analyses. It has been demonstrated that CFL-reachability with a parenthesis context-free grammar can be used to filter out infeasible paths (mis-matched calls and returns), that may become false-positives in the impact set. An alias analysis was conducted to identify functions that are able to access the aliased and changed variables. We consider those that are not able to access any of the aliased and changed variables to be false-positives, if they themselves are not directly changed. Also, we have demonstrated the practical applicability of the improved approach on a very large enterprise system, involving hundreds of thousands of classes. Such systems may be perhaps two orders of magnitude larger than the systems analyzed by other approaches, so our technique seems to be uniquely powerful.

Initally, considering the running time and effort expended, we were a little disappointed in the percentage of false-positives removed by this technique. However, after examining the results more carefully, we realized that: (1) the actual number of false-positives removed was significant; and (2) as we discussed earlier, because there is no user's application in our case study, the impact analysis is restricted to functions within the system, and in particular, many of the identified impacted functions are system APIs. Further study will be needed to determine whether results are better for a user application built on E-Business Suite. Also, the alias analysis we used is flow-insensitive and context-insensitive. It assumes statements in the program can be executed in any order and any number of times. In practice this is not a precise approach. The imprecision can also come from the context-insensitivity: method calls were treated conservatively, without computing the precise target addresses of the return statements. Hence, in the future, it is worth investigating whether a more precise approach can be derived for large-scale enterprise systems.

References

1. Chen, W., Iqbal, A., Abdrakhmanov, A., Parlar, J., George, C., Lawford, M., Maibaum, T., Wassyng, A.: Large-scale enterprise systems: Changes and impacts. In: Enterprise Information Systems. Springer (2013) 274–290
2. Chen, W., Wassyng, A., Maibaum, T.: Combining static and dynamic impact analysis for large-scale enterprise systems. Manuscript submitted for publication (2014)
3. AG, S.: Annual report 2012, financial highlights (2012)
4. Oracle: Oracle e-business suite integrated soa gateway implementation guide release 12.1 (June 2010)
5. Monk, E.F., Wagner, B.J.: Concepts in enterprise resource planning. CengageBrain. com (2008)
6. Bohner, S.A.: Software Change Impact Analysis. In: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW-27'02). (1996)
7. Gersting, J.L.: Mathematical structures for computer science. Macmillan (2007)
8. Reps, T.: Program analysis via graph reachability. Information and Software Technology **40**(11–12) (1998) 701 – 726
9. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM (2001) 54–61
10. Lhoták, O.: Spark: A flexible points-to analysis framework for java. (2002)
11. Center, I.T.W.R.: T. j. watson libraries for analysis main page (Jul 2013)
12. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '95, New York, NY, USA, ACM (1995) 49–61