Large-Scale Enterprise Systems: Changes and Impacts

Wen Chen[✉], Asif Iqbal, Akbar Abdrakhmanov, Jay Parlar, Chris George, Mark Lawford, Tom Maibaum, and Alan Wassyng

McMaster Centre for Software Certification, McMaster University, Hamilton, Canada
chenw36@mcmaster.ca
http://www.mcscert.ca/

Abstract. Changes and their impacts to large-scale enterprise systems are critical and hard to identify and calculate. This work focuses on analysing changes and their potential impacts, and in particular on how regression testing following such changes can be minimised. The target scope of the approach we describe here is systems containing hundreds of thousands of classes and millions of methods. It is extremely difficult and costly to apply regular regression testing techniques to such systems. It is very expensive and often unnecessary to retest everything after a change is introduced. Selective retesting is dangerous if the impacts of change are not understood, and analysing such systems to understand what is being changed and what the impacts are is difficult. This paper proposes a way to perform a change impact analysis which makes it possible to do efficient, targeted regression testing of enterprise systems. Our approach has been tried on a large system comprising 4.6 million methods with 10 million dependencies between them. Using our approach, maintainers can focus on a smaller, relevant subset of their test suites instead of doing testing blindly. We include a case study that illustrates the savings that can be attained.

Keywords: Large-scale enterprise systems · Impact analysis · Static analysis · Dependency graph

1 Introduction

Enterprise systems are typically large, complicated, and may also be inadequately documented and date back a number of decades. As a consequence they are also often legacy systems: poorly understood and difficult to maintain. To make matters worse, they are often mission critical, being found in critical roles as strategic systems in large companies. So they are typically seen as both critical and fragile.

Patches are supplied by vendors to the underlying middleware, and for a number of reasons may need to be applied. The latest IT Key Metrics Data from Gartner [1] report that in 2011 some 16 % of application support activity was devoted to technical upgrades, rising to 24 % in the banking and financial services sector. Hardware and operating systems change. The user organization develops new or changed requirements that need to be implemented. A perpetual problem for the organization is how to manage such changes with minimum risk and cost.

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

What is needed is a way to identify the *impact* of any change. What business processes might be affected by a patch to the middleware, or by a planned change to the application software, or the way data is stored in the database? If organizations know the possible impact of a change they can select only the relevant regression tests, confident that the others do not need to be run, because the results will not change.

The techniques and tools we have developed are based on static analysis of the code in the system and the code in the patch. Static analysis may be compared with the informal approach of reading the documentation, or with the formal one of dynamic analysis, where the code is instrumented in some way to generate output, logs in particular, that show what it is doing. Documentation, especially for legacy systems, may be incomplete, misleading, or just plain wrong. Dynamic analysis is precise, but essentially incomplete in the same way that testing is, and for the same reason: unless you have run your system with all possible inputs, you cannot know if you have found all possible behaviours. Static analysis looks for possible behaviours, or (in our case) possible dependencies. It typically finds too many, depending on how fine grained the analysis is, and how sophisticated it is, but what it rules out can be ruled out for certain. For example, if method A mentions in its code methods B and C, then we decide that A depends on B and C. It may be that, in practice, A never calls B, only C, but unless our tools can be certain about this, they take the conservative approach of assuming that if either B or C change, A may change. What we can be certain about is that A cannot change just because some other method D changes (unless the recursive analysis of B or C leads us to it). Static analysis is accurate, unlike documentation, because it is the actual system code we analyze, not a description of it. It is complete, unlike dynamic analysis, because it takes the conservative but safe approach outlined.

The approach is conventional. First, we calculate the dependencies between methods and fields in the existing middleware library and user's application, between objects in the database, and between database objects and program methods: *dependency analysis*. Second, we identify what is changed by a patch: *patch analysis*. Third, we calculate, by starting with the things changed in the patch and following the dependencies in reverse, what might be changed when the patch is applied: *impact analysis*.

The things that might be changed, the *affected* methods or methods, can stretch right through the middleware library and the user's application software. In practice, we will only be interested in some of these, usually those in the application software that appear in test cases. Identifying these means the user can identify the regression tests that will need to be applied and hence, as test cases are usually grouped by business process, which business processes may be affected. The methods we select we term the *methods of interest*: the precise way we identify these will depend on how the test cases are organized.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes dependency analysis. Patch analysis is introduced in Sect. 4, and impact analysis in Sect. 5. Section 6 describes a case study. Section 7 summarizes the paper, draws some conclusions, and outlines future work.

2 Related Work

Change Impact Analysis applied to software systems can be traced back to the 1970s. Reasons for doing change impact analysis are well known and understood: "As software components and middleware occupy more and more of the software engineering landscape, interoperability relationships point to increasingly relevant software change impacts." [2] Moreover, due to the increasing use of techniques such as inheritance and dynamic dispatching/binding, which come from widely used object-oriented languages, small changes can have major and nonlocal effects. To make matters worse, those major and nonlocal effects might not be easily identified, especially when the size of the software puts it beyond any maintainer's ability to adequately comprehend.

There is considerable research related to this field, but it seems that there are limited known ways of performing change impact analysis. Bohner and Arnold [2] identify two classes of impact analysis: traceability and dependency. What we are interested in this work is dependency: linkages between parts, variables, methods, modules etc. are assessed to determine the consequences of a change.

Dependency impact analysis can be either static, dynamic or a hybrid of the two. We discuss some of the work using these techniques below.

Static impact analysis [2–6] identifies the impact set - the subset of elements in the program that may be affected by the changes made to the system. For instance, *Chianti* [3] is a static change impact analysis tool for Java that is implemented in the context of the *Eclipse* environment, which analyzes two versions of an application and decomposes their differences into a set of atomic changes. The change impact is then reported in terms of affected tests. This is similar to our approach, but lacks the capability to deal with the database components.

Apiwattanapong et al. [7] argue that static impact analysis algorithms often come up with too large impact sets due to their over conservative assumptions: the actual dependencies may turn out to be considerably smaller than the possible ones. Therefore, recently, researchers have investigated and defined impact analysis techniques that rely on dynamic, rather than static, information about program behaviour [8–11].

The dynamic information consists of execution data for a specific set of program executions, such as executions in the field, executions based on an operational profile, or executions of test suites. [7] defines the dynamic impact set to be the subset of program entities that are affected by the changes during at least one of the considered program executions. *CoverageImpact* [8] and *PathImpact* [12] are two well known dynamic impact analysis techniques that use dynamic impact sets. *PathImpact* works at the method level and uses compressed execution traces to compute impact sets. *CoverageImpact* also works at the method level but it uses coverage, rather than trace, information to compute impact sets. Though the dynamic approach can make the analysis more efficient, it doesn't guarantee that all system behaviors can be captured by it. Thus it might cause a good number of *false negatives*, i.e. potential impacts that are missed.

Recently, a hybrid of static and dynamic analysis is being investigated. [13] proposes a hybrid technique for object-oriented software change impact analysis. The technique consists of three steps: static analysis to identify structural dependencies between code entities, dynamic analysis to identify dependencies based on a succession relation derived from execution traces, and a ranking of results from both analyses that takes into account the relevance of dynamic dependencies. The evaluation of this work showed it produced fewer false negatives but more false positives than a precise and efficient dynamic tool *CollectEA* [7].

The *Program Dependency Graph* (PDG) [14] and associated *Slicing* [15] techniques work at the statement level. We need to work at the level of methods and fields because of the size of the program being analyzed. Rothermel and Harrold [16] identified two kinds of *Regression Test Selection* (RTA) techniques: *minimization* and *safe coverage*. *Minimization* selects minimal sets of tests through modified or affected program components, while *safe coverage* selects every test in the test suite that may test the changed parts of the program. According to the definition, our work is a *safe coverage* approach. In our problem domain, precision is less important than safety.

The size of the program we want to analyse is a major factor driving our approach. The sizes of the software systems most current impact analysis associated techniques [8,17,18] are dealing with are orders of magnitude smaller than the enterprise systems we have targeted. Taking *DEJAVOO* [19] as an example, the largest system analyzed in the empirical study was JBOSS, which contains 2400 classes. As the two-phase process takes a considerable time to complete, systems of the size we are concerned with were clearly beyond the scope of that research. We have not found any related work that claims to handle such large systems.

Our approach is based on static analysis (a) because of the feasibility issue for large programs and (b) because we consider false negatives (missed impacts) much more dangerous than false positives (identified impacts which in practice cannot occur). The very large size of the middleware component means that in practice only small parts of it are likely to be exercised by a particular user, and so even a coarse analysis can produce dramatic savings in regression testing. In addition, dynamic analysis requires run-time information from running test suites and/or actual executions, which may not be available, may be very expensive to produce, and may interfere with the execution and so produce spurious results.

3 Dependency Analysis

We consider a system having, typically, three layers: (i) the user's application code, (ii) the middleware library, and (iii) a database. Our intention is to construct a *dependency graph* which we can use to calculate the potential impact of a change within such a system. In general, if item A refers to item B, then a change to B has a potential effect on A: A is *dependent* on B. A and B might be database tables, other database objects such as stored procedures or triggers, or methods or fields in the application code or library. In practice we divide this analysis into three parts, which we will discuss in turn. *Program dependencies* are between methods and/or class fields within the application code and library (which we analyse as a single program). *Program-database dependencies* are between the program and the database. *Database dependencies* are between database objects.

3.1 Program Dependencies

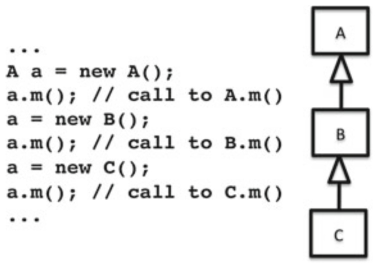
The system for which we have so far developed tools is written in Java, so we will use Java terminology and discuss the particular problems an object-oriented language supporting dynamic binding introduces.

Method invocations are candidates for dynamic binding, meaning that compile time calling of a method might cause calling of another method at runtime, due to class inheritance, interface implementation and method overriding. We illustrate the dynamic problem with two examples. Consider the situation in Fig. 1 where classes B and C override class A's method $m()$. Although statically all three calls are to $A.m()$, dynamically they redirect to $A.m()$, $B.m()$ and $C.m()$, respectively.

As a second example, consider Fig. 2 where classes B and C don't override class A's $m()$ method. Here, the compile time call to $C.m()$ redirects to $A.m()$ at runtime.

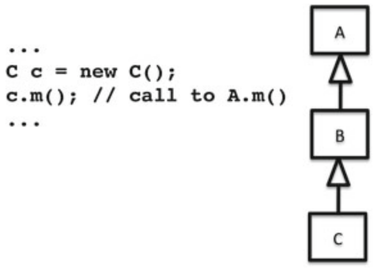
One way to handle dynamic binding statically is to include all classes from the inheritance hierarchy, as in *Class Hierarchy Analysis* (CHA) [20]. The drawback of this approach is the huge number of redundant call edges that might result: it creates an edge from each caller of a method m to every possible instance of m. Consider Fig. 3 where the edges to $C.foo()$ are redundant because only $A.foo()$ and $B.foo()$ have real bodies defined.

Similarly, in Fig. 4 the edge to $B.foo()$ is redundant because A is actually the closest transitive superclass of C that has a body of method $foo()$ defined.



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

Fig. 1. Dynamic Binding Example 1.



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

Fig. 2. Dynamic Binding Example 2.

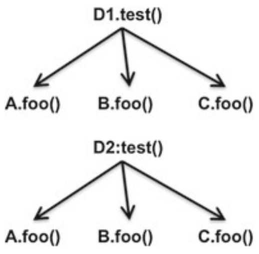
In the java library we analysed there was an interface with more than 50,000 transitive subclasses. If there were, say, 100 callers of a method of this interface, 5 million edges would be generated. In practice we found that only a few dozen of the transitive subclasses would define a particular method,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

Some techniques like *Rapid Type Analysis* (RTA) and *Variable Type Analysis* [20] do exist to tackle this problem and we tried these approaches using the tool Soot [21], but had to abandon it due to excessive memory consumption and memory overflow problems. These approaches turned out be unsuitable for our huge domain. This is one of the reasons we resorted to a new technique which we call *access dependency analysis*.

The full details of access dependency analysis are in a technical report [22], but we illustrate it with our two examples. Consider Fig. 5. The graph shown is the dependency graph resulting from the access dependency analysis of the code shown in Fig. 3a. Note that since $C.foo()$ has no real body of its own, it is not in the graph. We only consider the overridden versions of methods during the addition of extra edges for handling dynamic binding, which reduces the number of edges. Also instead of adding call edges from $D1.test()$ and $D2.test()$ to $B.foo()$, we add an edge from $A.foo()$ to $B.foo()$. What this edge implies is that a compile time call to $A.foo()$ *might* result in a runtime call to $B.foo()$. This kind of edge reduces the number of edges even further because each additional caller only increases the number by one (like the edge from $D2.test()$ to $A.foo()$).

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

(a) Sample code segment



(b) Graph generated

Fig. 3. Conservative Analysis Example 1.

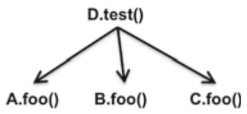
As for the second example, consider Fig. 6 where the graph shown is the dependency graph resulting from the access dependency analysis of the code shown in Fig. 4a. Here, since A is the closest transitive superclass of C for the function $foo()$, a compile time call to $C.foo()$ redirects to $A.foo()$, and we don't include $B.foo()$ in the graph. The result is, once again, a reduced number of edges.

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

As mentioned earlier, a class with over 50,000 transitive subclasses was found to have only a few dozen of them which override a particular method. Using our access dependency analysis, we only get a few hundred edges (rather than almost 5 million edges generated by the conservative analysis). Since the number

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

(a) Sample code segment



(b) Graph generated

Fig. 4. Conservative Analysis Example 2.

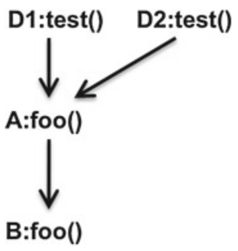


Fig. 5. Access Dependency Analysis Example 1.

of edges are reduced, we also get rid of the memory overflow problem we faced in applying other existing approaches.

3.2 Program-Database Dependencies

Program code interacts with the database via SQL commands generated as strings, and passed to the database through the SQL-API. Such strings are often dynamically created, so we used a string analysis tool, the Java String Analyzer [23] (JSA), which is capable of statically analyzing a section of code and determining *all* the strings which might possibly occur at a given string expression, including dynamically constructed strings. Currently the JSA can only work with Java code, but it is architected in such a way that a single layer

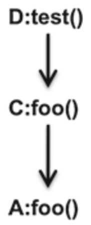


Fig. 6. Access Dependency Analysis Example 2.

of it can be replaced to add support for a different language, leaving the majority of the JSA unchanged.

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

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

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

To solve this, Tarjan's Algorithm [24] was employed to identify all the strongly connected components (i.e. cycles). These components were then compacted into a single node, preserving all of the incoming and outgoing edges of all nodes in the strongly connected component. The modified depth-first search is then run, recording all the possible SQL-related call-chains through the graph.

This approach still has two problems. First, some classes cause the JSA to fail. We built up a list of such classes and adapted JSA to exclude them from the analysis. Second, some strings are effectively just wild cards, because they depend on user inputs, or because of incomplete analysis (the use of call chains instead of complete call trees, and the exclusion of some classes). Again we take the conservative approach, and assume that a wildcard string can be dependent on any database object. Fortunately, the proportion of these in practice seems to be quite small.

3.3 Database Dependencies

The third component of our dependency graph are dependencies between database objects: tables, triggers, procedures, etc. Fortunately, we had a database in which such dependencies were already stored. Otherwise it would be fairly straightforward to write some SQL procedures to calculate them.

4 Patch Analysis

As software evolves, there are incremental changes to an existing, perhaps large, set of code and documentation [25]. Users often *have* to apply vendor patches to potentially fix issues or ensure continuing vendor support.

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

4.1 Database Changes

We need to identify which changes are capable of affecting database objects. Major changes to Oracle databases (for example), come from SQL and PL/SQL scripts. To do this, we employed an SQL parser to capture the names of those objects. We had to extend the original tool to deal with SQL statements that themselves contained SQL definitions.

Other patch files also use SQL for making their changes. In some the SQL is contained in other text, in others it is compiled and has to be decompiled to extract the SQL for the parser. We developed a suite of tools to handle all the relevant files found in Oracle patches.

4.2 Library Changes

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

Some tools and techniques do exist to detect difference between two versions of a program. The Aristotle research group from Georgia Tech. [17] showed an approach for comparing object-oriented programs. Their approach was not

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

(a) Original Code

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

(b) Modified Code

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

(c) XML Segment Describing Difference

Fig. 7. Detecting Modifications

applicable to our domain because it compares both versions of the whole program, rather than making individual class to class comparisons. Moreover, their application domain was several orders of magnitude smaller than ours. Meanwhile, there exist some open source tools like *JDiff* [26], *Jar Comparer Tool* [27], *JarJarDiff* and *ClassClassDiff* [28] which only give API differences between two *class* or two *jar* files. To achieve the level of detail we require, namely which methods and fields are changed, we decided to write our own modification detecting tool for class and jar files. This operates by first converting the Java to XML. We compare the two versions of an XML file, node by node, to detect differences in methods, fields, access flags, superclass, interfaces, etc. and list them in another XML file for use in the impact analysis phase.

As an example, Fig. 7a, b show an original and modified code sample, respectively. Note that the only difference is the inclusion of the call to method $bar()$ inside method $foo()$. The XML segment in Fig. 7c contains this modification information.

5 Impact Analysis

Our aim is to identify a subset of the previously selected *methods of interest*, typically methods appearing in test suites, that are affected by a directly changed method, field, or database object that has been identified by patch analysis (or, perhaps, is a candidate for change by the user).

The overall process, illustrated in Fig. 8, is:

1. We create the program dependency graph. The graph edges are in fact the reverse of the dependency relation, from each method or field to those methods which depend on it, because that is the direction in which we search.

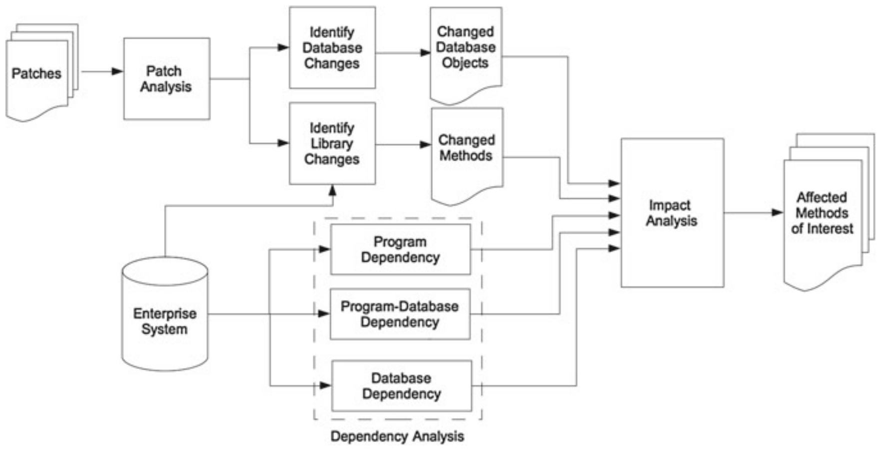


Fig. 8. System Flow Chart.

2. We similarly build the database dependency graph.
3. For each instance of the SQL-API, we use the program dependency graph to identify call chains which terminate in a method of interest, and so create a relation between methods of interest and SQL strings. This is the program-database dependency.
4. We use patch analysis to identify changed methods and fields in the library, and changed database objects. (If impact analysis is being used to investigate the impact of proposed changes by the user to application code and/or database objects, patch analysis is replaced by analysis of the design of the proposed changes to see what existing methods, fields, and database objects are to be changed.)
5 To calculate the dependencies on a database object, we proceed as follows:
   (a) We calculate the reflexive transitive closure of the dependents of our element using the database dependency graph, a set S, say.
   (b) For each element in S we find each SQL string in the program-database dependency that can include its name as a substring, and for each such add the corresponding method of interest to our results.
6. To calculate the dependencies on a program method or field, we simply search the program dependency graph to find all dependents of the method or field, noting any methods of interest that we encounter.
7. The last two steps constitute *impact analysis*. In either case, the result is a subset of the methods of interest.

Impact analysis is illustrated in Fig. 9. In this figure, stars are methods of interest, circles are library methods and fields, and squares are database objects. Directly changed items (methods, fields, or database objects) are black, and potentially affected ones are grey. Unaffected items are white. The dashed lines represent dependencies between items. Note the dependencies may occur not only between items of different layers, but also between items within the same

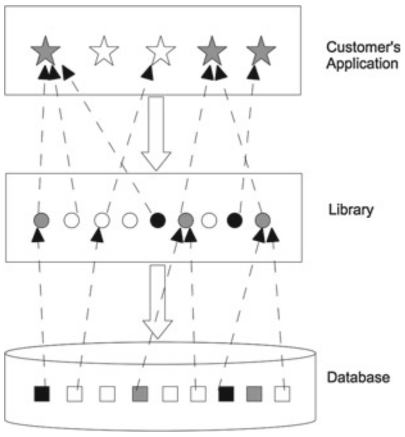


Fig. 9. Impact Analysis.

layer. For simplicity and clarity, we only show dependencies between layers here. We are searching for the grey stars, the potentially affected methods of interest.

6 Case Study

For a case study we took a particular (recent) vendor patch for a commonly used middleware system. The library is written in Java, and contains some 230,000 classes, and over 4.6 million methods. The database as supplied by the vendor contains over 100,000 objects: tables, triggers, procedures, etc.

The calculation of the program dependency graph took over 7 h on a quad core 3.2 GHz machine with 32 GB RAM running 64-bit Linux. This is a little above the average desktop, but by no means a supercomputer. The time is large, but quite manageable, especially as this analysis is independent of any patch or proposed change, and can be prepared in advance. This graph forms a substantial corporate asset for other kinds of analysis, and can be easily and quickly updated as the system changes, provided we do the proper analysis of the changes. The dependency graph has over 10 million edges. Searching this dependency graph takes only a few seconds for each starting point method or field.

The patch contained 1,326 files with 35 different file types. Among those 35 file types, 11 can possibly affect either the library or the database, or both. We ran our tools on each file with one of these 11 types and identified 1,040 directly changed database objects (to which database dependency analysis added no more) and just 3 directly changed Java methods.

The program-database dependency approach described in Sect. 3.2 found that 19,224 out of the 4.6 million methods had SQL-API calls, and that 2,939 of these methods (just over 15 %) had a possible dependency on one of the 1,040 affected database objects.

We adopted as our definition of “methods of interest” those which were not themselves called by anything else, “top callers”, and there were 33,896 of these, a fraction over 2 % of all top callers. The patch, as might be expected, only affects a tiny part of the library, and we can identify that part, and do so in a short space of time.

7 Conclusions

7.1 Achievement

The achievement of this work is threefold:

- We have developed an improved dependency model for dealing with object-oriented languages like Java that support inheritance and dynamic binding, and shown it to be equivalent (in terms of finding static dependencies) to other techniques that typically create much larger dependency graphs.
- We have demonstrated the practical applicability of the improved model to a very large enterprise system involving hundreds of thousands of classes. Such systems may be perhaps 2 orders of magnitude larger than the systems analyzed by other approaches, so our technique seems to be uniquely powerful.
- We have developed the techniques of string analysis beyond those of the Java String Analyser we started with in order again to deal with large size, and to overcome its inability to deal with some of the classes we encountered.

The last point is typical of the work we have done, in developing existing tools to deal with large size, and in developing our own tools, techniques, and data structures to deal with the magnitude of the problem. This has been above all else an exercise in software engineering.

Change impact analysis is performed in three stages: *dependency analysis* with granularity at the level of method or field for program code and database object (table, trigger, procedure, etc) for the database component. The granularity choice is coarser than examining code at the statement level, or database tables at the attribute or data item level, but enables the technique to be used on large, real systems. Dependency analysis generates dependency relations between methods, fields, and database objects that can be searched. The second stage is *patch analysis*, the identification of changed methods, fields, and database objects. Third, *impact analysis* combines the first two outputs to identify the potentially affected *methods of interest* in the user application. If the *methods of interest* are chosen to be those methods appearing in test cases, then we can identify a subset of the regression tests that need to be rerun after a change. Current indications from our case study are that these subsets may be comparatively small, giving a consequent substantial reduction in the considerable costs, resources and time involved.

The analysis is conservative: while we would like to reduce false positives, we are determined that there will be no false negatives, i.e. potential impacts that remain undetected. We can not tell our users that running the reduced set of tests our analysis generates will find any problems caused by the changes: that will depend on the quality of their test sets. But we can tell them that running any more of their test sets will be a waste of resources.

7.2 Future Work

Organizations tend to identify their test suites by the business process that is being tested, and to think of their system as consisting of (or supporting) business processes rather than code classes. HP Quality Centre, for example, organizes tests by business process. By analysis of test cases we will be able to relate the affected methods of interest to the business processes that might be affected, and hence present results in a way that is more meaningful to testing departments.

In the medium term, there are a number of other related applications that can be achieved with the techniques we have developed. First, we need to extend the work beyond the current Java tools, to systems written in other languages such as COBOL. The modular design of our system, especially an analysis based on XML, means that only language-dependent front ends would be needed for each such extension.

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

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

Acknowledgements. The authors are grateful to Ron Mison, Sundaram Viswanathan, and Vinayak Viswanathan of Legacy Systems International, for introducing us to the problem, and also for working with us to ensure that our research produces practical methods and tools. We also thank John Hatcliff for his advice and for pointing us to relevant work, and Wolfram Kahl for his technical advice throughout the project.

References

1. IT Key Metrics Data 2012. Gartner, Inc., December 2011
2. Bohner, S.A.: Software change impact analysis. In: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering, Workshop (SEW-27'02) (1996)
3. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. SIGPLAN Not. **39**, 432–448 (2004)
4. Pfleeger, S., Atlee, J.: Software Engineering: Theory and Practice. Prentice Hall, Englewood Cliffs (2006)
5. Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Softw. **25**, 22–29 (2008)
6. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: Proceedings of the 4th India Software Engineering Conference, ISEC '11, (New York, NY, USA), pp. 99–102. ACM (2011)
7. Apiwattanapong, T.: Efficient and precise dynamic impact analysis using execute-after sequences. In: Proceedings of the 27th International Conference on Software Engineering (2005)
8. Orso, A., Apiwattanapong, T., Harrold, M.J.: 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, vol. 28(5), September 2003
9. Breech, B., Danalis, A., Shindo, S., Pollock, L.: Online impact analysis via dynamic compilation technology. In: 20th IEEE International Conference on Software Maintenance (2004)
10. Patel, C., Hamou-Lhadj, A., Rilling, J.: Software clustering using dynamic analysis and static dependencies. In: 13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09, pp. 27–36. March 2009
11. Li, H.: Dynamic analysis of object-oriented software complexity. In: 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), pp. 1791–1794, April 2012
12. Law J., Rothermel, G.: Incremental dynamic impact analysis for evolving software systems., In: Proceedings of the 14th International Symposium on Software, Reliability Engineering (2003)
13. Maia, M.C.O., Bittencourt, R.A., de Figueiredo, J.C.A., Guerrero, D.D.S.: The hybrid technique for object-oriented software change impact analysis. In: European Conference on Software Maintenance and Reengineering, pp. 252–255 (2010)
14. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**, 319–349 (1987)
15. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. SIGPLAN Not. **19**, 177–184 (1984)
16. Rothermel, G., Harrold, M.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. **22**, 529–551 (1996)
17. Apiwattanapong, T., Orso, A., Harrold, M.: A differencing algorithm for object-oriented programs. In: Proceedings of the 19th International Conference on Automated Software Engineering 2004, pp. 2–13. September 2004

18. Canfora, G., Cerulo, L.: Impact analysis by mining Software and Change Request Repositories. In: IEEE International Symposium on Software Metrics, pp. 20–29 (2005)
19. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. SIGSOFT Softw. Eng. Notes **29**, 241–251 (2004)
20. Bacon, D., Sweeney, P.: Fast static analysis of C++ virtual function calls. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN Notices, vol. 31, pp. 324–341. ACM Press, New York, October 1996
21. Lam, P., Bodden, E., Lhotak, O., Lhotak, J., Qian, F., Hendren, L.: Soot: A Java Optimization Framework. Sable Research Group, McGill University, Montreal, Canada, March 2010. http://www.sable.mcgill.ca/soot/
22. Chen, W., Iqbal, A., Abdrakhmanov, A., George, C., Lawford, M., Maibaum, T., Wassyng, A.: Report 7: Middleware Change Impact Analysis for Large-scale Enterprise Systems. Tech. Rep. 7, McMaster Centre for Software Certification (McSCert), September 2011
23. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003). http://www.brics.dk/JSA/
24. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
25. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Tech. J. **5**(2), 169–180 (2000)
26. Doar, M.B.: JDiff - An HTML Report of API Differences (2007). http://javadiff.sourceforge.net/
27. Jar Compare Tool. http://www.extradata.com/products/jarc/
28. Tessier, J.: The Dependency Finder User Manual, November 2010. http://depfind.sourceforge.net/Manual.html