Identifying Opaque Behavioural Changes

Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1 CANADA
rtholmes@cs.uwaterloo.ca

David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350 USA
notkin@cs.washington.edu

ABSTRACT

Developers modify their systems by changing source code, updating test suites, and altering their system's execution context. When they make these modifications, they have an understanding of the behavioural changes they expect to happen when the system is executed; when the system does not conform to their expectations, developers try to ensure their modification did not introduce some unexpected or undesirable behavioural change. We present an approach that integrates with existing continuous integration systems to help developers identify situations whereby their changes may have introduced unexpected behavioural consequences. In this research demonstration, we show how our approach can help developers identify and investigate unanticipated behavioural changes.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Measurement

Keywords

Static analysis, dynamic analysis, impact analysis, unexpected behavioural change, research demonstration

1. INTRODUCTION

In his 1968 letter, Dijkstra noted that the programmer manipulates source code as a way to achieve a desired change in the program's behaviour; that is, the executions of the program are what is germane, and the source code is an indirect vehicle for achieving those behaviours. He also observed that "our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed" [2, p. 147]. This reasoning led Dijkstra and others to advocate the notions of structured programming [1], in particular the notion of one-in-one-out control structures that allow programmers to more easily reason about classes of behaviours consistently through a single static structure, as well as to compose those classes of behaviours more easily.

Dijkstra's plea to simplify the source-behaviour relationship has, however, been pushed aside over the past four decades, yielding to a number of powerful and useful programming mechanisms that provide developers with a great amount of flexibility but make this relationship more opaque. Examples of such mechanisms abound, for example: event-based programming [6], implicit invocation [5], exception handling [7, 9], and dependency injection [4, 3]. Inasmuch as our "intellectual powers" have not increased significantly and that the source-behaviour relationship has become more opaque rather than less so, programmers are left with relatively little help in identifying unintended behaviours.

In this paper we succinctly describe our approach and how it is implemented (Section 2). We then describe how this approach can be applied to three development scenarios to show how unexpected behavioural changes can be identified (Section 3). A video demonstration can be found online: `http://cs.uwaterloo.ca/~rtholmes/go/icse11demo`.

2. APPROACH

To identify opaque (or inconsistent) changes we contrast the static modifications made by the developer with the dynamic effects of those changes. We consider two program versions: one before the change (the baseline) and one after the change. For each version, we extract a static call graph of the system as well as the dynamic call graph collected by running the system's test suite for each of the versions. The developer can select any pair of versions they wish (e.g., they can compare their current state to the state last night or they can compare any two milestone versions).

Combining two analyses over two versions of the system enables us to create fifteen partitions (plus the empty set). Figure 1 provides a high-level overview of our approach. Partitions are labelled according to the elements they contain. A partition containing elements identified by the static analysis are denoted with s whereas those identified by the dynamic analysis are denoted by d. Elements that were newly detected (as a consequence of a change) are denoted with a $+$ whereas those that were removed are denoted with a $-$. For instance, a newly added method call that is executed by the test suite would be identified as s^+d^+; if the method call was not executed it would be classified as s^+.

We further group the fifteen partitions into four cate-

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

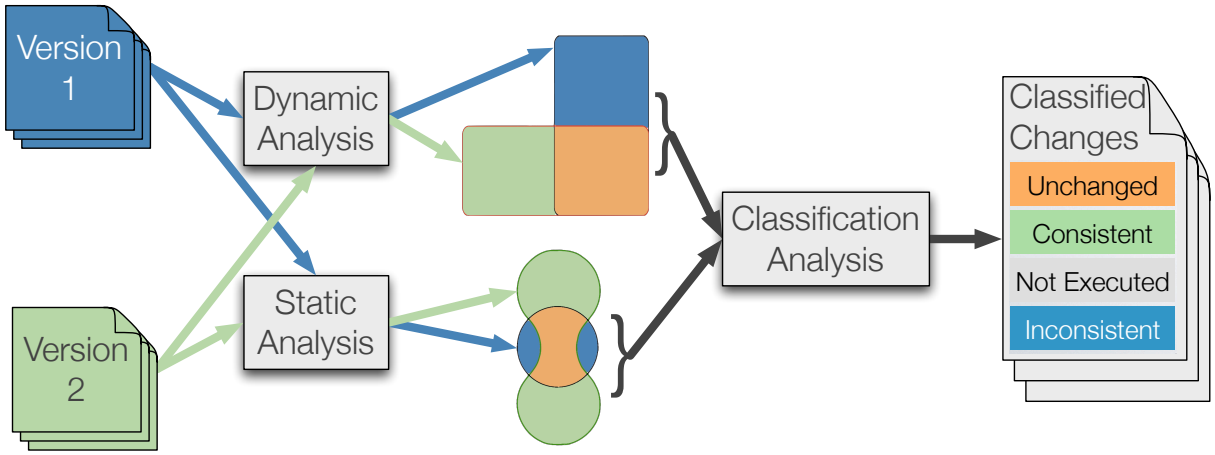

Figure 1: Analysis partitions with descriptive labels and coloured by their categorization.

gories (Figure 1) based on two assumptions: First, developers have a strong understanding of their static changes; second, that developers are only interested in the behavioural consequences of a specific change (or range of changes). We feel the first assumption is valid as developers typically modify their systems statically (e.g., by modifying the source code or its dependencies). The second assumption is simply one of filtering: in asking "have I broken anything by making this change?" the developer is asking about the behavioural differences of a particular change, not the accumulated behavioural differences of all past changes.

These assumptions lead us to believe that developers will be most interested in opaque or inconsistent changes, as these dynamic changes do not directly map to the well-understood static modifications they made to their system. Inconsistencies between partitions can also be interesting (for instance when there are many consistently executing new elements (s^+d^+) but only one that isn't executing (s^+). A more complete description of the approach and the partitions is reported elsewhere [8].

As a consequence of an industrial study [8], we have modified the tool to be integrated with nightly build and continuous integration systems. This mechanism allows both developers and managers view the results to check for unexpected behavioural changes and makes it easier to query between arbitrary program versions. Our approach integrates with `ant`, the most common build system used by large Java-based projects, by adding two new targets to any existing automated build process. The static call graph is built with an extension of `depfinder`[1]. The dynamic call graph is generated by weaving the project's code with a custom-build AspectJ aspect that is also applied through an `ant` target. The results of the static and dynamic analyses are written to XML files that constitute the inputs to our approach.

3. SCENARIOS

We provide three scenarios demonstrating the kinds of changes our approach highlights for developers.

[1] `http://depfind.sf.net/`

3.1 Code Change

We expand on the illustrative scenario provided in our previous paper [8] to show how an inconsistent change can arise. Figure 2(c) provides the partitions we generate after modifying the code in Figure 2(a) to the code in Figure 2(b). In this change, the developer has added a simple cache to their system. The sd and s^+d^+ partitions both contain exactly the elements the developer would expect having made their change; however, the d^+ partition shows something that they may not have expected. When they executed the code and added a (previously cached) value to the cache, a key collision occurred causing the `HashSet.add(..)` to check the equality of the new `LocalType` object to the previously-cached one. In this case, the developer may not have been expecting this callback and might check to ensure his `equals` method is correct.

3.2 Test Change

Sometimes the volume of changes being made can make it difficult for a developer to be sure their change is complete. For example, JodaTime currently has more than 3,500 unit tests in its test suite, each of which is identified by an `@Test` annotation. If the developer adds ten new tests to their system and executes their suite these will all appear in the s^+d^+ partition, as expected. But if the developer forgets to add an `@Test` annotation to one of the new tests, nine of them will appear in s^+d^+ and one will appear in s^+; in this case the inconsistency between these two partitions is interesting. This kind of error is easy to make in practice unless the developer expects a failure when they added the test or were keeping explicit track of the number of tests executing before their change and the number of tests that executed after the change.

3.3 Environmental Change

Developers changing the execution context for their system often want to know whether their system behaves consistently for different environmental configurations. For example, they might want to upgrade some version of a library or framework they use, or they might want to upgrade the version of the Java (JDK) their system executes with. In these cases, the static structure of the system is held constant (unless the upgrade caused some compilation errors

```
void genStore() {
    int val = compute();
    ...
}
```

(a): Original code.

```
Collection _collection = new HashSet();
void genStore() {
    int val = compute();
    cache(val);
    ...
}

private void cache(int val) {
    LocalType lt = new LocalType(val);
    _collection.add(lt);
}
```

(b): Modified code.

s^+d^+

```
<clinit> → HashSet()
genStore() → cache(int)
cache(int) → LocalType(int)
cache(int) → Collections.add(int)
```

sd

```
genStore() → compute()
... → ...
```

d^+

```
Collections.add(int) → LocalType.equals(Object)
```

(c): Partitions arising from change.

Figure 2: Inconsistent change example.

the developer needed to resolve) and only behavioural differences are of interest.

As a concrete example, the developer of JodaTime wanted to check to make sure his system behaved the same for JDK6 as it did for JDK5. Interestingly, upon applying our tool he was presented with a single element in both the sd^+ and sd^- partitions. The elements in each of these partitions were statically unchanged but as a consequence of the JDK upgrade one started executing and another stopped executing. Specifically, our approach highlighted the two method calls listed in Figure 3(a). Figure 3(b) shows the code corresponding to the difference; in JDK6 one method is called by reflection whereas in JDK5 a different method is explicitly called. While this difference may not be significant, it might be useful for a developer to understand that the behaviour of their system is not identical for both JDK5 and JDK6 if they encounter future problems with this functionality with users who are using alternate JDKs.

4. CONCLUSION

In this paper we described three concrete scenarios whereby our approach can help developers understand the behavioural consequences modifications to their systems. By highlighting inconsistent changes for developers, we aim to complement their static understanding of their systems by providing additional focused insight into how the dynamic behaviour of their system has changed. Our approach integrates into existing nightly build systems to enable teams to easily view information about inconsistent changes without altering the system being examined or installing any new tools on individual developer's machines.

sd^-

```
org.joda.time.DateTimeUtils.getDateFormatSymbols(Locale) →
 java.text.DateFormatSymbols(Locale)
```

sd^+

```
org.joda.time.DateTimeUtils.getDateFormatSymbols(Locale) →
  java.lang.reflect.Method.invoke(Object, Object[])
```

(a): Partitions arising from JDK upgrade.

```
DateFormatSymbols getDateFormatSymbols(Locale locale) {
    try {
        Method method = DateFormatSymbols.class.
            getMethod("getInstance", new Class[] Locale.class);
        return (DateFormatSymbols) method.
            invoke(null, new Object[] locale);
    } catch (Exception ex) {
        return new DateFormatSymbols(locale);
    }
}
```

(b): Code snippet causing JDK difference.

Figure 3: Environmental change code and partitions.

Acknowledgments

We wish to thank our industrial partner for providing access to their source code and build environment and Rylan Cottrell and Yuriy Brun for their insightful comments.

5. REFERENCES

[1] O.-J. Dahl, E. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press, 1972.

[2] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[3] M. Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html. "Last significant update: 23 Jan 04".

[4] M. Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, Jul/Aug 2001.

[5] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 447–455, 1993.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[7] J. B. Goodenough. Structured exception handling. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 204–224, 1975.

[8] R. Holmes and D. Notkin. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 10 pages, 2011.

[9] B. G. Ryder and M. L. Soffa. Influences on the design of exception handling: ACM SIGSOFT project on the impact of software engineering research on programming language design. *SIGPLAN Notices*, 38(6):16–22, 2003.