Identifying Program, Test, and Environmental Changes That Affect Behaviour

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 evolve a software system by changing the program source code, by modifying its context by updating libraries or changing its configuration, and by improving its test suite. Any of these changes can cause differences in program behaviour. In general, program paths may appear or disappear between executions of two subsequent versions of a system. Some of these behavioural differences are expected by a developer; for example, executing new program paths is often precisely what is intended when adding a new test. Other behavioural differences may or may not be expected or benign. For example, changing an XML configuration file may cause a previously-executed path to disappear, which may or may not be expected and could be problematic. Furthermore, the degree to which a behavioural change might be problematic may only become apparent over time as the new behaviour interacts with other changes.

We present an approach to identify specific program call dependencies where the programmer's changes to the program source code, its tests, or its environment are not apparent in the system's behaviour, or vice versa. Using a static and a dynamic call graph from each of two program versions, we partition dependencies based on their presence in each of the four graphs. Particular partitions contain dependencies that help a programmer develop insights about often subtle behavioural changes.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Tools and Techniques
D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Measurement

Keywords

Static analysis, dynamic impact analysis, comparative analyses, software behaviour

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA

1. INTRODUCTION

When developers make a change they focus on two primary questions: "Did I successfully implement the intended change?" and "Did I break anything else?" Most variations in behaviours between the original and modified executables provide evidence useful for answering these questions. Confidence that a change performs as expected may, for example, be increased when tests for the newly added feature pass. Concern that a change may have inadvertently broken something is raised, for example, when a regression test fails.

Some behavioural variations, however, may not be easily isolated or identified as pertinent to either question. Such a variation may be benign, may represent a subtle bug, or may only manifest as benign or buggy over time as it interacts with other behaviours. Our approach intends to help developers more easily identify specific variations of this kind, allowing them to thoughtfully decide whether the variations of immediate or long-term concern.

A developer can evolve a software system's behaviour in at least three ways: changing the program source code, modifying the test suite, and altering the environment. Given a set of changes in one or more of these dimensions, we extract both a static and a dynamic call graph from each of two program versions, and partition the calls based on their presence in each of the four graphs. Some partitions contain dependencies that are unlikely to concern the developer; for example, if new tests are added alongside new methods in the program source, these will be captured in the partition that contains pairs that are not observed in the original version but are observed both statically and dynamically in the new version. Other partitions isolate pairs that likely deserve more scrutiny; for example, a call that appears dynamically without any corresponding change to the program source may occur because of a change to a configuration file that is not apparent in the source code.

Our contributions include:

- Defining and implementing a simple but novel mechanism for partitioning program changes in terms of static and dynamic method calls.
- Identifying which of these partitions represent consistent, inconsistent, unchanged, unlikely and unexecuted changes to the program's dependencies; the inconsistent category represents those changes that likely deserve deeper developer attention. By focusing on a developer's current change, rather than the aggregation of all past changes, we are able to return a small subset of pertinent dependencies for the developer to examine.

- Evaluating our approach over 10 versions of three different open source systems, demonstrating various quantitative and qualitative properties of our approach. For example, we show that fewer than 1% of the extracted pairs are of likely interest to the programmer with respect to behavioural changes.
- Applying our approach to an industrial code base for one complete development sprint. The limited feedback we received suggests that the behavioural information can provide a useful alternative view for tracking system evolution. Our industrial partner valued the information that our approach provided enough to request that it be incorporated into their existing nightly build system.

Section 2 details our approach for defining these partitions and categorizations; it also provides a concrete scenario to illustrate our partitioning approach. Section 3 presents the static and dynamic extraction tools we use, and how we reconcile static and dynamic calls with one another. Section 4 reports on application of our approach to three open source systems, describes our initial industrial application, and discusses threats to validity. Section 5 covers key related work, and Section 6 concludes.

2. APPROACH

We model a program's structure using call graphs denoting a program's methods and the calls between them. We extract four dependence graphs: a static call graph from before and after a change ($V1S$ and $V2S$) — each of these caller/callee pairs is *statically observed* — and a dynamic call graph from before and after the same change ($V1D$ and $V2D$) — each of these pairs is *dynamically observed*. This classification depends on the specific analyses that are used; our prototype uses the lightweight analysis tools described in more detail in Section 3. Our exposition generally focuses on source code changes, but our approach does not restrict the kinds of changes the developer can make: in addition to the source code, we also explicitly address changes to test suites and the environment of the program. Like all other approaches, our approach cannot provide guarantees about how a source change will effect subsequent executable behaviour. We believe it to provide a complementary perspective that developers can use to help determine if a static change is likely to affect the program's runtime execution in unforeseen and perhaps troublesome ways.

We use static dependencies to approximate some dimensions of programmer expectation. In particular, we consider changes to the static call dependencies as expected changes, because the programmer explicitly modifies the program source. For example, if a programmer adds a method call and a corresponding test to a program, we consider it unsurprising that there are new corresponding static and dynamic call dependencies in the modified program. Thus we model the notion of unforeseen consequences of a source change in terms of static and dynamic call dependences. We hypothesize that variations in the dynamic dependences between two program versions suggest unforeseen behaviours when related changes are not found in the static dependences. That is, when a programmer makes a change, some set of dynamic dependences may be expected to appear: if those behaviours do not appear, or if other apparently unrelated behaviours appear or disappear, then the programmer should consider those deviations in more depth.

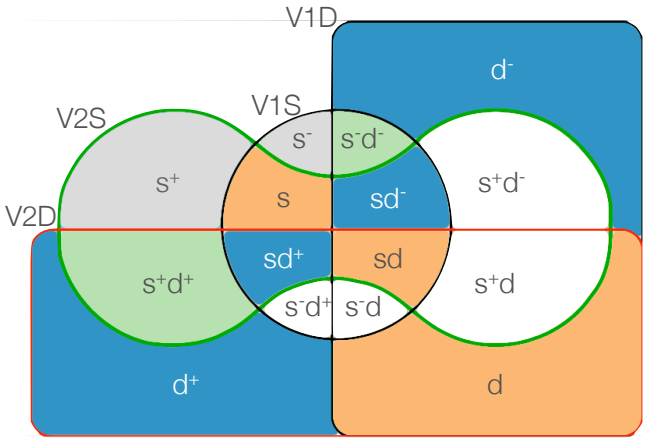

Figure 1: Analysis partitions with descriptive labels and coloured by their categorization.

From these four graphs, we compute all set intersections, as shown in Figure 1 using a standard four-set Venn diagram. The circle in the centre represents the statically observed pairs from the first version ($V1S$), the barbell-shape represents the statically observed pairs from the second version ($V2S$), the vertically-oriented rectangle on the right represents the dynamically observed pairs from the first version ($V1D$), and the horizontally-oriented rectangle at the bottom represents the dynamically observed pairs from the second version ($V2D$).

Each partition containing at least one statically observed pair is marked

- with an s if its members are statically observed in both versions,
- with an s^+ if its members are not statically observed in the first version but are in the second, and
- with an s^- if its members are statically observed in the first version but not in the second version.

Analogously, a partition containing at least one dynamically observed pair is marked with d, d^+ or d^-. By convention, we mark the static property of the partition, if any exist, followed by the dynamic property of the partition, if any exist. The partition containing no pairs from any of the four graphs is uninteresting and has no label.

For example, partition s^- includes only those pairs that were statically observed in the initial version ($V1S$) but not in the modified version ($VS2$) and that were not dynamic observed in either version ($V1D$ and $V2D$). Similarly, partition s^+d^+ contains only those pairs extracted statically and dynamically in the second version but not extracted by either analysis in the original version.

Illustrative scenario.

A developer adding caching functionality to an application might structure code as shown in Figure 2(a), with the original source code shown in orange and the new code shown in green. The statically observed pairs, the dynamically observed pairs, and the combination of those pairs are shown in Figures 2(b), (c) and (d), respectively.

The three method calls the developer added appear in partition s^+d^+; the developer would likely be more surprised at their absence than at their presence, because the calls were explicitly added to source code that was exercised by the test suite. If, however, one of the new pairs was not observed dy-

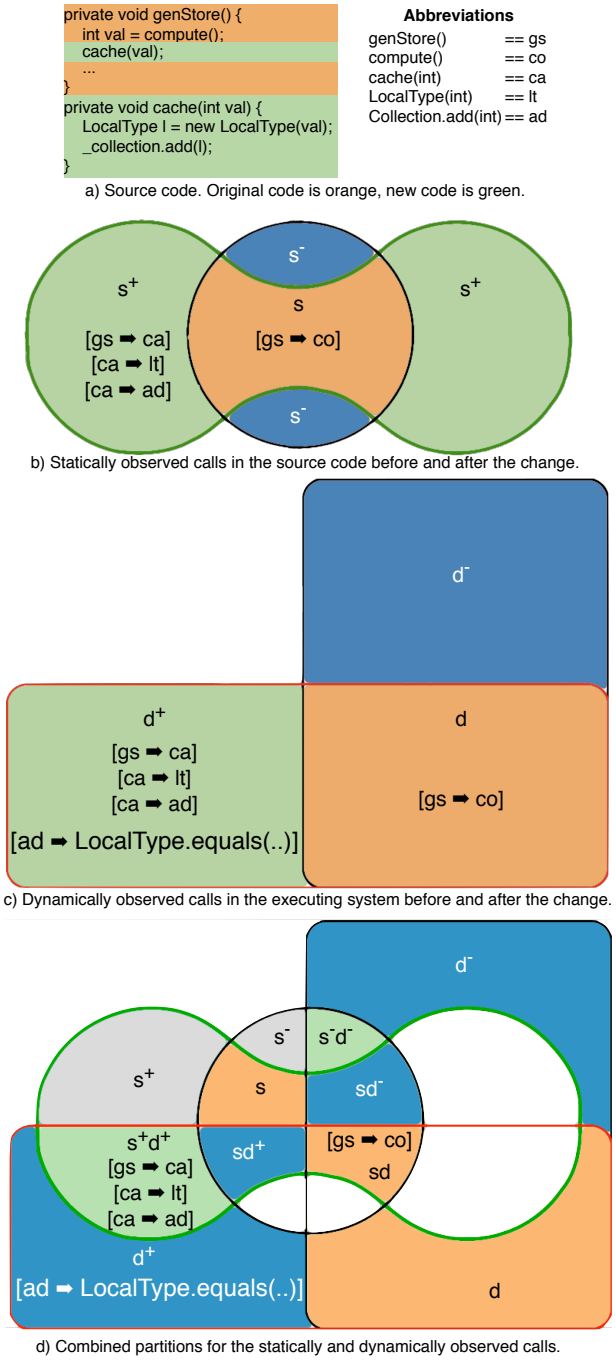

Figure 2: Scenario partitions.

namically — for instance, an exception might interrupt the computation — it would instead appear in partition s^+.

Partition d^+ contains a dynamically observed callback from `Collections.add(...)` to `LocalType.equals(...)`; this pair is executed when the system tries to add a `LocalType` object to the cache that has been previously cached. This causes a key collision since the cache is a `HashSet`, requiring `HashSet` to check the equality of the two `LocalType` objects. By looking at both versions of the source code, the developer would likely not expect this method call to occur. A dynamic differencing approach would identify this element but the developer would be responsible for discriminating the one d^+ callback from pairs appearing in s^+d^+.

2.1 Categorizing Dependency Partitions

We categorize the above partitions based on two related assumptions. First, we assume that a developer modifies code with clear intention. For example, inserting a method call carries with it the expectation that it will sometimes be executed in the modified version. Second, a developer generally focuses on the part of the program relevant to the intended change, rather than trying to understand the program in its entirety [4, 11, 23].

Based on these assumptions, we group the 15 partitions into five categories: INCONSISTENT, CONSISTENT, NOT EXECUTED, UNCHANGED and UNLIKELY. Figure 1 shows the partition assignment, colouring the categories blue, green, grey, orange and white, respectively. While this categorization of the partitions make the most sense in terms of the source code changing, it is also pertinent when the test suite changes or the environment is altered.

Inconsistent.

Dependencies in the INCONSISTENT partitions (d^+, d^-, sd^+, and sd^-) represent divergences between statically and dynamically observed pairs. A method call appearing in d^+ represents a pair that became dynamically observed after a change was made, even though a corresponding call was not statically observed in the code before or after the change. Conversely, a method call found in d^- represents a call no longer observed dynamically even though no corresponding method call was statically removed from the source code. A method call appearing in sd^+ represents a newly observed invocation without a corresponding static change. This can happen, for example, if a change in the program's control flow — or the addition of a test case — allows pre-existing but previously unexecuted code to execute. Partition sd^- represents a statically observed pair that remains unchanged across versions but where it is no longer dynamically observed in the second version.

We posit that the partitions in this category are the most likely to capture unforeseen behavioural changes. If the test suite were changed sd^+ and sd^- would represent previously-existing method calls that the tests either newly exercise or cease to exercise; while these elements may not be inconsistent in this case, the developer is still relieved from the task of differentiating them from s^+d^+ and s^-d^-. Environmental changes (e.g., changing a non-source resource) can sometimes be detected through new relationships appearing in the d^+ and d^- partitions; in these cases, as the source code itself is held constant, these partitions contain the dynamic differences between two executions that are not statically obvious.

Consistent.

Dependencies in the CONSISTENT partitions (s^+d^+ and s^-d^-) represent changes that are coherent in their static and dynamic representations. s^+d^+ represents the static addition of new method calls that dynamically execute in the second version. Conversely, s^-d^- represents a method call that was deleted and whose corresponding executions disappeared after the change was made. These changes can be made to either the program source or its test suite.

Not executed.

Dependencies that are NOT EXECUTED (s^+ and s^-) represent method calls that were statically added or removed from the source but were not observed dynamically in either ver-

sion. This might arise if, for instance, if a developer added a new JUnit test case but forgot to add the `@Test` annotation to it; in this case, the added calls would appear in s^+, rather than in (s^+d^+). Incongruities between the CONSISTENT and NOT EXECUTED may help a developer who expects a static change to be dynamically corroborated.

Unchanged.

Partitions s, d, and sd represent pairs that were UNCHANGED between the two versions. Pairs observed consistently (statically, dynamically, or both statically and dynamically) before and after a change are likely to be unsurprising to a developer. The overwhelming majority of the dependencies in the system fall into these partitions (see Section 4).

Unchanged partitions can be viewed as aggregating past changes across a sequence of versions. That is, once a pair appears in an unchanged partition, it will remain there across future changes unless that pair is modified again. For example, a method call added and executed in the fifth version of a program would appear in s^+d^+, but an analysis of the sixth version would find this same fact appearing in sd unless that call was changed again.

Unlikely.

The UNLIKELY partitions (s^-d^+, s^-d, s^+d, and s^+d^-) represent states that are highly unlikely (or not possible) given our analysis tools (described immediately below) and thus will likely never be populated. For example, it would be bizarre to statically delete a call that is only dynamically observed *after* the deletion (s^-d^+). We have not observed these UNLIKELY partitions in practice.

2.2 Partitioning Non-Source Changes

Our approach can also be used to detect behavioural differences brought about by modifications to libraries or non-code resources. Developers can use our approach to hold their source code constant and change their environment. In these cases, the CONSISTENT and NOT EXECUTED partitions would always be empty because the source would not have changed. Elements in the INCONSISTENT partitions would be especially interesting because they would represent behavioural changes in an environment the developer would likely expect there to be none. For example, a developer updating a third-party library expecting their system to behave the same would be surprised to find dependencies in sd^+ or sd^-; these dependencies would indicate that the control flow of their program has changed as a consequence of the library update. (Section 4 shows a real example of this when the system is run on two different JDKs.) As another example, if the developer changes a configuration XML file, they may find dependencies appearing in the d^- partitions as some callbacks stopped executing as a consequence of the metadata change.

2.3 Expressiveness

By differencing two dynamic and and two static analyses our approach provides developers with an expressive set of partitions, enabling them to interpret each partition as appropriate for their task. Consider a pair in the d^+ partition in a dynamic-only approach: all the developer would know was that "there is a new method call executing." In contrast, in our approach the developer can differentiate between "there is a new method call being executed that I explicitly added" (s^+d^+), "a method call that was previously written is now being executed" (sd^+), and "an unexpected method call that is now being executed" (d^+); this resolution allows them to focus on only those elements that they are interested in and easily ignore the rest. Each of our partitions can be directly related to a developer's day-to-day development activities. Greater partitioning allows developers to focus their attention on the partitions relevant to their task.

3. DEPENDENCY GRAPH GENERATION

Our implementation uses dependence graphs that contain nodes that represent methods and edges that represent calls between methods.

Static analysis.

We generate the static dependence graph using Robillard's Eclipse-based JayFX tool with the class-hierarchy analysis option disabled.[1] JayFX reports static dependencies for a given project but does not consider external library code. As such, its results approximate what a developer might generate through a manual code inspection of their own project's code.

This static analysis is approximate: not all calls that can arise at run-time are reported, and not all calls that are reported can be executed at run-time. For realistic systems, this notion of approximation is pervasive. Few if any static analysis tools for widely-used programming languages report all possible calls: common stumbling blocks include event-based invocation, calls through the reflection interface, calls made based on XML-descriptions that link middleware layers, calls that arise through external libraries that are hard to analyze, etc.

Even if provided with a genuinely sound static analysis, the categorization of pairs would remain stable. The soundness of the analysis would ensure that partitions d, d^+ and d^- would be empty. Any pairs that another static analysis would have missed would shift into the corresponding partitions sd, sd^+ and sd^-, which appear in the same category as d, d^+ and d^-, respectively.

Dynamic analysis.

Our prototype dynamic graph is generated using a custom tracer written using AspectJ.[2] The tracer maintains a call stack as the system executes and creates a method call relation at every call site as the program executes. The tracer is not applied to any call within an external library; this means that edges to a library (e.g., `myMethod()`→`HashSet.add(..)`) are traced and this may result in another call edge (e.g., `HashSet.add(..)` → `LocalType.equals(..)`), even if `add(..)` does not call `equals(..)` directly; from the developer's point of view, these alternatives are equivalent. The dynamic traces are generally collected by running a test suite; the developer could use our approach to compare the execution of a single test, the system's entire test suite, or any arbitrary execution of the system.

Reconciling analyses.

Matching elements between analyses is done by comparing the signatures of the caller and the callee for each method call. For like analyses this is straightforward; between static and dynamic we perform some straightforward signature ma-

[1] http://www.cs.mcgill.ca/ swevo/jayfx/
[2] http://eclipse.org/aspectj/

nipulations to ensure the signatures align correctly. Reconciling the analyses and constructing the partitions is linear in the size of the program being analyzed (consisting mainly of simple set differencing).

Our current signature matching approach is brittle with respect to type hierarchies. For example, JayFX might extract a call to `new Vector(Collection)`, whereas dynamically our dynamic tracer could detect this as `new Vector(ArrayList)`. While this misregistration would cause some calls to appear in the wrong partitions (e.g., in this case one call in s and one call in d, instead of one call in sd), we have not yet fixed this problem because these three partitions are still in the same UNCHANGED category.

4. EVALUATION

Our evaluation has two parts. First, we applied our approach to three existing open source software systems to see how our approach partitioned behavioural changes. Second, we ran our prototype tool on an industrial code base for one complete development sprint as a basis for an industrial perspective on our approach.

4.1 Retrospective Evaluation

We retrospectively applied our approach pairwise to ten consecutive versions of three existing systems. The intent of this evaluation was (1) to see if the partitions of interest were small enough to allow developers to reasonably study the actual dependencies, and (2) to qualitatively examine the dependencies to see if they were useful and non-obvious. In this study, we focused on a per-commit granularity. (In the next section, we instead consider end-of-day versions.)

Each of the three systems we used had open repositories and some form of a test suite. The open repository requirement allowed us access to past versions at per-commit granularity. The test suite requirement allowed us to observe dynamic calls without constructing (potentially biased) tests of our own. The three systems we evaluated were the Google Visualization Data Source Library, JodaTime, and the Google RFC 2445 Library.[3] Table 1 provides basic information about the most recent version of each system we analyzed.

Project	KLOC	# Tests	Last Version
Visualization	17	365	30
JodaTime	76	2,525	1396
RFC 2445	7	171	22

Table 1: Evaluation systems, indicating their size, number of tests, and the final version we used in our analysis.

4.1.1 Methodology

We selected the 10 most recent change sets from each project's source repository that involved committing a modification to a source code file; a few changes to documentation-only files were ignored. To analyze the effects of each commit we compared all consecutive pairs of versions for each of the systems; that is, we considered 27 changes across three systems. Ten of these changes involved fixing a bug and updating a test, three involved adding a feature and updating a test, two involved fixing a bug without a test, three involved adding a feature without a test, two changed source code documentation, and the rest were simple refactorings or code cleanups.

We extracted the static dependence graph from each version and collected the dynamic dependence graph by running each project's entire test suite (using the tools described in Section 3). The external environment — libraries and runtime environment — for every version of the same system was held constant. The UNLIKELY partitions were always empty as expected (and thus we do not discuss them further).

4.1.2 Quantitative Results

Key results for the pairwise evaluation are in Table 2, which shows the number of dependencies in key partitions.[4] The table has three horizontal sections, arranged by project: the first column of each project's section indicates the check-in identifier for each associated commit.

For each of these projects, the table has five vertical sections; the first describes the number of edges in the graphs captured by the four analyses. Each category described previously is displayed in the next four vertical sections. The sections are ordered in terms of their potential interest to the developer.

The INCONSISTENT partitions were non-empty for 16 of the 27 pairs of versions we investigated. In total, we identified 84 unexpected behavioural changes, an average of 3.1 unforeseen dependencies per program version for a developer to investigate.

The CONSISTENT partitions were non-empty for 15 of the 27 pairs of versions. In total, 235 method calls were identified as being consistent between the static change and their runtime behaviour. The NOT EXECUTED partitions were non-empty for 15 of the 27 pairs of versions. In total, 153 method calls were added or removed but not executed at runtime.

Even if the developer were to consider all of the elements in the INCONSISTENT, CONSISTENT, and NOT EXPECTED partitions, they would only have to examine an average of 17 calls per commit. We do not expect that developers would often, if ever, examine all of these dependencies in practice.

Small sets of dependencies.

By providing developers a means for focusing only on the behavioural effects of their *current* change, we are able to greatly reduce the number of elements they might otherwise have to consider. For the 27 pairs of program versions we analyzed, the UNCHANGED partitions, representing those elements not affected by the current change, aggregated a total of 751,539 dependencies whereas the INCONSISTENT, CONSISTENT, and NOT EXECUTED partitions contained only 472 dependencies (84, 235, and 153 respectively), a 99.94% reduction. The distribution of these aggregated totals is shown in Figure 3. The minimum reduction for any individual program run we analyzed was 97.2% and the average reduction was 99.5%. From the programmers' point of view, this means that the number of dependencies that our approach suggests they look at is manageable in practice.

[3] http://code.google.com/p/google-visualization-java/, http://joda-time.sf.net, http://code.google.com/p/google-rfc-2445/.

[4] For simplicity, we only include counts of edges — dependencies between pairs of program elements — and omit counts of the program elements themselves.

	Edges				Inconsistent				Consistent		Not Executed		Unchanged		
	V1S	V2S	V1D	V2D	d+	d-	sd+	sd-	s+d+	s-d-	s+	s-	s	d	sd
Visualizer															
17→19	7,555	7,583	6,597	6,612	2		4		9		19		2,430	1,476	5,121
19→20	7,583	7,585	6,612	6,614					2				2,449	1,478	5,134
20→21	7,585	7,585	6,614	6,614									2,449	1,478	5,136
21→22	7,585	7,585	6,614	6,614									2,449	1,478	5,136
22→23	7,585	7,596	6,614	6,621		1			8		3		2,449	1,477	5,136
23→24	7,596	7,596	6,621	6,622	1								2,452	1,477	5,144
24→28	7,596	7,597	6,622	6,623					1				2,452	1,478	5,144
28→29	7,597	7,615	6,623	6,641	3	3	2		16		2		2,450	1,475	5,145
29→30	7,615	7,617	6,641	6,642	1						2		2,452	1,478	5,163
RFC-2445															
9→12	2,019	2,021	1,840	1,841	1						6	4	742	567	1,273
12→13	2,021	2,021	1,841	1,840		1					1	1	747	567	1,273
13→15	2,021	2,021	1,840	1,841	1				1	1			748	567	1,272
15→16	2,021	2,060	1,841	1,874	13				22	2	21	2	746	568	1,271
16→17	2,060	2,060	1,874	1,874									767	581	1,293
17→18	2,060	2,086	1,874	1,900	7	2	4		29	12	14	5	758	579	1,281
18→20	2,086	2,088	1,900	1,902	7	6			11	10	15	14	758	580	1,304
20→21	2,088	2,099	1,902	1,909	6	2			26	23	10	2	771	585	1,292
21→22	2,099	2,134	1,909	1,932	10		1		16	4	23		780	591	1,314
JodaTime															
1366→1367	66,343	66,357	30,565	30,581	3				13		1		41,230	5,452	25,113
1367→1374	66,357	66,362	30,581	30,587	1				5				41,231	5,455	25,126
1374→1378	66,362	66,363	30,587	30,587							3	2	41,229	5,456	25,131
1378→1379	66,363	66,363	30,587	30,587									41,232	5,456	25,131
1379→1380	66,363	66,371	30,587	30,597	2				13	5			41,232	5,456	25,126
1380→1381	66,371	66,374	30,597	30,599					4	2	1		41,232	5,458	25,137
1381→1388	66,374	66,374	30,599	30,599									41,233	5,458	25,141
1388→1389	66,374	66,374	30,599	30,599							1	1	41,232	5,458	25,141
1389→1396	66,374	66,374	30,599	30,599									41,233	5,458	25,141

Table 2: Quantitative results demonstrating the sizes of each of the partitions for the 27 pairs of program versions we analyzed.

Compared to a static- or dynamic-only approach.

To provide a quantitative sense of the difference between our approach with a purely static or purely dynamic approach we performed a comparison using the data from Table 2. The static call graphs resulted in 388 pairs (298 in s^+ and 90 in s^-) while our approach only returned 153 pairs for these partitions (122 in s^+ and 31 in s^-), a 61% reduction. The dynamic call graphs resulted in 319 pairs (245 in $d+$ and 74 in d^-); in contrast, our approach only returned 73 pairs for these partitions (58 in d^+ and 15 in d^-), a 77% reduction in the number of elements the developer would have to consider. The majority of the reduction comes from elements being split between d^+ and s^+d^+ (and the splitting of s^- and s^-d^-) by our approach.

4.1.3 Qualitative Results

The quantitative analysis argues that our approach effectively identifies small sets of incongruous dependencies in practice; however, the numbers tell only one part of the story. Our approach attempts to provide pertinent information to the developer about the source-behaviour relation as they make changes to their source code. By identifying specific dependencies in each partition, locating the source code associated with each dependence is straightforward, enabling the developer to quickly determine if an element is indeed interesting. The kinds of insight a developer would consider interesting depends on their role and the nature of their change. In this section, we examine key partitions from our experiment describing pertinent examples from these systems.

Inconsistent partitions.

Partitions d^+ and d^- represent a kind of information that is both difficult to identify through static code inspections and problematic while debugging a system. In Visualizer $v22 \rightarrow v23$, a call from an external library `Ordering.givenOrder(List)` into the developer's code (`AggregationColumn.equals(Object)`) disappeared. Through static inspection, the developer cannot tell that the call from the external `givenOrder(..)` to their `equals(..)` method isn't happening anymore. Additionally, as `givenOrder(..)` and the developer's `equals(..)` method are not obviously related, it is more likely that they could inadvertently make a change that would cause this edge to disappear (for instance

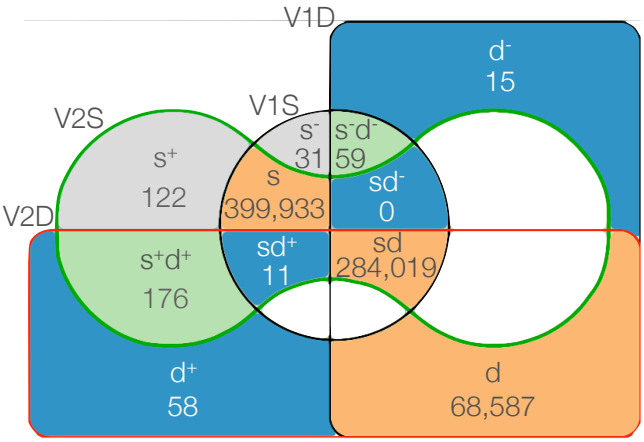

Figure 3: Aggregate totals for each partition from the 27 pairs of program versions in our analysis.

by removing the `equals(..)` method because it does not seem to be needed).

Edges appear in d^+ for the opposite reasons of d^-, that is, when statically opaque calls are added to the system. One common cause of these edges is when a new test is added to the system; for example, in JodaTime $v1366 \rightarrow v1367$ three new test methods are added to an existing class; JUnit uses reflection to identify these methods and executes them at runtime; while these edges are not statically obvious, they clearly effect the system's behaviour. In another JodaTime change ($v1379 \rightarrow v1380$) an unexpected call from `LocalDate.plusDays(int)` to `PreciseDurationField.add(long, int)` is reported. Looking at the code, `PreciseDurationField` is one of 19 subtypes of `DurationField` and is two levels down the type hierarchy. By highlighting this previously non-existent and statically opaque element, the developer can decide if this call was intentional.

Consistent and not-executed partitions.

From a test manager's point of view, s^+ represents additions not exercised by the test suite. Using $V2D$ to differentiate s^+ from s^+d^+ enables the developer to quickly ensure that the dependencies they added to their system are exercised as they intended. For example, in JodaTime $v1366 \rightarrow v1367$, the programmer added a new feature and corresponding test cases; the 13 s^+d^+ edges are unlikely to be surprising, as the programmer would expect the new code to execute, but the one s^+ edge, a call to `Assert.fail()`, could be unexpected. This dependency appears in s^+ because one of the tests throws an exception that causes the program exit before reaching the expected method call. In general, we do not expect developers will consider the results contained in the consistent partitions, except when they are numerically anomalous compared to other partitions they are investigating.

In summary, each inconsistent behaviour surfaced by our approach was non-obvious by manual inspection of the source code alone. While each of them could have been identified by inspecting the program with a debugger at exactly the right statement in the program, they would not have been easily otherwise isolated.

4.1.4 Non-Code and Environmental Changes

Changes to dependencies such as external libraries can cause subtle changes in the behaviour of a system; as a consequence, systems often ship with outdated versions of libraries as a way to reduce risk. Our approach can be used in situations where the developer wishes to compare how their system behaves with non-code changes applied. To test this idea, we executed JodaTime v1367 on JDK 5 and JDK 6. Our tool reported interesting and related dependencies in partitions sd^+ and sd^-. Specifically. in JDK 5 an exception is thrown via reflection when JodaTime tries to call a method that does not exist; the sd^+ represents the reflective method working in JDK 6 while the sd^- captures the alternate call JodaTime made to compensate for the reflective method's absence in JDK 5.

4.2 Industrial Code Base Study

We also applied our tool to an industrial code base comprising of approximately 120 KNCSL[5] that provide a widely-used online marketing platform. This code base is under active development using an agile methodology that delivers milestone builds after every three-week development sprint. Our objective was to gain insight into how an industrial developer would consider applying the information provided by our approach.

We applied our approach to one sprint lasting July 1 to July 22, 2010; during this time 241 change sets were committed to the repository. We computed the behavioural differences at the end of each day to correspond to execution of their nightly test suite. The code base involves dozens of external libraries and employs many different development techniques including mock objects, aspects, and a large amount of parallelization; we had to make some improvements to our tools (Section 3) to efficiently accommodate some of these techniques.

Table 3 shows the scale of the partitions over this time period (less-interesting columns have been elided). As with the the open source systems in Section 4.1, more than 99% of elements are in partitions s, d, and sd; since our industrial developer was only investigating the impact of a specific change he never sought out the information from these partitions. Table 3 also shows that the *consistent* and *not executed* partitions are the most heavily populated while the *inconsistent* partitions remain at a manageable scale.

Our participant is a development manager with more than 15 years of industrial development experience. While he frequently commits changes to the system, his primary role is to oversee the project and development team. To present the data to him we created a web page for each of the analyzed revisions; each page consisted of a tabular overview of the partitions and a specific listing of the changes. When investigating change `15579` $\rightarrow$ `15608` (Table 3 row 8) he noted, "Oh, that's interesting, we have been creating a new implementation of this code but it's all in the s+ partitions. If we wanted to do this right and have confidence in the new code we should be creating tests here." While considering how the approach could provide data to his team in the future he said that "right now upgrading our libraries is an ad hoc operation where we just try things out and hope they work; [this approach] could give us one more data point to help us consider whether a new component behaves as the old one did." And in terms of code reviews, he said, "I think that the partitioned information would be helpful [for assessing impact] especially in the context of a single known change and

[5]thousands of non-comment source lines

	Inconsistent				Consistent		Not Executed	
	d+	d-	sd+	sd-	s+d+	s-d-	s+	s-
15477->15485					1		63	1
15485->15488								
15488->15497								
15497->15519	33				42	18	19	4
15519->15548		2			5		1	7
15548->15575					18		41	3
15575->15578							24	
15579->15608	4				4		59	
15608->15624	3		4		21	6	22	21
15624->15633					14	12	8	2
15633->15672					4		2	2
15672->15689							6	
15689->15690	2		7		35		32	
15690->15697	1		6		28	10	20	11
15697->15703								
15703->15718					6		49	38

Table 3: Study results over 3-week code sprint.

a code review." An encouraging outcome of this study was that the development manager felt there was enough value to the data we generated that they requested we integrate our approach with their nightly build routine, so his team can continue to use this data in the future.

4.3 Threats to Validity

The main threat to the external validity of our findings is our use of a small set of systems over a limited sequence of versions. One commonality of the four systems we evaluated was their investment in unit testing; it is unclear how our approach would fare without a reliable way of executing the target system. A conspicuous risk to the construct validity of our qualitative evaluation is our reliance on our own experience and judgment as programmers and researchers to interpret the whether an INCONSISTENT dependency could be easily identified statically.

The projects we investigated retrospectively were at different points in their development lifecycle. JodaTime is a mature project, as can be seen by the very few code deletions (s^- and s^-d^-) as well as comparatively more elements in the CONSISTENT partitions than the NOT EXECUTED partitions (due to its high level of testing). Conversely, RFC-2445 is a project undergoing active development with many additions and deletions. The Visualizer is also fairly stable as evidenced by their lack of code churn. These distinctions may be material in ways our evaluation did not identify.

A less serious threat to the construct validity of our evaluation is the use of the two analyzers, JayFX and our own AspectJ tracer, to produce dependence graphs. Our tracer is straightforward, unconcerned at present with performance, and we doubt that another dynamic tracer would give significantly different results. It is clear that a different static analyzer would surely reshape our partitions to some degree. For instance, a more precise static analysis approach would decrease the size of d^+, d^-, and d — indeed, removing unexecutable dependencies is the core objective of increasing the precision of static analyses. Similarly, a less conservative analysis could decrease the size of s. We speculate, but have not confirmed, that these differences in precision would change the partition, but not the category, in which each dependency is placed. The reason for this speculation is that the more precise static analysis would likely capture the same dependencies in both versions, and most of these would be "cancelled" out by the differencing approach. In any case, our tool does not require a specific static or dynamic analysis to generate the four call graphs. Further research is needed to determine the degree to which the analysis matters in practice.

5. RELATED WORK

Software changes. Literature arguing that changes to behaviours are especially daunting includes: Brooks' observation that defects tend to arise from changes that have non-obvious system-wide ramifications [7, p. 123]; Ko et al.'s evidence that feedback about the fidelity of their changes proved to be the developers' most-sought piece of information [9]; and Sillito et al.'s study documenting that developers are keenly interested in the impact of their changes [20]. By providing developers feedback as they modify their system, we aim to help them identify dependencies that are otherwise difficult to isolate and that are likely to help build confidence in their understanding of these questions.

Unintended consequences of changes to programs are widely documented. Since 1985, the Risks Digest has documented thousands of computer-related risks to the public;[6] many of these, with effects from trivial to catastrophic, have been traced to unintended consequences of changes to programs. The observation that fixing errors is itself an erroneous process — often called imperfect debugging — has been modelled as part of software reliability engineering since the mid-1970's [12].

Belady and Lehman [10] have argued empirically that program change is inevitable (in part) because the needs of users evolve. Their Law of Increasing Complexity asserts that as programs change they become increasingly less structured (unless such entropy is consciously counteracted). Repeated behavioural changes alongside structural degradation naturally lead to a progressively opaque relationship between the static and the dynamic structures.

Exceptions in logical structural differencing [8] identify static changes that may have not been made consistently and completely. A study of refactoring argues that in practice some flurries of refactorings lead to an increase in bugs, while other flurries do not [24].

Not all program changes cause problems. Purushothaman & Perry [14], based on an extensive study of a major commercial system, have shown that about 10% of changes altered a single line of code and of these about 4% resulted in additional faults. They also showed that approximately 40% of all changes resulted in additional faults. Although rarely observed in the literature, these data mean that (at least in this study) roughly 60% of all changes and 96% of one-line changes improve, or at least do not harm to, the program.

Nonetheless, few, if any, would argue that programmers make changes with absolute certainty about their possible consequences when the program is executed. Our approach, like all other approaches, does not and never will lead to a situation in which programmers can attain such certainty. Rather, our approach is meant to provide one way among many in which programmers can increase their confidence that a change improves a program in intended ways.

[6] http://catless.ncl.ac.uk/Risks/

Impact analysis. Another very broad area of related work is impact analysis, which is generally concerned with identifying possible consequences of program changes [2]. A number of impact analysis approaches can be contrasted to our approach based on our partitions.

One common approach to impact analysis for regression testing relies on comparing two static dependence graphs: "most techniques select tests based on information about the code of the program and the modified version" [18, p. 529]. A classic example is safe regression test algorithms that eliminate all tests from an original program's test suite that cannot (under specified conditions) expose a fault in the modified program; Rothermel & Harrold have analyzed many variants [18], and a meta-analysis of empirical results is available as well [6]. Another example is test prioritization, which orders a test suite to increase the likelihood that newly introduced paths in the modified program are tested before unchanged paths. An example is Echelon [21], which exploits binary differencing of versions to identify paths and tests to exercise. We believe that our technique augments existing regression testing approaches by recording more behavioural data that can be further analyzed in the future, regardless of whether an explicit assertion in a regression test failed.

Person et al. apply differential symbolic execution to characterize the behavioural aspects of a code change more precisely than traditional diff-based mechanisms [13]. While the focus of their work is proving whether a change affected the behaviour of a system our approach focuses on partitioning and implicitly ranking behavioural changes, albeit with less precision, in a lightweight way that developers can easily reason about.

Any analysis based on comparing static dependence graphs across two versions can distinguish precisely three combinations of partitions: those with a label including s — that is, s, sd, sd^+ and sd^- — as well as those with a label including s^+ and those including s^-. Regardless of the kind of static dependence graphs that are extracted and regardless of the algorithm used to compare the two static call graphs, other distinctions cannot be made: for example, a dynamic dependence that appears or disappears cannot be determined using this approach.

Another collection of impact analysis approaches execute two distinct test suites across a single program. Software reconnaissance uses this approach to identify parts of a program that implement a particular feature: the feature is exercised by the first test suite, but not the second [25]. Eisenberg and de Volder have extended this approach to relax the explicit requirement of exhibiting and non-exhibiting test suites [5]. Reps et al. [17] used a similar approach to identify programs that might be susceptible to problems such as Y2K. The Tripoli system [19] can be used to compare two arbitrary executions of a system and determine their coverage differences, assuming that the source code remains unchanged. This class of approaches, based on two dynamic graphs and one static graph, can exploit up to eight partitions. Some distinctions available in our approach, however, cannot be made using these partitions. As an example, s cannot be distinguished from s^-, as only one static dependence graph is present. The degree to which this matters is empirical: if distinctions like this one arise in practice and represent useful information for the programmer, then our approach can provide additional information over these approaches. In addition, while traditional coverage metrics would report the same coverage percentage if one statement was removed and another added to the same method, our approach would capture both of these changes for the developer to consider.

Ren, Chesley, and Ryder present an approach to help perform root cause analysis (RCA) to identify the change that caused a test case to fail [16]. The AVA technique [3] further extends RCA to include rationale that can differentiate between successful and passing tests to differentiate passing behaviour from failing behaviour. Our approach aims to detect behavioural changes whether the test suite succeeds or fails but does not try to determine the root cause of any behavioural change. Raghavan et al.'s Dex tool extracts a variety of metrics about C patches using syntactic and semantic differencing; in contrast to our approach, Dex enables defect classification to enhance test prioritization [15].

One form of impact analysis that is incomparable using our partitioning is that based on historical information [27, 22]. These approaches generally look for patterns in source code repositories that likely represent co-dependences among, for instance, checked-in files; a programmer could be alerted upon attempting to commit a set of files that omit a resource that has usually been edited and checked-in alongside those files. These approaches are complementary, exploiting historical information not directly related to the source-behaviour relation.

Several previous approaches have used mixed analysis, using both static and dynamic dependence analysis in concert. They generally, and very reasonably, apply static and dynamic dependences to distinct parts of the problem, exploiting the strengths and weaknesses of each style of analysis. For example, Chen et al. [26] perform selective retesting of a system by using static analysis to determine which parts of a system changed and then comparing this to dynamically-derived coverage data. Rohatgi et al. [1] perform a software reconnaissance-like approach to extract features by comparing dynamic traces, following up by ordering the returned components based on their static relationships to one another. In contrast, our approach treats and manipulates the static and dynamic dependence graphs as peers, instead leveraging them by labelling specific partitions in terms of the underlying program source and its behaviour.

6. CONCLUSION

Software derives enormous power from its malleability. Experience shows that this power is often harder to harness in practice than in theory. One particular way in which this flexibility is often compromised arises when the behaviours of the executable program are hard to understand from the program's source. Even when this association is clear in a program's initial design and implementation, this relationship tends to becoming increasingly opaque as successive changes are made to the system.

We have presented an approach intended to concisely identify specific dependencies that suggest to programmers when a program change and the subsequent program behaviours may be less consistent than they may have intended. The approach relies on existing techniques and tools to extract static and dynamic dependence graphs from pairs of versions, along with set-based manipulations that partition these dependencies based on their absence or presence in the four graphs. We have argued, theoretically, that this partitioning

provides an opportunity for finer-grained and more concise results than classes of existing approaches; and we have argued empirically that the partitioning identifies a small set of apparently useful dependencies that can be difficult to succinctly identify using current analysis approaches.

Acknowledgments

We wish to thank our industrial partner for providing access to their source code and build environment and Rylan Cottrell, Brad Cossette, and Yuriy Brun for their insightful comments. This work was supported by the Natural Sciences and Engineering Research Council and by the National Science Foundation under award NSF-CCF-1016490.

7. REFERENCES

[1] A. H.-L. Abhishek Rohatgi and J. Rilling. An approach for mapping features to code based on static and dynamic analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 236–241, 2008.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of International the Conference on Software Maintenance (ICSM)*, pages 292–301, 1993.

[3] A. Babenko, L. Mariani, and F. Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–248, 2009.

[4] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] A. D. Eisenberg and K. D. Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 337–346, 2005.

[6] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: A systematic review. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 22–31, 2008.

[7] F. P. B. Jr. *The Mythical Man-Month (Anniversary Edition)*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1995.

[8] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 309–319, 2009.

[9] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 344–353, 2007.

[10] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, 1985.

[11] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341 – 355, 1987.

[12] I. Miyamoto. Software reliability in online real time environments. In *Proceedings of the International Conference on Reliable Software (ICRS)*, pages 194–203, 1975.

[13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 226–237, 2008.

[14] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions of Software Engineering*, 31(6):511–526, 2005.

[15] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 188–197, 2004.

[16] X. Ren, O. C. Chesley, and B. G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE Transactions of Software Engineering*, 32(9):718–732, 2006.

[17] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the European Software Engineering Conference held jointly with the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 432–449, 1997.

[18] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[19] K. D. Sherwood and G. C. Murphy. Reducing code navigation effort with differential code coverage. Technical report, University of British Columbia, September 2008.

[20] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 23–34, 2006.

[21] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–106, 2002.

[22] S. D. Thomas Zimmermann, Peter Weisgerber and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 563–572, 2004.

[23] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 1995.

[24] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 112–118, 2006.

[25] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[26] D. S. R. Yih-Farn Chen and K.-P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 211–220, 1994.

[27] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.