Detecting Unforeseen Program Behaviours

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

ABSTRACT

A developer modifies a program to achieve particular changes while avoiding the introduction of new bugs. Many changes also introduce behaviours that are not so easily characterized as "good" or "bad": for example, a new callback into the program from a library that arises after a change. Even if the change passes all regression tests, this behaviour may or may not be benign. Whether it causes problems may in fact only become apparent over time as the behaviour interacts with other changes.

We present an approach to identify specific program call dependencies where the programmer's changes to the program source 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 are likely to represent unforeseen parts of a change; by analyzing these partitions, the programmer can glean insight that is otherwise difficult to obtain.

1. INTRODUCTION

When programmers 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; for example, coverage of code that implements a new feature pertains to the first question, and a failing regression test pertains to the second one. Some behavioural variations, however, may not be easily isolated or identified as pertinent to either question. Such a variation may be benign, may represent subtle bugs, or may only make its colours known over time as it interacts with other behaviours.

Our approach identifies specific program call dependencies where a programmer's change to the program source is not apparent in the system's behaviour, or there is no apparent static change in the source corresponding to a behavioural change. For example, if a configuration file is updated as part of a larger program modification, a call that previously executed may no longer execute in the new version: this would be hard to determine from inspecting the program source directly. To identify dependencies like these, we extract a static and a dynamic call graph from each of two program versions, and partition the dependencies based on their presence in each of the four graphs. Particular partitions contain dependencies that are likely to represent unforeseen parts of a change. In the example of changing the configuration file, for instance, that dependency would be identified as newly appearing in the modified program's dynamic call graph and (most likely) as not being visible in the static call graph for either the original or the modified program.

We focus heavily on behaviours largely because they are especially daunting as programs change. Some of the vast literature supporting this notion includes: Brooks' observation that defects tend to arise from changes that have non-obvious system-wide ramifications [16, 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 [18]; and Sillito et al.'s study documenting that developers are keenly interested in the impact of their changes [29]. 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.

At the same time, 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. The high-level idea is 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

Submitted to FSE 2010.

the programmer should consider those deviations in more depth.

Partitioning static and dynamic call graphs over the original and modified program yields 15 partitions (the 16th would be the immaterial partition containing dependencies not found in any of the four graphs). We identify four of these partitions as represent inconsistent changes, while four more partitions are of potential interest to the developer; the remaining seven partitions are less relevant and are generally ignored. Our evaluation shows that more than 99% of dependencies fall into the seven partitions we believe are not material to the developers task; that is, less than 1% of the dependencies represent those of likely interest to the programmer with respect to unforeseen behavioural changes.

Our approach is novel in several specific ways. One, by combining the static and dynamic call graphs we are able to differentiate between dependencies undistinguishable using either type of call graph alone. Two, by focusing on the 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. Three, identifying inconsistent changes can be used to detect the effects of changes to the source code itself; it can also be used to hold the source code constant while changing the environment, for example altering the choice of external libraries, the virtual machine, or the network topology.

Our approach, like all other approaches, does not lead to a situation in which programmers can attain certainty about how a source change will effect subsequent process behaviour. Rather, our approach is meant to provide a complementary perspective that programmers can use to help determine if a static change is likely to affect the program's runtime execution in unforeseen ways.

Section 2 describes our partitioning approach, in particular characterizing what can and cannot be found in specific partitions. This section also briefly describes our prototype implementation including the extracted dependence graphs, how they are generated, how static and dynamic dependences are matched with one another, and some shortcomings of the current mechanisms. Section 3 shows the results of applying our approach to 10 versions of three different systems, providing both quantitative and qualitative insights into our approach and how it behaves relative to basic static-only and dynamic-only approaches. Section 4 discusses several dimensions of the approach. Section 5 lays out key related work and is followed by a brief conclusion.

2. APPROACH

We model a program's structure using call graphs that denote a program's methods and the calls between them. Our prototype extracts the four dependence graphs: the static call graph from before and after a change ($V1S$ and $V2S$) and the dynamic call graph from before and after the same change ($V1D$ and $V2D$) using lightweight analysis tools (see Section 2.2).

Given these four graphs, we compute all set intersections. Figure 1 shows a four-set Venn diagram of these intersections. The circle in the centre represents the static dependencies from the first version ($V1S$), the barbell-shape represents the static dependencies from the second version ($V2S$), the vertically-oriented rectangle on the right represents the dynamic dependencies from the first version ($V1D$), and the horizontally-oriented rectangle at the bottom represents the dynamic dependencies from the second version ($V2D$). Individual partitions are given descriptive labels; s means that all the dependencies in the partition are statically observable, d means the dependencies are dynamically-observable. Appending a $^+$ to s or d means the dependency was added as a consequence of the change; appending a $^-$ means the dependency was removed as a consequence of the change. For example, partition s^- includes only those dependencies that are in $V1S$ but are not in any of the other three graphs; that is, it was visible statically in the initial program but not in the modified program, and it was not executed dynamically in either version. Partition s^+d^+, as another example, contains only those dependencies that are found both statically and dynamically in the second version but are not found either statically or dynamically in the first version.

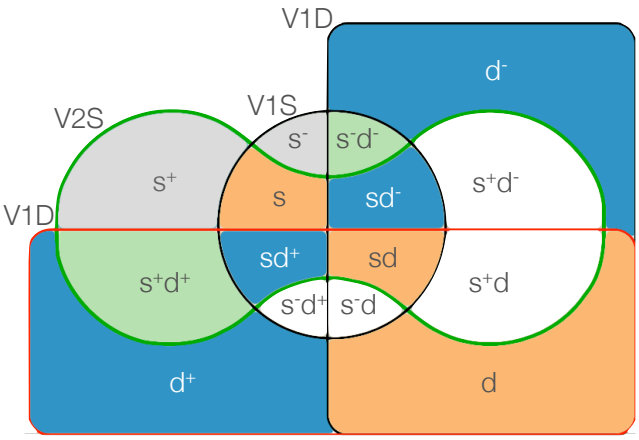


Figure 1: Analysis partitions with descriptive labels and coloured by their categorization.

2.1 Categorizing dependency partitions

The objective of a program change is generally to modify the program's behaviour, be it adding a feature or fixing a bug. Our categorization of the partitions makes two assumptions with respect to such changes. First, we assert that programmers have a strong sense of the code that they are modifying. For example, they add a method call with the expectation that it will sometimes be used in the modified executable, and they delete a method call with the expectation that it will never be executed. Second, when modifying a program developers do not generally strive to understand the program in its entirety, they instead focus on that part of program relevant to their change. Building this understanding may focus on the added behaviours, often assessed using new tests; or it may focus on not breaking existing behaviours, often assessed using regression testing. It is behaviours that may not fit into either category that we focus upon.

According to these assumptions, we group the fifteen partitions into five categories: INCONSISTENT, CONSISTENT, NOT EXECUTED, UNCHANGED, and UNLIKELY. Figure 1 shows

the fifteen partitions and is coloured by the five categories we have assigned the partitions to (respectively blue, green, grey, orange, and white).

Inconsistent. Dependencies in these INCONSISTENT partitions (d^+, d^-, sd^+, and sd^-) represent divergences between the static structure of the program and its dynamic execution. For example, a method call appearing in d^+ represents a call that started executing after a change was made, even though a corresponding call was not added to the source code. Conversely, a method call found in d^- represents a call being removed even though no corresponding method call was statically removed from the source code. A method call being added to sd^+ represents a new method call executing at runtime without a corresponding static change. This can happen if the change calls some code that was previously dead; any method call made by this dead code would appear in this region. Partition sd^- represents the case where a method continues to exist but is no longer called. Of the fifteen partitions, we consider the four in this category to be the most interesting and most likely to capture unforeseen behavioural changes.

Consistent. Dependencies in the CONSISTENT partitions (s^+d^+ and s^-d^-) represent changes that are coherent in their static and dynamic representations. For example, s^+d^+ represents the static addition of new methods that were also dynamically executed. Conversely, s^-d^- represents a method call that was deleted and whose corresponding executions disappeared after the change was made. We characterize these as unlikely to be surprising to the programmer as the developer usually has a strong understanding of the static nature of their change and the dynamic correspondence is unsurprising.

Not executed. Dependencies that are NOT EXECUTED (s^+ and s^-) represent method calls that were added or removed from the source but were never actually called (e.g., dead code). Dead code is rarely written intentionally, so we consider these two partitions to be of slight interest to the developer and include them primarily for completeness. In practice, they may arise, 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 where they might expect (s^+d^+). Incongruities between the CONSISTENT and NOT EXECUTED may be interesting if the developer expected their static change to be dynamically corroborated but was not.

Unchanged. Partitions s, d, and sd represent dependencies that were UNCHANGED between the two system versions. As our approach tries to surface facts that are interesting or useful about the current change, any facts that were present before the change was made, and were not altered by the change are uninteresting in this context. The overwhelming majority of the dependencies in the system fall into these three partitions. A dependency appearing in a non-unchanged partition for version n will appear in one of the unchanged partitions for version $n+1$ of the program unless it is changed again. For example, a method call added and executed in $v12$ of a program would appear in $s+d+$, but an analysis of $v13$ would find this same fact appearing in sd. As such, the unchanged partitions essentially aggregate all past changes.

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 (see Section 2.2) and thus will likely never be populated. For example, it is difficult to conceive a change that would statically delete a call from one specific method to another but would still execute after the change was made (s^-d^+). We have not observed an UNLIKELY dependency in practice.

2.2 Prototype implementation.

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

Static Analysis. In our prototype we generate the static dependence graph using Robillard's JayFX tool with the class-hierarchy analysis option disabled.[1] JayFX extracts many structural facts, but we only use those pertaining to `Relation.CALLS`. JayFX examines only the source code provided to it and does not consider any external library code. As such, its results seem to approximate a developer might generate through a manual code inspection.

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. In practice, 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.

In the unlikely case of being given a truly sound static analysis, our approach would never place any dependencies in the d, d^+, or d^- partitions; these dependencies would shift into corresponding partitions, most likely sd, sd^+, or sd^-, which would in any case leave then the same category (UNCHANGED or INCONSISTENT). This enables our approach to leverage any static analysis approach to generate the static call graph, although we have only used JayFX thus far.

Dynamic Analysis. Our prototype dynamic graph is generated using a custom tracer written using AspectJ.[2] This 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 was not applied to any external library call; this means that edges to a library (e.g., `myMethod()` → `Hashtable.put(..)`) are traced and this may result in another call edge (e.g., `Hashtable.put(..)` → `MyObject.equals(..)`), even if `put(..)` does not call `equals(..)` directly; we believe that from the developer's point of view this distinction does not matter, as from their perspective they are equivalent.

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

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 as the signatures are always generated with the same signature; between static and dynamic we perform some straightforward signature manipulations 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)`. This shortcoming will not prevent calls from appearing; for this example, while the calls are the same, one of them would appear in s^+ and the other would appear in d^+, instead of s^+d^+. We only ever observed this misregistration in the s and d partitions and have not needed to fix this problem as those are large UNCHANGED partitions anyway.

2.3 Interpreting surfaced dependencies

The INCONSISTENT, CONSISTENT, and NOT EXECUTED partitions capture the dependencies we think will be of most utility to the developer. Among these, we believe that the INCONSISTENT partitions have the highest utility. Being able to confirm that a change was executed as expected — by comparing the CONSISTENT and NOT EXECUTED partitions — can help a programmer build confidence in the behaviours of the program.

How a developer may interpret the results generated by our approach depends on the nature of the change the developer made and what they expected to happen. Below we describe two general scenarios that we believe capture a large subset of changes a developer may make to a system and how they may interpret the results in each case.

2.3.1 Identifying unexpected consequences of a change

Modifying systems to fix bugs or add new features comprise the bulk of all changes to a system. In these changes, developers typically apply some combination of modifying some existing code, adding some new code, and removing some unneeded code. In these situations, our approach can both confirm that the developer's static change altered behaviours consistently with static changes and also help identify situations wherein the new behaviours are incongruous with respect to the static changes.

Specifically, partitions s^+d^+ and s^-d^- describe the dependencies the developer added (or removed) from their system that were consequently executed (or not); in general, these facts are what a developer would expect when they added or removed code. As a counterpoint to that, partitions s^+ and d^+ describe the code the developer added and removed from their system that was not executed. While s^- should not be surprising (e.g., the developer removed code that was not run originally), dependencies appearing in s^+ that the developer expected to execute may prompt them (or a test manager) to investigate why the code they added is not being executed.

Elements appearing in any of the INCONSISTENT partitions represent code that started or stopped executing as an indirect consequence of the change. In particular, d^+ and d^- play an important role in identifying behavioural changes that are non-local with respect to the static change the developer made. In general, any dependency appearing in the INCONSISTENT partitions is sufficiently incongruous that the developer may wish to investigate it further. (Section 3 shows that these partitions are in practice small, often empty and rarely over a handful of dependencies, making such investigation feasible in practice.)

2.3.2 Detecting unexpected non-source changes

Instead of varying the source code and comparing to the corresponding behaviours, developers can use our approach to hold their source code constant and change their environment. Non-code changes that can be tested include updated external libraries, different virtual machines, running the system on a different physical machines or networks, or modifications to a non-source code file such as a metadata and configuration files.

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 3 shows a real example of this in upgrading an application from one JDK version to another.) As another example, if the developer changes some metadata XML file, they may find dependencies appearing in the d^- partitions as some callbacks stopped executing as a consequence of the metadata change.

3. EVALUATION

We evaluated our approach by applying it 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 programmers to reasonably study the actual dependencies, and (2) to qualitatively examine the dependencies to see if they were useful and non-obvious. While our approach could be applied to non-consecutive versions (for instance comparing two minor releases), our evaluation has focused on a per-commit granularity.

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 compare versions more easily without constructing potentially biased tests of our own. The three systems 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.

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

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.

3.1 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 files were ignored. To analyze the effects of each commit, for each system we compared all consecutive pairs of versions; 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 test suite. 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).

3.2 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 the associated commit operation to the repository.

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; we expect the elements in the UNCHANGED category would never be displayed.

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. These data show that method calls that are statically difficult for our prototype to identify are regularly added to the source code. (As discussed above, even if they were identified statically, this would generally shift them to a new partition within the INCONSISTENT category.)

The CONSISTENT partitions were non-empty for 15 of the 27 pairs of versions. In total, 231 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, although we do not expect they would ever investigate 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 2. 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.

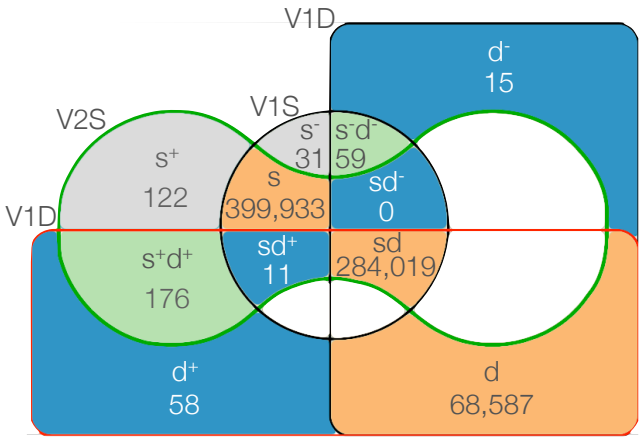


Figure 2: Aggregate totals for each partition from the 27 pairs of program versions in our analysis.

3.3 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 each change to their source code. By identifying specific dependencies in each partition, locating the source code associated with each is dependence straightforward, enabling the programmer 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.

[4]For simplicity, we only include counts of edges — dependencies between 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.

The projects we investigated were at different points in their development lifecycle, which can be seen by examining how their changes were partitioned. For example, JodaTime, a mature project, has very few code deletions (s^- and s^-d^-), it also tended to have 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.

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 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 edge, the programmer can decide if this invocation 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 may have 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.

Summary. Ultimately, every inconsistent behaviour surfaced by our approach in this evaluation 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.

3.4 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. DISCUSSION

4.1 Threats to validity

The main threat to the external validity of our findings is our use of only three specific systems and our analysis of their changes over a limited sequence of versions. Furthermore, all three systems are written in Java, are open source, come with test suites, and two are Google libraries. We did not note other particular commonalities such as the use of particular development tools or methods, although that cannot be excluded from consideration.

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.

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 is not directly dependent upon the static or dynamic analysis used to generate the four call graphs. Further research is needed to determine the degree to which the analysis matters in practice.

The theoretical power of our combination of the four dependence graphs across versions is by itself substantial; the presence of identifiable empirical differentiation in our experiment demonstrates the potential value of our approach.

4.2 Static- or dynamic-only approaches

An alternative approach would be to consider simply differencing the two static analyses or the two dynamic analyses. Examining only the static or dynamic analysis would be less expressive than our approach as the three partitions would only contain elements that were added, deleted, or common.

For our 27 program version pairs differencing the dynamic call graphs would result in 319 edges (245 in $d+$ and 74 in d^-); in contrast, our approach only returns 73 elements 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. Conversely, differencing only the static call graphs results in 388 edges (298 in s^+ and 90 in s^-) while our approach only returns 153 elements for these partitions (122 in s^+ and 31 in s^-), a 61% reduction. So, in contrast to static-only or dynamic-only differencing, splitting these partitions allows the developer to look at far fewer elements in general.

4.3 Expressiveness

One of the greatest advantages differencing two dynamic and and two static analyses has over differencing only one kind of analysis is the expressiveness it provides the developer. Each of the partitions in our approach has a specific meaning for the developer, enabling them to interpret them as appropriate for their task. As a specific example, consider edges 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.

5. RELATED WORK

Relating source code and program behaviour. In his legendary 1968 letter, Dijkstra made two relevant observations. First, he 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. Second, he 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" [7, p. 147]. This reasoning led Dijkstra and others to advocate the notions of structured programming [5], 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 make this relationship more opaque. Examples of such mechanisms abound: event-based programming [14], implicit invocation [13], upcalls [4], and related mechanisms decouple naming from invocation [31]; exception handling, where off-normal occurrences can cause nonlocal source code to be executed [15, 27]; dependency injection, where two disparate objects are made to work together by an external injector [12, 11]; and more. 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.

Wim De Pauw and colleagues have built visualization tools for object-oriented programming languages that are in part motivated by "the dichotomy between the code structure as hierarchies of classes and the execution structure as networks of objects" [6, p. 326]. Visualization techniques like these focus on relating the static and dynamic nature of a given program rather than on a pair of related programs. Many others have also used visualization with a similar goal, such as Baecker's [2] use of graphics as part of program source along with colored animation to better associate source code and program behaviour. QUAIL [32] approaches this gap by extending type checking across program layers that communicate by dynamically constructed strings, a common situation in which the source-behaviour relation is clouded.

Software change. Unintended consequences of changes to programs are widely documented. The idea 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 [20]. Since 1985, the Risks Digest has documented thousands of computer-related risks to the public; many of these, with effects from trivial to catastrophic, have been traced to unintended consequences of changes to programs.[5] Numerous published articles cite Weinberg's example where a change of a single character in a program cost a company US (circa 1980)$1.6 billion [34].

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

Belady and Lehman [19] have argued empirically that program change is inevitable because the needs of users evolve (in addition to changes in technology and the need to fix errors). Furthermore, they have documented the Law of Increasing Complexity, which asserts that as programs change they become increasingly less structured (unless work is done to maintain or improve structure and complexity). Repeatedly changing the behaviour of a program along with degradation of the structure of the source code naturally leads to an progressively opaque source-behaviour relationship. Parnas has argued many similar points [22].

Exceptions in logical structural differencing [17] 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 [35]; a later study [21] distinguishes styles of refactoring that may in part account for this mixed result.

Not all program changes cause problems. Purushothaman & Perry [23], based on an extensive study of a major commercial system, have shown that approximately 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. (Data like these are needed to tune the parameters of the software reliability models that account for imperfect debugging.) Although still unsatisfying to many, these data still 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, the program.

Nonetheless, few, if any, would argue that programmers make changes with 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.

Impact analysis. Another very broad area of related work is impact analysis, which is generally concerned with identifying possible consequences of program changes [1]. A number of impact analysis approaches can be characterized, as they contrast to our approach, using the partitions in Figure 1.

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" [26, 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 [26], and a meta-analysis of empirical results is available as well [10]. 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 [30] that exploits binary differencing of versions to identify paths and tests

to exercise. An empirical study of the effects of constraints in regression testing, which naturally leads to prioritization, provides some comparisons among approaches [8].

Any analysis based on comparing static dependence graphs across two versions can distinguish precisely three groups of partitions from Figure 1. 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 [36]. Eisenberg and de Volder have extended this approach to relax the explicit requirement of exhibiting and non-exhibiting test suites [9]. Reps et al. [24] used a similar approach to identify programs that might be susceptible problems such as Y2K. The Tripoli system [28], can be used to compare two arbitrary executions of a system and determine their coverage differences, but assumes the source code has not changed. This class of approaches, with 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.

One form of impact analysis that is not comparable using our partitioning is that based on historical information [37, 33]. 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 file 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. [3] 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. [25] 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 show that this power is often harder to harness in practice than in theory. One particular way in which this power 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 program.

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 intended by the programmer. 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.

7. REFERENCES

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

[2] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 356–366, 1988.

[3] Y.-F. Chen, D. S. Rosenblum, 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.

[4] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the ACM Symposium on Operating Systems Principles (SoSP)*, pages 171–180, 1985.

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

[6] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 326–337, 1993.

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

[8] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 71–82, 2008.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

[16] F. P. B. Jr. *The mythical man-month (anniversary edition)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[17] 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.

[18] 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.

[19] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[20] I. Miyamoto. "software reliability in online real time environment". In *Proceedings of the International Conference on Reliable Software*, pages 194–203, 1975.

[21] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 287–297, 2009.

[22] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.

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

[24] 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.

[25] A. Rohatgi, A. Hamou-Lhadj, 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.

[26] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996.

[27] 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.

[28] K. D. Sherwood and G. C. Murphy. Reducing code navigation effort with differential code coverage. Technical report, University of British Columbia, 09 2008.

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

[30] 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.

[31] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, 1992.

[32] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *Proceedings of the Conference on Object-oriented programming systems languages and applications (OOPSLA)*, pages 37–52, 2008.

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

[34] G. Weinberg. Kill that code. *Infosystems*, page 49, August 1983.

[35] 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.

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

[37] A. 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.