Industrially Validating Longitudinal Static and Dynamic Analyses

Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
rtholmes@cs.uwaterloo.ca

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

Mark Hancock
Department of Management Sciences
University of Waterloo
Waterloo, ON, Canada
mark.hancock@uwaterloo.ca

***Abstract*—Software systems gradually evolve over time, becoming increasingly difficult to understand as new features are added and old defects are repaired. Some modifications are harder to understand than others; e.g., an explicit method call is usually easy to trace in the source code, while a reflective method call may perplex both developers and analysis tools. Our tool, the Inconsistency Inspector, collects static and dynamic call graphs of systems and composes them to help developers more systematically address the static and dynamic implications of a change to a system.**

We have quantitatively validated the Inconsistency Inspector and have convinced ourselves that it can expose both interesting and surprising facets of a system's evolution. An initial case study with an industrial organization showed promise leading to the Inconsistency Inspector being installed at the organization for the past several months in preparation for a more in depth analysis.

In July 2012 we will have the opportunity to examine 8 months of industrial data, enabling us to perform an in-depth longitudinal evaluation of how their system has evolved and whether the Inconsistency Inspector can expose surprising and helpful facts for the industrial team. At the USER workshop, we hope to gather opinions about evaluation options for validating the industrial utility of our approach and the complex longitudinal data we have collected.

I. Introduction

We have developed an approach to concurrently capture a system's static and dynamic call graphs as the system evolves. By leveraging both static and dynamic data, we are able to compose the collected data to help the developer gain insight into their system's changes [3].

For example, consider a scenario where we have the static call graphs before ($vs1$) and after ($vs2$) a developer makes a change along with the dynamic call graphs before ($vd1$) and after ($vd2$) the same change. We can compose $vs1$ and $vs2$ to learn what calls the developer statically added or removed in their change. We can also compose the $vd1$ and $vd2$ to learn what the runtime effects of the developer's changes were. While this information may help the developer note inconsistencies in the call graphs (for example, a method call was statically added but didn't execute due to some other error), the developer must manually reconcile the different call graphs (for instance to remove all of the method calls that were statically added and dynamically executed). To help developers better understand the dynamic impact of their static changes, our approach overlays all four sets to split the results into 15 partitions (plus the uninteresting empty partition).

In this way, we believe "interesting" or "surprising" inconsistencies can be exposed more effectively than when only two sets of one kind of call graph are analyzed. In particular, we hypothesize that four of the partitions represent likely inconsistencies between a static change and their dynamic impact, while four others could be of interest depending on the developer's task. We do not believe the remaining partitions contain information developers would be interested in. We also believe that useful data could be highlighted when considered longitudinally, that is when weeks or months of data is considered at a time.

At its core, our approach aims to address Dijkstra's concern 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" [1, p. 147].

Dijkstra made this observation because with just a glance at the code of a structured program, a developer could gain a strong understanding of the runtime behaviour of their system. For other reasons, however, powerful language mechanisms, such as implicit invocation and dependency inversion have increased the cognitive gap between the static source code and its dynamic behaviour. Our approach is aimed squarely at reducing this gap using tools rather than language structures.

II. Our approach

We envision three key applications of our approach from the developer's point of view:

1) Proactive detection of changes that make the dynamic behaviour of the system more opaque.
2) Test-suite augmentation: by recording the dynamic call graph with each test suite execution, test reports can be augmented to provide more data about how the system executes beyond just reporting pass or fail and execution time.
3) Longitudinal analysis of dynamic program behaviour to assess program stability and change impact analysis.

978-1-4673-1859-4/12/$31.00 © 2012 IEEE

USER 2012, Zurich, Switzerland

Our current tool has focused on efficiently collecting the static and dynamic data without developer intervention as a part of the nightly build process. As such, the only interface we have currently developed simply shows the output data in a complex venn diagram, annotating the partitions with a label, a count of the edges in the partition, and a colour associated with our categorization of the partition; a sample view of our current interface is shown in Figure 1. In order to populate the diagram, the developer must select the two versions they are interested in; the data can then be explored by clicking on the regions in the venn diagram which takes the developer to HTML pages displaying the call relationships in that partition.

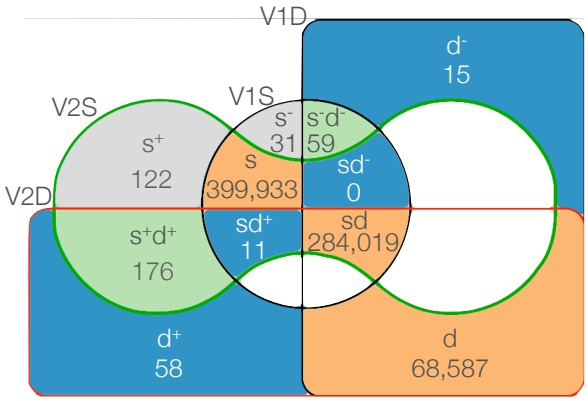

Figure 1. The current inconsistency inspector interface.

This initial interface allows basic visualization of the large set of data we have collected, but it provides only limited interaction and relies heavily on labels. The inconsistency inspector also focuses on individual changes rather than on trends and patterns that appear over a long period of time (e.g., 100's or 1000's of changes).

III. Design Strategy

We are actively exploring a variety of design alternatives to the visualization of this information. Specifically, we are investigating ways of providing a visual representation of (a) the members of each partition, (b) the changes across two versions of the code (rather than using labels), and (c) changes across more than two versions.

To represent the members of each partition, we are considering a variety of possible mappings. One alternative is to represent each change as a point on our existing visualization. These points could then be coloured using the same scheme, and another visual variable (such as shape or texture) could be used to represent a change over two versions of code. The changes over a longer period of time could then be represented using animation. We are also exploring the possibility of using a technique such as BubbleSets [2] to represent set membership. With this method, each change could again be represented as a coloured point, but these points would be mapped spatially to time across the horizontal axis. The vertical axis could then be used for the combinations of s and d and set membership would be represented using an isocontour. These are two examples of very early design ideas; however, we intend to use an iterative design strategy to create this next visualization, and so we expect these ideas to evolve significantly.

IV. Evaluation Strategy

Our approach has been tailored to help developers discover *surprising* properties of their system by carefully filtering relationships we do not believe are relevant to the developer. While we can perform retroactive quantitative evaluations on our own, ultimately we need feedback from real developers to effectively evaluate the utility of our approach.

To date we have performed two main evaluations (both previously reported [3]). First, we performed a quantitative evaluation that examined 10 versions of three different systems (9 pairwise changes per system). This evaluation confirmed that more than 99% of elements fall into partitions we believe are uninteresting to the developer; this leaves an average of 3 elements in the interesting partitions for the developer to investigate. We also found that the changes we expose and classify as 'inconsistent' were difficult (or impossible) to predict given a static code inspection alone.

Next, we ran the tool over 21 development versions of an industrial project. One industrial developer examined the results and confirmed that the results seemed promising. At their request, we installed the tool backend at their jobsite; it has now been collecting data nightly for the past several months.

We will be spending a month at the company this summer and would like to use this workshop as an opportunity to determine an effective way to validate our approach with them. By the time we arrive we will have collected 8 months worth of data; we will also have the opportunity to try some prototype interfaces for exploring the data with them in advance.

References

[1] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[2] C. Collins, G. Penn, and S. Carpendale, "Bubble sets: Revealing set relations with isocontours over existing visualizations," *Transactions on Visualization and Computer Graphics*, pp. 1009–1016, 2009.

[3] R. Holmes and D. Notkin, "Identifying program, test, and environmental changes that affect behaviour," in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 371–380.