Customized Regression Testing Using Telemetry Usage Patterns

Jeff Anderson
North Dakota State University, Microsoft
jeffrey.r.anderson@ndsu.edu

Hyunsook Do
University of North Texas
hyunsook.do@unt.edu

Saeed Salem
North Dakota State University
saeed.salem@ndsu.edu

***Abstract*—Pervasive telemetry in modern applications is providing new possibilities in the application of regression testing techniques. Similar to how research in bioinformatics is leading to personalized medicine, tailored to individuals, usage telemetry in modern software allows for custom regression testing, tailored to the usage patterns of an installation. By customizing regression testing based on software usage, the effectiveness of regression testing techniques can be greatly improved, leading to reduced testing costs and enhanced detection of defects that are most important to that customer. In this research, we introduce the concept of fingerprinting software usage patterns through telemetry. We provide various algorithms to compute fingerprints and conduct an empirical study that shows that fingerprints are effective in identifying distinct usage patterns. Further, we discuss how usage fingerprints can be used to improve regression test prioritization run time by over 30 percent compared to traditional prioritization techniques.**

I. Introduction

Regression testing is one of the most common and important activities performed during software development to ensure the quality of the modified system [4]. The use of regression testing allows developers to make changes to software with confidence, ensuring that faults introduced to existing functionality will be discovered prior to releasing the software to customers. As software evolves over time, the size of the test suite can grow rapidly, which introduces a new problem. The costs of executing and analyzing regression test results can become expensive, especially in larger projects.

Efforts have been made to mitigate the cost of large regression test suites through various regression testing techniques such as regression test selection and test case prioritization [9][17]. Traditionally, these techniques have been applied statically, such as by using code churn to identify areas which most require testing.

Two recent trends in software are a move to Software-as-a-Service (SaaS) and pervasive telemetry. With SaaS, software is provided via the Internet and not installed individually on a user's own computer. With this shift in how software is delivered comes a shift in how software issues are managed. When downtime occurs in traditional on-premises software, front line support is provided by IT professionals on site, and software fixes from the software vendor are expected to take some time to be implemented. With SaaS, the software vendor is expected to keep the software running at much higher levels of reliability than traditional on-premises software. When downtime occurs, service level agreements (SLAs) mean that the speed at which issues can be fixed can have major economic impacts on the profitability of a service.

Pervasive telemetry means that much more execution data is now collected than ever before. This is partially due to the move to SaaS, as execution logs can now be collected over the Internet as opposed to being written to local log files. It is also due to the decrease in the cost of storage memory; the collection of many gigabytes of data is now economically feasible for all products.

We feel that these two trends interact to provide a unique new opportunity in the area of regression testing. Similar to how genome sequencing in bioinformatics has sparked the new field of personalized medicine [10], we believe that pervasive telemetry from SaaS products can be used to create customized regression testing and more accurately prioritize test cases based on actual usage. This will increase quality and at the same time reduce testing and repair times, both of which have positive economic benefits.

Telemetry has been used in the area of software engineering research. For example, Brooks and Memon used telemetry data to generate test cases for graphical user interfaces (GUIs) [5], and similarly, Amalfitano et al. [1] used telemetry from rich Internet applications to generate test cases. Elbaum et al. [8] have attempted to use profiling from deployed systems (telemetry data) in more traditional regression testing techniques. Their research was limited by the available telemetry at the time. The research described above has focused on regression testing for full product releases. In contrast, our research seeks to use pervasive telemetry across all installations in order to provide custom prioritization of regression tests tailored for a single installation when responding to critical situations requiring immediate patches.

In this research, we introduce the concept of telemetry fingerprinting. A telemetry fingerprint is the result of a fingerprinting algorithm that determines how similar one set of telemetry is to another. If good fingerprint algorithms can be identified, then test suites that closely match a telemetry fingerprint are known to be similar to the usage of that particular service. We hypothesize that algorithms that yield a higher fingerprint score will be more effective at identifying test suites that are similar to the actual usage of the service. This allows for efficiently testing the aspects of the service that will be of most importance to users.

The primary contribution of this research is the introduc-

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.30

tion of the concept of telemetry fingerprinting for regression testing. We apply this new concept to a pre-release industrial product, *Microsoft Dynamics AX*. From our empirical study, the technique of fingerprinting is shown to be useful in prioritizing test cases based on actual product usage. We discuss issues encountered while attempting to apply this technique, as well as propose future research.

The rest of the paper is organized as follows. In Section II, we discuss the approach used in the research as well as formally define telemetry fingerprinting. In Section III, we detail the empirical study that we performed. Section IV provides the results of the study. Section V discusses the results and the implications of these results, and Section VII presents related work. Finally, in Section VIII, we provide conclusions and discuss future work.

II. Proposed Approach

A. Motivation

Large-scale products such as *Microsoft Dynamics AX* often have extremely large regression test suites due to the vast set of features available in the product. While this is beneficial in avoiding regressions, the tests are significantly costly to execute. With the advent of Software-as-a-Service (Saas), customer expectations have changed and the time available to patch serious issues in production is shrinking significantly. Downtime is now often measured in seconds or minutes, meaning it may not be feasible to run a full regression test suite prior to deploying a patch [7]. Reliability of two-nines (99%) allows 7.2 hours of downtime per month. Going to three-nines (99.9%) allows only 43.8 minutes of downtime. At five-nines (99.999%) only 25.9 seconds of downtime is allowed per month.

Traditional regression testing techniques such as reduction, selection, and prioritization can decrease testing costs. Much research has been done showing these techniques are effective at maintaining quality while reducing costs and testing time. In the world of SaaS, however, another quantum leap in the effectiveness of regression testing techniques is necessary. We believe our approach of customized regression testing based on usage fingerprinting can yield these significant improvements.

A second motivation for this approach is based on the priority companies place on stability of their software. Major enterprise resource planning (ERP) products have functionality spanning a wide range of features, everything from accounting to shop floor control to retail functionality and beyond. Virtually no company using the product ends up using all the functionality, usually it is only a relatively minor subset of the total product footprint. This gives rise to an interesting phenomenon. Even if an extremely serious, data damaging bug exists somewhere in the product, that company does not care about it as long as it is not in an area they are using.

To address this consideration, we introduce the concept of "discoverable bugs." Discoverable bugs are those which will impact one or more users of the software. If a bug exists but no user ever is impacted by it, that bug is less important than the bugs which users are impacted by.

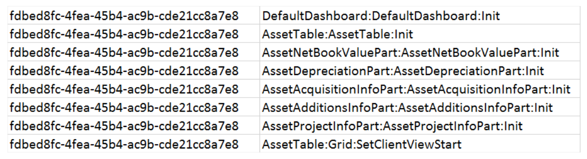
Once a company is running an ERP product in a stable state, they often choose *not* to take hot fixes for even very serious bugs if that bug is not impacting them. This is because any hot fix, no matter how important, has the potential for causing a regression. Even if a regression does not occur, the cost of applying the hot fix, including downtime and testing, is often not worth the return on value. Here too our approach of fingerprinting usage is beneficial, as we are able to identify whether or not a given hot fix will impact a given customer to help in the decision on whether or not it should be applied.

B. Approach

1) Computing Fingerprint Scores: In this section, we describe the approach used in fingerprinting data processes and tests, and how that fingerprint is utilized in regression testing techniques. This research was done using a pre-release version of *Microsoft Dynamics AX*. This version was installed and used in a live environment by multiple companies. Telemetry was collected from these installations.

In this research, we use the term *telemetry fingerprint* or just *fingerprint* to mean the unique product usage signature derived from telemetry information. In all telemetry sessions, the product and version are both the same, containing the same code. The difference is in the usage of that product in a given installation. For instance, a distribution company will have a very different pattern of usage from a financial services company, focusing on very different areas of the product. We call these differences in usage *fingerprints*.

Figure 1 shows an example of the raw data. The first column is the session identifier, which identifies a single user navigating through the product. The second column is the set of distinct interactions performed by the user that caused a call to the server. The syntax of the interactions is of the format (**Form name**):(**Control name**):(**Command name**).



fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	DefaultDashboard:DefaultDashboard:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetTable:AssetTable:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetNetBookValuePart:AssetNetBookValuePart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetDepreciationPart:AssetDepreciationPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetAcquisitionInfoPart:AssetAcquisitionInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetAdditionsInfoPart:AssetAdditionsInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetProjectInfoPart:AssetProjectInfoPart:Init
fdbed8fc-4fea-45b4-ac9b-cde21cc8a7e8	AssetTable:Grid:SetClientViewStart

Fig. 1. Sample Interaction Session

Using the example from Figure 1, the user performed the following steps or interactions:

- The user started on the default dashboard (home page).
- The user clicked on the fixed assets form and launched it.
- While the fixed assets form was loading, a number of form parts loaded.
- The user clicked in the main grid on the fixed assets form.

A collection of sessions similar to the one shown in Figure 1 exists in telemetry for each installation. In this paper, we are trying to determine whether the collection of sessions provides a unique fingerprint for that installation. To determine if a fingerprint exists, we need a way of determining how similar two sessions are. A variety of different session similarity functions are discussed later in this paper.

We define a session S to be an ordered list of user interactions. A single session comes from a single user and a single installation of the software product.

$$S = \{a_1, a_2, \cdots, a_l\}$$

where a_i is an interaction and l is the length of the session.

We define a set of sessions G_i to be an unordered collection of sessions from the same installation of the software product and is defined as follows:

$$G_i = \{S_1, S_2, ...S_m\}$$

where S_i is a session that belongs to the installation, and m is the number of sessions in the installation.

The set of all sessions from all installations is denoted as D and is defined as follows.

$$D = \bigcup_{i=1}^{c} G_i$$

where c is the total number of installations.

To determine fingerprints, we introduce a similarity function $O : D \times D \rightarrow \mathbb{R}$ that measures the similarity between two sessions normalized between 0 and 1.

For sessions in an installation, G_i, we introduce two fingerprint measurements, internal and external. The internal similarity measures the average pairwise session similarity of all the sessions that belong to the same installation.

$$F_{internal}(G_i) = \frac{2}{|G_i| \times (|G_i| - 1)} \sum_{S_j \in G_i} \sum_{S_k \in G_i, k \neq j} O(S_j, S_k)$$

The external similarity measures the average similarity between the sessions in an installation and all the sessions in the other installations.

$$F_{external}(G_i) = \frac{1}{|G_i| \times |D \setminus G_i|} \sum_{S_j \in G_i} \sum_{S_k \in D \setminus G_i} O(S_j, S_k)$$

Finally, we produce a fingerprint score FP. The fingerprint score is the ratio of how similar sessions from the same installation are compared to sessions from other installations for a given similarity algorithm.

$$FP(G_i) = \frac{F_{internal}(G_i)}{F_{external}(G_i)}$$

In this research, we seek to find session similarity functions that will maximize the value of FP for a dataset D. We hypothesize that similarity functions that yield a high FP value will be most effective at identifying usage patterns.

2) Evaluating Fingerprint Effectiveness: To evaluate the effectiveness of fingerprinting, we wish to determine how accurately the fingerprint score can correctly identify a usage pattern. A simple way to do this is to remove n sessions randomly from each installation G_i and add them to a set R_i, then see if fingerprint scores can be used to derive which installation those sessions originally came from. We then define G'_i to be the set of sessions for an installation i with those randomly selected sessions in R_i removed.

$R_i = \{S_1, S_2, ...S_n\}$, $R_i \subseteq G_i$, $G'_i = G_i \setminus R_i$

For each R_i we then compute the percentage of sessions which would be properly attributed to having come from G_i by selecting the installation giving the highest fingerprint score. We compute the membership score M_i for session S_j in installation i as follows.

$$M_i(S_j) = \frac{1}{|G'_i|} \sum_{S_k \in G'_i} O(S_j, S_k)$$

We say the session S_j is "owned" by the installation i with the largest membership score M_i. Since we know the true owner of the session, we can then compute the percentage of sessions for which the owning installation was found.

3) Applying to Regression Testing Techniques: Because of how this product is architected, automated regression tests produce the same set of telemetry that user interactions do. So a test session T_y is equivalent to a user session S_j.

$$T_y = S_j = \{a_1, a_2, \cdots, a_l\}$$

This means the similarity function O previously introduced can also be used to compute the fingerprint score for a test and an installation.

$$F_{test}(T_y, G_i) = \frac{1}{|G_i|} \sum_{S_k \in G_i} O(T_y, S_k)$$

These test fingerprint scores F_{test} can then be applied in regression testing techniques. In the test prioritization technique, tests are ordered such that the most important tests are executed first. The fingerprint score $F_{test}(T_y, G_i)$ is computed for all regression tests for an installation based on previous usage. The test cases are then ordered by descending fingerprint score so tests with the highest fingerprint scores are executed first.

C. Example

This section provides an example of the computations explained in the previous section. Suppose we have two datasets G_1 and G_2. $D = G_1 \cup G_2$

For each of these datasets, we have three sessions of data, labeled $G_1 = \{S_a, S_b, S_c\}$ and $G_2 = \{S_d, S_e, S_f\}$. This is a total of six sessions. Further suppose that the algorithm being used for fingerprinting yields the pairwise similarity measurements listed in Table I.

In order to compute the $F_{internal}$ score for G_1, we average all the pairwise similarities from within G_1.

$$F_{internal}(G_1) = \frac{0.786 + 0.627 + 0.658}{3} = 0.690$$

Similarly, we compute $F_{internal}(G_2)$.

$$F_{internal}(G_2) = \frac{0.682 + 0.841 + 0.773}{3} = 0.765$$

TABLE I
EXAMPLE PAIRWISE SIMILARITIES

Session 1	Session 2	Pair-wise Similarity
S_a	S_b	0.786
S_a	S_c	0.627
S_a	S_d	0.423
S_a	S_e	0.400
S_a	S_f	0.128
S_b	S_c	0.658
S_b	S_d	0.351
S_b	S_e	0.109
S_b	S_f	0.220
S_c	S_d	0.423
S_c	S_e	0.726
S_c	S_f	0.101
S_d	S_e	0.682
S_d	S_f	0.841
S_e	S_f	0.773

The $F_{external}$ score is computed the same way, only using all pairwise similarities from two different installations.

$$F_{external}(G_1) = \frac{0.423 + 0.400 + 0.128 + ...}{9}$$

$$F_{external}(G_1) = 0.320$$

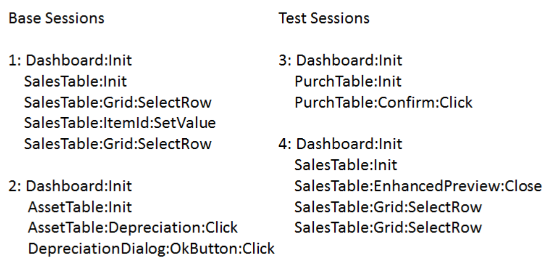
In this case, $F_{external}(G_2)$ is also 0.320 since the same nine pairwise similarities exist for G_2. The final fingerprint score for each is the ratio of internal to external score.

$$FP(G_1) = \frac{0.690}{0.320} = 2.156$$

$$FP(G_2) = \frac{0.765}{0.320} = 2.391$$

D. Fingerprinting Algorithms

In this section, we describe the fingerprint algorithms in detail using examples. In these examples, we will use the sample sessions shown in Figure 2.



Base Sessions	Test Sessions
1: Dashboard:Init	3: Dashboard:Init
SalesTable:Init	PurchTable:Init
SalesTable:Grid:SelectRow	PurchTable:Confirm:Click
SalesTable:ItemId:SetValue	
SalesTable:Grid:SelectRow	4: Dashboard:Init
	SalesTable:Init
2: Dashboard:Init	SalesTable:EnhancedPreview:Close
AssetTable:Init	SalesTable:Grid:SelectRow
AssetTable:Depreciation:Click	SalesTable:Grid:SelectRow
DepreciationDialog:OkButton:Click	

Fig. 2. Examples of Session Data

In this example, session one shows a user starting from the main dashboard, launching the sales order form, switching rows, filling in a field, then switching rows again. Similarly, session two shows a user starting from the dashboard, going to the fixed assets form, opening the depreciation dialog, and clicking "ok". Session three starts from the dashboard, opens the purchase order form, and confirms a purchase order. Session four again starts from the dashboard and opens the sales order form, but this time closes the enhanced preview and then switches rows twice.

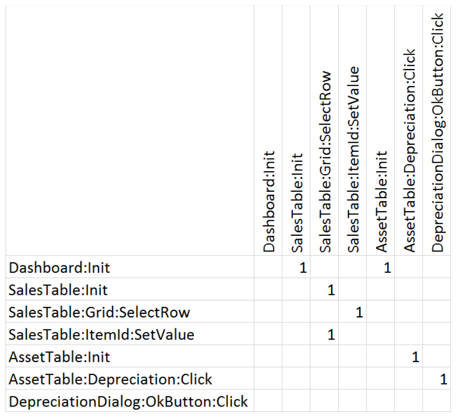
1) By length: Fingerprinting by length is done by taking the length of the shorter session and dividing by the length of the longer session. This value is averaged across all combinations in the two sets. So for this example, session three is length 3 and session one is length 5, giving a ratio of $3/5 = 0.6$. Sessions two and three yield a ratio of $3/4 = 0.75$. Sessions one and four yield $5/5 = 1$. Sessions two and four yield $4/5 = 0.8$. Averaging across all four combinations gives a final fingerprint score of $(0.6 + 0.75 + 1 + 0.8)/4 = 0.788$.

2) By form match: Form match is the ratio of the number of times each form from the small session exists in the big session. Using Figure 2 as an example, the length of session three is shorter than the length of session one. The first form in session three is the *Dashboard*, which does exist in session one. The second is *PurchTable*, which does not exist in session one. The third is again *PurchTable*, which does not exist in session one. The overall score for this is $(1+0+0)/3 = 0.333$.

Using the same technique for sessions one/four, two/three, and two/four, we get scores of 1, 0.333, and 0.25, respectively. This yields an overall fingerprint score of $(0.333+1+0.333+0.25)/4 = 0.479$.

3) By action match: Action match scoring is nearly identical to form match, only the full action is compared as opposed to just the form name. When comparing sessions one and four by form match we compute a score of 1, but action match only scores 0.8 because the *SalesTable: ItemId:SetValue* action is not found in session four. Completing this example, the scores for combinations one/three, one/four, two/three, and two/four are 0.333, 0.8, 0.333, and 0.25, respectively, for a total score of $(0.333 + 0.8 + 0.333 + 0.25)/4 = 0.429$.

4) By Markov chain: The first step in computing the Markov chain probability is to generate a matrix of how many times each ordered pair of interactions was observed in the base sessions. An ordered pair of interactions is also referred to as a "navigation" by the user from one interaction to another. This matrix serves as the base emission matrix for the Markov chains. Figure 3 shows this initial matrix.

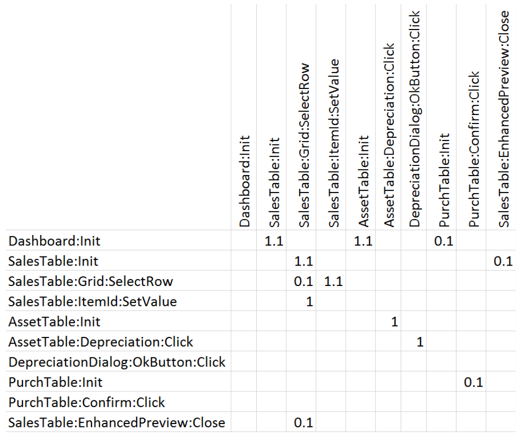


	Dashboard:Init	SalesTable:Init	SalesTable:Grid:SelectRow	SalesTable:ItemId:SetValue	AssetTable:Init	AssetTable:Depreciation:Click	DepreciationDialog:OkButton:Click
Dashboard:Init		1			1		
SalesTable:Init			1				
SalesTable:Grid:SelectRow				1			
SalesTable:ItemId:SetValue			1				
AssetTable:Init						1	
AssetTable:Depreciation:Click							1
DepreciationDialog:OkButton:Click							

Fig. 3. Example of Initial Emission Matrix

A correction must be made to account for transitions that do not exist in the base sessions but do exist in the test sessions. For example, the navigation from the *Dashboard* to *PurchTable* does not exist at all in the base sessions, nor does the navigation to *SalesTable:EnhancedPreview:Close*. So without modifying the emission matrix, the resulting probability would almost always be zero.

To account for these missing navigations, we apply a correction. This is illustrated in Figure 4. A small value is added to the cell for the *Dashboard:Init* to *PurchTable:Init* navigation to avoid yielding a zero probability. To avoid impacting the existing navigation ratios, that same value of 0.1 is also added to the two other navigations. This same process is used for all missing navigations.



	Dashboard:Init	SalesTable:Init	SalesTable:Grid:SelectRow	SalesTable:ItemId:SetValue	AssetTable:Init	AssetTable:Depreciation:Click	DepreciationDialog:OkButton:Click	PurchTable:Init	PurchTable:Confirm:Click	SalesTable:EnhancedPreview:Close
Dashboard:Init		1.1			1.1			0.1		
SalesTable:Init			1.1							0.1
SalesTable:Grid:SelectRow			0.1	1.1						
SalesTable:ItemId:SetValue			1							
AssetTable:Init						1				
AssetTable:Depreciation:Click							1			
DepreciationDialog:OkButton:Click										
PurchTable:Init									0.1	
PurchTable:Confirm:Click										
SalesTable:EnhancedPreview:Close			0.1							

Fig. 4. Example of Emission Matrix with Corrections

Once all necessary corrections have been made to the emission matrix, a probability can then be computed. The probability of seeing session three given the emission matrix is the probability of each navigation multiplied together. The probability of the *Dashboard:Init* to *PurchTable:Init* navigation is $0.1/(1.1 + 1.1 + 0.1) = 0.0435$. The probability of *PurchTable:Init* to *PurchTable:Confirm:Click* is $0.1/0.1 = 1$. So the total probability of seeing session three given the emission matrix is $0.0435 * 1 = 0.0435$.

The same technique is used to compute the probability of session four; $(1.1/(1.1+1.1+.1))*(.1/(1.1+.1))*(.1/.1)*(.1/(1.1 + .1)) = 0.478 * 0.083 * 1 * 0.083 = 0.00302$.

The average fingerprint score for these sets of sessions is therefore $(0.0435 + 0.00302)/2 = 0.0233$. Note that the Markov chain scores are very small compared to other scores. But since both the internal and external scores are similarly small, the resulting fingerprint ratio is still directly comparable with other techniques.

5) By E count: The last technique is arguably very naïve, but is included to show the power of this fingerprinting approach. In this technique, the ratios of the number of occurrences of the letter *E* from one session to another session are compared. In session one, the letter *E* occurs fourteen times (twice in *SalesTable:Init*, four times in *SalesTable:Grid:SelectRow* and so on). It occurs eight times in session two, twice in session three, and seventeen times in session four.

The ratio of smaller to larger count in each comparison of one/three, one/four, two/three, and two/four yields a fingerprint score of: $((2/14) + (14/17) + (2/8) + (8/17))/4 = 0.422$.

While this technique may appear somewhat random, note that the number of times the letter *E* occurs in a given session is somewhat related to which forms are in use in that session. *SalesTable* has two, while PurchTable only has one. This means that even a silly technique like counting letters will encode some information about whether or not the same forms are being used in two sessions.

III. Study

In this study, we seek to determine if fingerprints can be found for the various ways in which a software product is used. First, we must determine if fingerprint scores can be obtained and if they are relatively stable values. Once fingerprint scores have been obtained, we need to determine if they are effective in identifying usage patterns. Finally, we need to determine if fingerprint scores can be used to improve regression testing techniques such as test case prioritization.

Our hypothesis is that a higher fingerprint score will lead to better identification of usage patterns and therefore effective application in regression testing techniques. To investigate this hypothesis, we consider the following three research questions.

- RQ1: Can stable high fingerprint scores be computed for different installations?
- RQ2: Are fingerprint scores effective at identifying usage patterns?
- RQ3: Can fingerprinting be useful for improving regression testing techniques?

When evaluating RQ1, the primary concern is the ratio of internal to external scores (the overall fingerprint score), because high-scoring algorithms must exist for this research to be applicable. A random scoring would on average yield a fingerprint score of 1, so the higher the fingerprint score, the more effective the algorithm.

In addition to the overall score, stability is also important. In order to be effectively employed in regression testing techniques, those fingerprint scores must be stable with little variability when the dataset is randomized. If the scores are not stable, then application of that algorithm in regression testing techniques will have high randomness.

In RQ2, we wish to determine if the fingerprint scores investigated in RQ1 are useful in identifying usage patterns. Even if high and stable fingerprint scores can be found, they are not of much use unless they uniquely identify how the software is being used. To investigate RQ2, we take a random set of interactive sessions and see if the fingerprint of the session can be used to identify the source installation. If fingerprint scores are successful in identifying the source installation, then we know that the fingerprint is effective.

Finally, in RQ3, we examine how these techniques can be applied. Regression testing techniques such as test case prioritization seek to find correlations between tests and aspects of a software product to improve the effectiveness of testing. We need to evaluate how fingerprinting can be used in regression testing techniques and how useful it may be in reducing the number of tests to be retested or improving the fault detection rate of a test suite.

A. Object of Analysis

For this research, we used telemetry from a pre-release version of the software product *Microsoft Dynamics AX*. The newest version of this software product is delivered in a Software-as-a-Service (SaaS) model. This means that the software is hosted on servers owned by Microsoft Azure, and the installation and management of the service is handled by Microsoft or other service providers.

As part of the monitoring of the software service to ensure availability, a standard set of telemetry events is collected from the running service. One such event is the *UserInteraction-Marker* event, which indicates an action performed by a user. While the specifics of the action are not directly available for analysis due to privacy concerns, a hashed value for each interaction is maintained. This means that unique patterns of interaction with the application can be identified even if there is no way of determining what those specific interactions were.

Figure 5 illustrates the pipeline for collecting and reporting telemetry of this product. While not fully detailed, it provides a rough approximation of how telemetry is collected. The product has a browser-based user interface. Each time the user performs an action in the product, the Application Object Server (AOS) to which the browser is connected logs a *UserInteractionMarker* event. The overhead for this telemetry collection has been found to be around 1 to 3 percent in performance testing.

Approximately once every 5 minutes the collection of *UserInteractionMarker* events are sent to the cloud-based Azure Monitoring and Diagnostic Service (MDS). As part of this transfer, all identifiable user information is scrubbed and hashed out from the events to ensure user privacy. The resulting aggregate information is stored in Azure Hadoop BLOB storage, from which telemetry reports are generated. Because user information has been hashed out, there is no way to determine individual user actions, but usage trends can be derived.

In this study, we utilized telemetry streams from seven pre-release software installations. Because we are working closely with these seven companies to validate the pre-release software version, we were able to obtain detailed telemetry information. We will refer to these installation as "Installation A" through "Installation G".

For each of the seven installations, we randomly selected 500 user sessions, each session consisting of at least ten user actions. These user sessions were all collected at the same time, so seasonal variances in usage are not accounted for in this study. Table II describes the specific datasets. As described in Section II, a session is an ordered set of actions performed by a user. We do not have access to user information, so some users may have performed only one session while other users may have performed many sessions.

TABLE II
SESSION INFORMATION BY DATASET

Dataset	Number of Sessions	Average Session Length
Installation A	500	182
Installation B	500	281
Installation C	500	421
Installation D	500	141
Installation E	500	193
Installation F	500	479
Installation G	500	173

B. Variables and Measures

In RQ1 and RQ2, we manipulate one independent variable, the fingerprint algorithm. For RQ1, the dependent variable is the fingerprint score per installation, while in RQ2 the dependent variable is the percentage of sessions for which the owner was correctly identified. In RQ3, the independent variable is the prioritization technique and the dependent variables are the APFD and total runtime of the test suite until all discoverable bugs are found.

C. Experimental Process

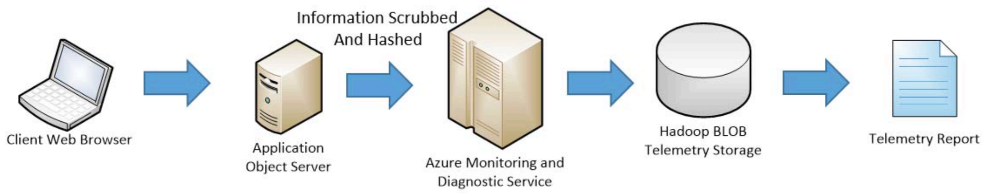
Figure 6 shows the process used for determining fingerprint scores for each similarity algorithm. The sessions for each installation G_i are used to compute the internal and external fingerprint scores, from which the final $FP(G_i)$ for that installation can be computed.

Once the fingerprint scores are known, we then wish to see if fingerprint scores can be used to accurately identify the owning installation of a given session. Figure 7 shows the process for this step.

For each installation G_i, we randomly select ten percent of the sessions and add them to R_i. Other values above and below ten percent were also tried and yield similar results. The selected sessions are removed from the initial data set leaving G_i'. We then compare each of the randomly selected sessions with all the sessions in each installation's G_i' to compute the membership score M_i for each. Whichever installation yields the highest membership score is selected as the owning installation. From that we compute the percentage of sessions that were correctly mapped to their original installation.

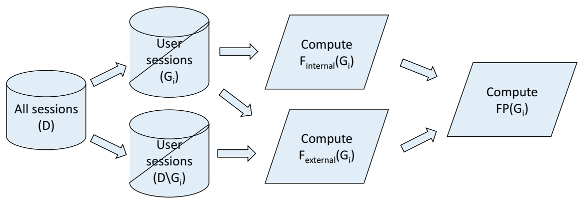
Finally, to investigate RQ3, we wish to see what impact fingerprint techniques would have on test case prioritization. The most common measure of prioritization effectiveness is APFD (average percentage of faults detected) [14]. The APFD value ranges between 0 and 100, and higher numbers imply faster fault detection rates. We also examine the total runtime of the tests necessary to expose all discoverable bugs.

Figure 8 illustrates the process for RQ3. We start by injecting artificial faults, essentially marking individual interactions from test sessions as having a fault. Artificial faults were necessary as this is a new product so faults from updates do not exist yet. These are all the faults that could potentially be found by running tests. We chose to simulate 10 percent of the interactions being faulty. Similar results are seen with both higher and lower percentages, but 10 percent yields the most stable results across repeated runs. Smaller percentages yield the same average results, but with more variability.



Information Scrubbed And Hashed
Client Web Browser
Application Object Server
Azure Monitoring and Diagnostic Service
Hadoop BLOB Telemetry Storage
Telemetry Report

Fig. 5. Telemetry Pipeline



All sessions (D)
User sessions (G_i)
User sessions ($D\backslash G_i$)
Compute $F_{internal}(G_i)$
Compute $F_{external}(G_i)$
Compute FP(G_i)

Fig. 6. Experiment Process for RQ1

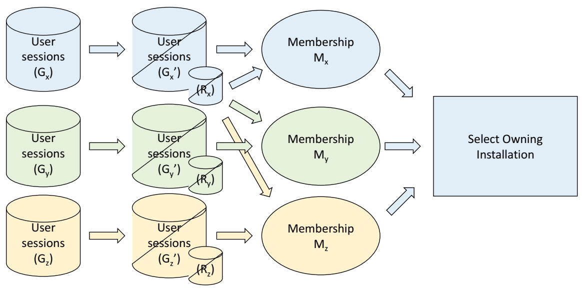


Fig. 7. Experiment Process for RQ2

The user sessions are then split into two groups to simulate what exists before and after a software update is shipped. The set from before the update is used to prioritize the test cases using their fingerprint scores. The second set, from after the update, is used to determine which of the injected faults would have been discovered by users after the update. If a faulty interaction was not invoked by any users, then that fault was not discoverable.

Finally, we use the prioritization of tests based on fingerprints to compute the APFD for the discoverable bugs. The "action match ratio" technique was used in this research question because it was shown to be one of the most effective in RQ1 and RQ2. We also compute the total runtime of the tests (in terms of number of total interactions) necessary to expose all discoverable bugs because test execution time varies widely across test cases. By considering the total runtime to uncover all bugs, we have a better understanding of the effectiveness of the prioritization.

The baseline we compare against is "greedy prioritization" which seeks to prioritize by coverage, that is which test covers the most distinct interactions first. The greedy technique is one of the most successful and often used prioritization techniques in research in this area[12], [15].

IV. Data and Analysis

In this section, we present the results of this study as described in the previous sections.

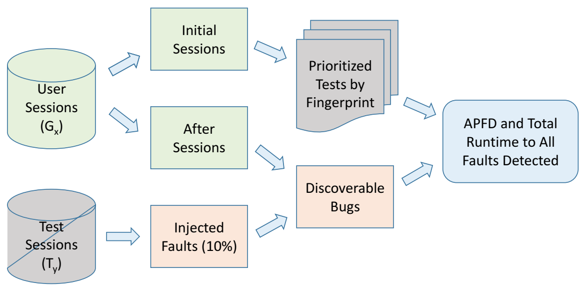


Fig. 8. Experiment Process for RQ3

A. RQ1 Analysis

RQ1 seeks to find algorithms for which there is a high internal to external ratio, yielding a high fingerprint score. A purely random algorithm with no measure of similarity between sessions would yield a fingerprint score (ratio) averaging 1 across repeated experiments. There is an upper bound on fingerprint score as well, since not all sessions from the same installation are identical. Theoretically, the upper bound approaches infinity as internal scores approach 1 and external scores approach 0. Pragmatically, however, this is not possible, unless internal sessions are nearly identical and external sessions are very different. In practice, since customers are all using the same software, there will be an upper limit on any fingerprint score. We do not have a quantification of that limit at this time.

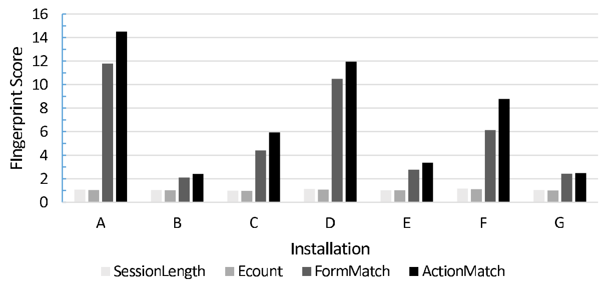


Fig. 9. Fingerprint Score by Installation and Method

Figure 9 shows the fingerprint score by installation and similarity method. As expected, naïve approaches such as comparing by session length or the number of "E" characters in names yield fingerprint scores very close to one. The methods of ratio of form matches and action matches are much better, yielding fingerprint scores up to fifteen.

Also note that the relative effectiveness of each fingerprint method remains relatively constant across installations. Some installations such as “Installation 1” have higher overall values as they are more unique in their usage and therefore easier to fingerprint. “Installation 2” and “Installation 7” in this case are not very unique, yet still exhibit fingerprint scores close to three, meaning they are three times more internally similar than they are similar across installations.

One method not shown in this graph is the similarity by Markov chains. That is because its scores are very skewed, having a fingerprint score up to 10,480 for “Installation 3”. While the fingerprint score is very high, the internal similarity is very low (less than 0.0001 in most cases). It is only because the cross-session similarity is orders magnitude smaller yet that the scores are so large. While this may seem very successful, as we discuss in the next section, there are some drawbacks.

B. RQ2 Analysis

In RQ2, we wish to see if the fingerprint scores discussed in RQ1 have any ability to accurately capture usage patterns. To test this, we randomly took 10 percent of all sessions from each installation and held them as a pool of 350 sessions from unknown installations. We then computed the membership score M_i for each with the remaining 450 sessions in each installation. Whichever installation yielded the highest membership score was selected as the owner. Since we know which installation each session actually came from, this lets us compare the predicted membership to the actual.

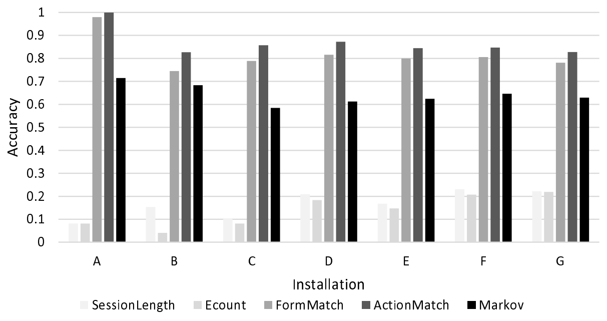


Fig. 10. Accuracy of Owner Installation Prediction

Figure 10 shows the results of this exercise. As expected, we find that session length and E-count techniques were very poor predictors of membership. Random chance yields an 1 in 7 (14 percent) chance of accurate membership selection. Form count and action count on the other hand were very effective, steadily yielding around 80 percent accurate predictions across all installations. From this, we can answer RQ2 is yes, meaning the use of fingerprint scores *does* accurately capture the usage characteristics of an installation.

We like to add one final note of interest. As discussed in the previous section, the Markov technique yielded by far the highest fingerprint scores. However, the accuracy of the prediction of the Markov technique was lower, hovering around 60 percent. While this is good, it is worse than two other methods that had lower fingerprint scores. After manual analysis of the raw data, we found that this can be explained by high variation in the Markov scores. When the Markov technique properly matches two sessions, it yields a very high internal similarity together with very low external similarities. This artificially inflates the fingerprint score. However, for a large portion of the sessions, the Markov technique yields low scores both internally and externally. When averaging fingerprint scores those small number of very high fingerprint scores artificially inflate the average.

C. RQ3 Analysis

RQ3 investigates the applicability of usage fingerprinting in the area of test case prioritization. As previously described, we introduced injected faults in the interactions that tests invoke and then determined which of those faults would be discoverable by at least one user after the update. We then computed the APFD and total test runtime in terms of the number of interactions necessary to expose all discoverable faults. Figure 11 shows the results. The bar graphs show the APFD and the line plots indicate the test runtime. We included one additional installation in the results which became available after the research on RQ1 and RQ2.

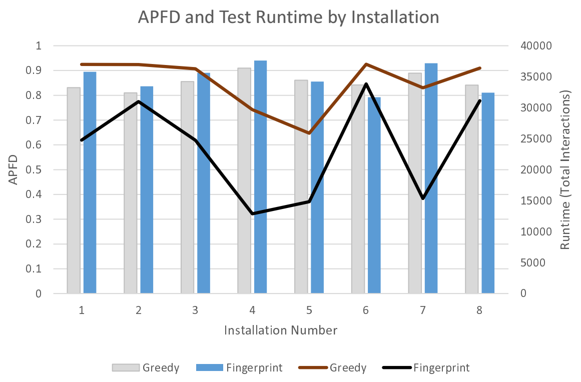


Fig. 11. Accuracy of Owner Installation Prediction

For this experiment, 1,483 tests were used. Those 1,483 tests used a total of 10,610 distinct interactions, thus 1,061 were randomly selected as faulty. This experiment was performed 20 times with random selections each time and the results shown are the average across all runs. The 1,061 faulty interactions were then intersected with the interactions from the “after sessions” for each installation, yielding on average between 20 and 60 discoverable bugs in each case. APFD values were computed by prioritizing all 1,483 test cases based on those 20 to 60 discoverable bugs. The total runtime was computed based on the total sum of interactions in the prioritized test cases until all discoverable bugs were exercised. Thresholds both higher and lower were also tried and yielded similar results. 10 percent was picked as it provided enough discoverable bugs so that APFD values were fairly stable between experiment runs, with a standard deviation of less than 5 percent across all installations.

In 5 of the 8 installations, the prioritization by fingerprint yielded a higher APFD than greedy prioritization. However, all the APFD values were within a few percent of each other. While the average APFD was around 1.5 percent higher for prioritization by fingerprint, it seems the effectiveness of fingerprint prioritization is roughly equivalent to greedy prioritization in terms of APFD.

A more interesting result is the total test runtime by prioritization technique. In all installations, fingerprint prioritization yielded a much smaller runtime to expose all discoverable bugs compared to greedy prioritization. This is because greedy prioritization merely orders by the tests with the most unique interactions first. Those same tests are usually the longest running tests as well. Fingerprint prioritization however focuses on the tests which are actually more likely to contain discoverable bugs. On average the total test runtime was reduced by more than 30 percent using fingerprint prioritization.

V. Discussion

In this section, we discuss the implications of these results, as well as some of the considerations that must be made in attempting to apply the techniques.

A. Comparison with Churn-Based Prioritization

A common prioritization technique for regression testing is to use churn, meaning to prioritize tests that exercise modified code. Churn information is important when a small percentage of the code based has changed, as a large portion of the regression tests can be skipped. This makes churn popular for test selection as well. Test prioritization becomes important when product churn spans a significant portion of the product, such as when a cumulative hotfix is being applied. In these cases churn still selects a large number of tests. In this situation, it is often important to instead prioritize test by some other measure, or prioritize in addition to churn-based selection. In this environment, we feel that fingerprint-based prioritization is an effective additional tool.

B. Application of Techniques

The first obvious question arising from this research is how these results can be applied in practice. For this discussion, we will use *Microsoft Dynamics AX* as a canonical example.

Different installations of the software product will exhibit varying usage characteristics. For instance, a retail company may heavily use item catalog functionality, while a financial services company would not use it at all. That means that regressions in the item catalog module would have little to no impact on the financial services company, while they would cause significant harm to the retail company.

To apply this technique, we try to determine a usage pattern from an installation when testing critical software fixes. A given installation of the *Microsoft Dynamics AX* product in the cloud may span multiple servers, but it is marked by a single "tenant ID," a unique identifier. The source tenant is listed on each telemetry event, allowing a personalized application of regression testing techniques for each installation.

In a business-down critical situation, the risk of a change introducing a new bug must be weighed against the cost of remaining down, which in many cases may be hundreds to thousands of dollars per hour. In these situations, it may make sense to perform preliminary testing, fix the service after only partial testing, and then complete the remainder of testing and analysis once the service is back up. To do this, fingerprint scores could be used to prioritize test cases which match the usage from that business, reducing overall risk.

C. Specialization of Workloads

Fingerprinting is easiest when there are very specialized aspects to a telemetry stream that make identification of similar sessions obvious. This is the case with two of the datasets examined in this study. The company described as "Installation A" in Table II used the *Microsoft Dynamics AX* product as a platform and built a completely custom application on top of it. That means that the forms seen in each session never occurred in any sessions from other datasets. This makes the measures by form count and action count highly selective, as illustrated in the high fingerprint scores in Figure 9 and the high accuracy of ownership identification in Figure 10.

Similarly, the company described as "Installation D" only used a single module of the product. Thus, the relatively small number of interactions from this module made up a majority of this company's sessions and allowed for higher fingerprint scores. This may seem to discount the validity of this research at first, but in practice this is how many companies use this product. Also, if they are only using one of the many modules, prioritization of tests from that module is valuable.

Because only a small number of datasets were available at the time of this research, we would like to examine additional datasets in the future. Markov chains were not nearly as useful as simple form and action name matching, but this was largely due to the high specialization of workloads. We believe that in workloads performing different business processes using the same forms, techniques such as Markov chain analysis would become much better than form and action name matching.

VI. Threats to Validity

The primary threat to validity in this study is the relatively small number of datasets available. The data taken was also for a relative short period of time. It is possible that the usage of the software may change over time. If this is the case, a windowed approach to the data may be necessary to maintain accuracy, as was shown by Anderson et al. [2]. We also wish to extend this work to other applications in the future to determine if it generalizes to all products.

Another threat is the performance characteristics of performing the analysis. Computing the fingerprint scores for all tests takes around an hour per installation. If fingerprint scores for an installation are stable over time as we suspect, then these scores can be computed and stored for future use. If not, the performance of computing the scores may impact the usefulness in optimizing regression testing techniques.

Another risk is the difficulty in finding a proper ground truth with which to measure fingerprinting effectiveness. This paper, as well as traditional research on regression testing techniques, relies on fault detection rates. However, fault detection rates are often *not* the most important predictor of the effectiveness of a technique as they treat all bugs equally while discoverable bugs are more important than non-discoverable bugs. The value a customer places on the detection of a fault is relatively amorphous and is based heavily on how it impacts their business operations. We hope to find better, more concrete measures of ground truth in future research in this area.

VII. Background and Related Work

Regression testing techniques of selection and prioritization have been heavily researched for well over fifteen years. In selection, more important tests are selected to be run while less important tests are skipped. In prioritization, tests are ordered in such a way that faults can be detected earlier, or so the run may be discontinued early if necessary without significantly reducing detection rates. To date, various regression testing techniques have been proposed and recent surveys [6], [9], [17] provide overviews of many different techniques.

Bioinformatics is a growing area of research which uses similar techniques to regression testing, taking large amounts of input data to prioritize treatments or predict diseases. One area of recent interest is "personalized medicine"[10] in which detailed genomic information from a single person is used to custom tailor treatment for that individual. The analog to bioinformatics genomic data in software is telemetry data.

Telemetry has been used in regression testing, most commonly in the area of graphical user interface (GUI) test generation. This includes research by Brooks and Memon [5] and Amalfitano et al. [1]. Brooks and Memon used telemetry from user interface sessions to generate "usage profiles," which are the sequences of events from user sessions. These were then used to generate test cases similar to what users had done in practice. Amalfitano et al. similarly collected usage data from rich Internet applications to build finite state models, which were then used in test case generation. Elmbaum et al. [8] used telemetry from installations for regression prioritization, but focused on prioritization at a product level. In this research, 1,200 user sessions were collected from a 155 KLOC deployed system and then used in regression testing techniques. Their work was extended by others looking at topics like duplicating bug detection (Wang et al. [16]), performance issue detection (Parsons [13]), reliability testing of rule-based systems (Avritizer et al. [3]), and failure reproduction (Jin and Orso [11]).

Our research seeks to apply traditional regression testing techniques based on telemetry data. The difference from the existing work is that we wish to apply those techniques using telemetry from a Software-as-a-Service environment in order to personalize the prioritization to a particular installation. To our knowledge, this method of prioritization is novel and has not yet been explored in research.

VIII. Conclusions

In this research, we explored a novel concept of fingerprinting software usage in order to improve regression testing techniques. The empirical results with an industrial application have shown that fingerprint algorithms based on usage telemetry can be identified, that those fingerprints are effective in identifying usage patterns, and that the fingerprints can be applied to improve regression testing techniques. We believe that the proposed techniques are a promising way of applying traditional regression testing techniques in a new software world of Software-as-a-Service in which telemetry is pervasive and downtime is unacceptable. If software testing can be customized to the usage in a particular installation, downtime costs can be reduced while quality is retained. We also believe that this will become increasingly important as software velocity continues to increase.

For future work, we wish to continue and expand on this research as more data becomes available across a wider range of customers and usages. While the preliminary results are promising, we would like to see if these results generalize to a large number of active customers. It will also be interesting to see how this research applies to different types of telemetry streams coming from different software services.

Acknowledgment

This work was supported, in part, by NSF CAREER Award CCF-1564238 to University of North Texas.

References

[1] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops*, pages 274–283, 2010.

[2] J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 142–151, 2014.

[3] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Software*, 13(5):76, 1996.

[4] R. V. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Upper Saddle River, NJ, 1999.

[5] P. A. Brooks and A. M. Memon. Automated gui testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, 2007.

[6] C. Catal and D. Mishra. Test case prioritization: A systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.

[7] D. Durkee. Why cloud computing will never be free. *Queue*, 8(4):20, 2010.

[8] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[9] E. Engstrom, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14 – 30, 2010.

[10] M. A. Hamburg and F. S. Collins. The path to personalized medicine. *New England Journal of Medicine*, 363(4):301–304, 2010.

[11] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 474–484. IEEE Press, 2012.

[12] S. Mirarab, S. Akhlaghi, and L. Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE Transactions on Software Engineering*, 38(4):936–956, 2012.

[13] T. Parsons. *Automatic detection of performance design and deployment antipatterns in component based enterprise systems*. PhD thesis, Citeseer, 2007.

[14] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Int'l. Conf. Softw. Maint.*, pages 179–188, Aug. 1999.

[15] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[16] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *International Conference on Software Engineering*, pages 461–470, 2008.

[17] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation : A survey. *Software Testing, Verification, and Reliability*, Mar. 2010.