Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks

Murali Haran, Alan Karr, Michael Last, Alessandro Orso, *Member, IEEE,*, Adam Porter, *Senior Member, IEEE,*, Ashish Sanil, Sandro Fouché, *Student Member, IEEE,*

***Abstract*— There is an increasing interest in techniques that support analysis and measurement of fielded software systems. These techniques typically deploy numerous instrumented instances of a software system, collect execution data when the instances run in the field, and analyze the remotely-collected data to better understand the system's in-the-field behavior. One common need for these techniques is the ability to distinguish execution outcomes (e.g., to collect only data corresponding to some behavior or to determine how often and under which condition a specific behavior occurs). Most current approaches, however, do not perform any kind of classification of remote executions and either focus on easily observable behaviors (e.g., crashes) or assume that outcomes' classifications are externally provided (e.g., by the users). To address the limitations of existing approaches, we have developed three techniques for automatically classifying execution data as belonging to one of several classes. In this paper, we introduce our techniques and apply them to the binary classification of passing and failing behaviors. Our three techniques impose different overheads on program instances and, thus, each is appropriate for different application scenarios. We performed several empirical studies to evaluate and refine our techniques and to investigate the tradeoffs among them. Our results show that (1) the first technique can build very accurate models, but requires a complete set of execution data; (2) the second technique produces slightly less accurate models, but needs only a small fraction of the total execution data; and (3) the third technique allows for even further cost reductions by building the models incrementally, but requires some sequential ordering of the software instances' instrumentation.**

Keywords: Execution classification, remote analysis/measurement

I. INTRODUCTION

Several research efforts are focusing on tools and techniques to support the remote analysis and measurement of software systems (RAMSS) [1]–[13]. In general, these approaches instrument numerous instances of a software system, each in possibly different ways, and distribute the instrumented instances to a large number of remote users. As the instances run, execution data are collected and sent to one or more collection sites. The data are then analyzed to better understand the system's in-the-field behavior.

M. Haran is with the Department of Statistics, Penn State University, 326 Thomas Building University Park, PA 16802.

A. Karr, M. Last and A. Sanil did this work while at the National Institute of Statistical Sciences, 19 T. W. Alexander Drive, Research Triangle Park, NC 27709-4006.

A. Orso is with the Georgia Institute of Technology. Atlanta, GA

A. Porter and S. Fouche are with the Computer Science Department at the University of Maryland, College Park, MD 20814.

RAMSS techniques typically collect different kinds of data and use different analyses to achieve specific software engineering goals. One characteristic common to many of these techniques is a need to distinguish execution outcomes. There are many scenarios in which this information is useful. For example, remote analyses that use information from the field to direct debugging effort need to know whether that information comes from a successful or failing execution (e.g., [7], [14]). For another example, self-managing applications that reconfigure themselves when performance is degrading must be able to determine when the system has entered a problematic state. Modeling remote execution outcomes could also be useful for identifying specific problematic behaviors in order to prioritize debugging efforts.

Despite many recent advances, existing techniques suffer from numerous problems. First, they often make oversimplifying assumptions (e.g., they equate failing behaviors with system crashes) or assume that the classification of the outcome is provided by an external source, such as the users. These assumptions severely limit the kinds of program behaviors that can be analyzed and the techniques' applicability. Second, these techniques often impose significant overheads on every participating program instance. Example overheads include code bloat due to code rewriting, bandwidth occupation due to remote data collection, and slowdown and perturbed performance due to code instrumentation.

This paper proposes and evaluates three new techniques for automatically classifying execution data, collected from deployed applications, as coming from runs having different outcomes. To be able to perform controlled experimentation and to suitably validate our results, in this paper we focus our empirical investigation on binary outcomes only: "pass," which corresponds to executions that produce the right results, and "fail," which corresponds to incorrect executions. The techniques are, however, equally applicable to any discrete set of outcomes.

Our techniques use statistical learning algorithms to model and predict execution outcomes based on *runtime execution data*. More specifically, the techniques build behavioral models by analyzing execution data collected from one or more program instances (e.g., by executing test cases in-house or in the field or by examining fielded program instances running

under user control). Developers can then either analyze the models directly (e.g., for fault localization) or use the models to gather further information from other program instances. In the latter case, they lightly instrument numerous instances of the software (i.e., the instrumentation captures only the small subset of execution data referenced by the behavioral model) and distribute them to users who run them in the field. As the instances run, the collected execution data is fed to the previously built model to predict execution outcomes.

In previous work, we defined an initial classification technique and performed a three-part feasibility study [15]. Our initial approach is effective and efficient in cases where we can fully train the models in-house, using accurate and reliable oracles. However, the approach is not applicable if part (or all) of the training must be performed on deployed instances because it imposes too much time and space overhead during the training phase. To address this issue, we extend our initial work by developing and evaluating two improved classification techniques that can build models with substantially less data than that required by our initial technique, while maintaining its accuracy. The first new technique can build reliable models while observing less than 10% of the complete execution data. Each instance collects a different subset of the execution data, chosen via uniform random sampling. The second technique is also able to reliably classify executions based on a small fraction of execution data, but it adds the ability to adapt the sampling over time so as to maximize the amount of information in the data.

In the paper, we also present several empirical studies in which we applied these techniques to multiple versions of a medium-sized software subject and studied the techniques' performance. The goal of the studies was to evaluate the techniques, better understand several issues crucial to their success and, thereby, refine the techniques and improve their ultimate implementations. The first set of studies looks at whether it is possible to reliably classify program executions based on readily-available execution data, explores the interplay between the type of execution data collected and the accuracy of the resulting classification models, and investigates how much data is actually necessary for building good classification models. The second and third studies examine the extent to which our two newly defined classification techniques allow for minimizing data collection (and data collection overheads), while maintaining the accuracy of the classification.

The main contributions of this paper are:

- A high-level vision for an approach that automatically and accurately classifies execution data, collected with low overhead from fielded programs, as coming from runs with specific execution outcomes.
- Three instantiations of the approach, two of which are based on newly-defined classification techniques, that can classify execution data according to a binary outcome.
- An empirical evaluation of several key issues underlying these (and similar) techniques as well as an evaluation of the instantiated techniques themselves.

II. Background and Motivation

In this section, we first provide background information on techniques for classifying program executions and then motivate our research by presenting three applications of classification techniques to support software engineering tasks.

A. Classification of program executions

Machine learning techniques are concerned with the discovery of patterns, information and knowledge from data. They often work by analyzing a training set of data objects, each described by a set of measurable *features* (also called *predictors*[1]), and by concisely modeling how the features' values relate to known or inferred higher-level characterizations. These models can then be used to predict the characterizations of data objects whose characterizations are unknown. Supervised learning is a class of machine learning techniques in which the characterization of each training set object is known at model building time. When the possible characterizations come from a discrete set of categorical values (e.g., "good," "average," and "bad"), supervised learning is called *classification*. In this work we focus on classification techniques.

Classifying program executions means using readily-available execution data to model, analyze, and predict (more difficult to determine) program behaviors. Figure II-A shows a high-level view of the classification approach that we use in this research. In the training phase, we instrument program instances to collect execution data at runtime and, in the case of collecting data in the field, attach a built-in oracle to the deployed instances. Then, when the instances run, we collect the resulting data. The figure shows two extremes of training phase data collection. In one case, data collection is performed in-house on instances running existing test cases. In the other case, it is performed in the field on instances running under user control. In the former case, a traditional oracle can be used to label each run based on its outcome (e.g., "high," "average," and "low throughput," if the behavior of interest is performance; "pass" or "fail" if the behavior of interest is functional correctness). In the latter case, the execution data would be collected while the instances run on the users' platforms against actual user input, and each run would be labeled by a built-in oracle attached to the deployed programs.

Note that, in these two cases, there is a clear trade-off between strength of the oracle and completeness of the data. In-house, we can often rely on accurate oracles because we usually have complete control over the execution and the (computational and space) overhead of the data collection is usually not an issue. However, in-house we can observe only a limited number of behaviors (the ones exercised by the test inputs available and occurring in the hardware and software configurations available), and the models constructed from these observations may not be representative of real executions. In the field, we must typically rely on limited oracles to limit the overhead. On the other hand, field executions are typically much more numerous, varied, and representative

[1] Hereafter, we use the terms feature and predictor interchangeably.

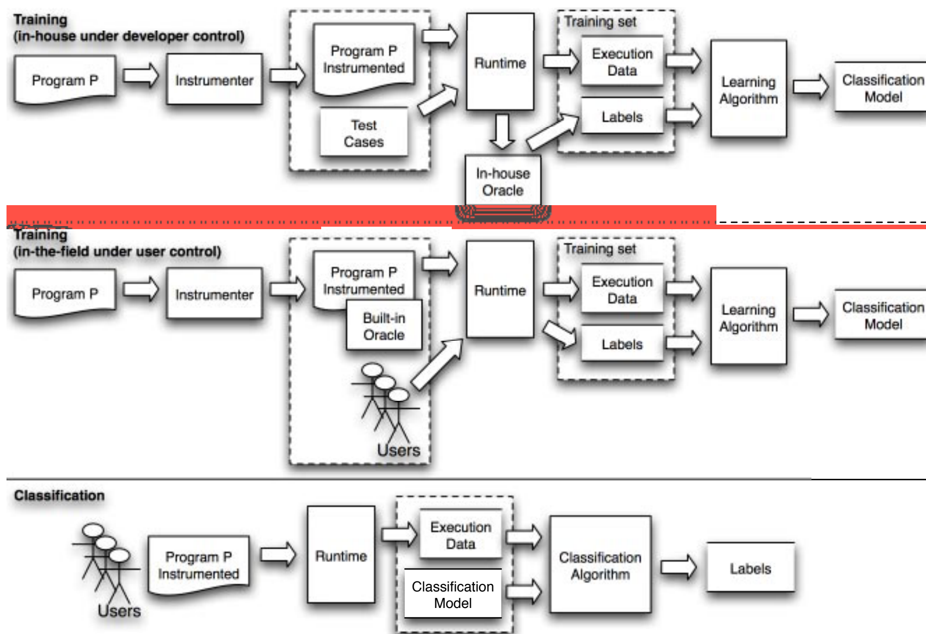

Fig. 1. Overview of our general approach.

than in-house test runs. Which approach is more appropriate depends on the task at hand, as discussed in Section II-B.

Once the labeled training-phase data has been collected, it is fed to a learning algorithm, which analyzes it and produces a classification model of program executions.

In the classification phase, the code is re-instrumented to capture only the data needed by the models. When the instances are later run in the field by actual users, appropriate execution data are collected and fed to the previously-built classification model to predict the label associated with the current execution.

To carry out these processes, developers must consider several issues. First, they must determine which **specific behaviors** they want to classify. For instance, one might want to identify previously-seen behaviors, such as open bugs identified during regression testing, or previously-unseen behaviors, such as potential performance problems on less popular hardware platforms. The specific application considered drives several process choices, including:

- *The outcomes that must be detected.* Developers may want to classify executions in which the system crashes, exceptions are thrown, incorrect responses are given, transaction times are too long, and so on. There may be only two outcomes (e.g., "pass" or "fail") or multiple outcomes. Developers must create oracles and measurement instruments that make these outcomes observable.
- *The environments in which the system must run.* Developers may want to observe executions on different operating systems or with the system in different configurations.
- *The range of inputs over which system execution will be monitored.* In some cases, developers may be interested in a small set of behaviors captured by a specific test suite. In others, they may want to see the system execute under actual end-user workloads.

Second, developers must determine the **execution data** on which the classification will be based. Many different types of execution data can be considered, such as execution counts, branch frequencies, or value spectra. Developers must create appropriate instruments to capture these different kinds of data and must be aware of the amount of data they will capture. The more data captured, the higher the runtime overhead and the more resources needed to analyze the resulting data.

Third, developers must decide the **location** where training-phase data collection occurs—in-house or in-the-field. As stated above, we assume that fielded data collection can be done on many more program instances than in-house data collection and that fielded instances cannot tolerate as much data collection volume as in-house instances.

Finally, developers must decide which **classification technique** to use to create the classification models. There is a vast array of established classification techniques, each with its strengths and weaknesses. They range from classical statistical methods, such as linear and logistic regression, to neural network and tree-based techniques (e.g., [16], [17]), to the more recent Support Vector Machines [18].

Figure 2 shows, in an intuitive way, how the various process decisions described above are intimately intertwined. For example, developers who want to understand possible performance problems experienced by end users may want to monitor different execution environments and observe actual usage patterns. As a result, they are forced towards in-the-field data collection, which in turn tends to limit the volume of data collected by each instance and necessitates observing many instances. For another example, developers may be interested in collecting execution paths corresponding to failing runs of an existing regression test suite to study and understand specific open bugs. In this case, the developers may opt for using sophisticated, heavyweight test oracles and for collecting substantial amounts of data in-house on a few instances. In either case, the learning technique chosen must be suitable for

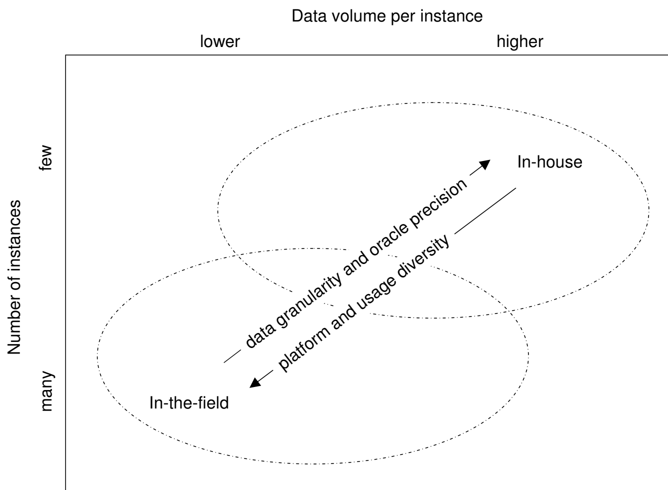

Fig. 2. Building a classification process.

the type of data collected.

B. Scenarios

We present three techniques for classifying executions of deployed programs. To provide context for their evaluation, we discuss three possible scenarios where they might be used:

1) Automated Identification of Known Failures in Deployed Programs: In this scenario, developers want to automatically distinguish fielded program executions that succeed or that fail in one of several known modes. Such information could be used in many ways. For instance, it could be used to automatically trigger additional data collection and reporting when a program fails. The information could also be used to measure the manifestation frequencies of different failures or to identify system configurations in which specific failures occur. If failures can be predicted early, this information could also be used to trigger failure avoidance measures. One way to implement this scenario might be to test a program in-house with a known test suite, collect execution data and build classification models for each known failure, and attach the resulting models to fielded instances to predict whether and how a current run fails (or will likely fail).

This scenario sits in the upper right corner of Figure 2. Because developers will train the classification technique in-house for known failures, they are free to collect a substantial amount of data per instance and can use heavyweight oracles. On the other hand, they will be limited to instrumenting relatively few instances and observe a narrower range of platforms and usage profiles than they could if they instrumented fielded instances. In Section IV, we describe how our first technique, based on random forests classifiers, can support this scenario.

2) Remote Failure Modeling: In this scenario, developers are still interested in modeling passing and failing executions. However, they want to collect the training data from fielded instances running under user control, rather than from in-house instances running existing test suites. The goal of the data collection is in this case to perform some failure analysis, such as debugging.

Developers might implement this scenario by instrumenting fielded instances and then collecting labeled execution data from them. The labels would indicate various outcomes, such as “successful”, “crashed,” “not responding,” or “exception X thrown at line Y.” The specific labels used, and the degree of confidence in the labeling, would depend on the oracles installed in the fielded instances. The collected data is then classified to relate execution data patterns with specific outcomes. At this point, developers might examine the models directly and look for clues to the outcome’s cause. (See discussion of Liblit et al. [6] in Section VII.) Alternatively, developers might execute a large number of test cases in-house and use the models to flag exemplars of a given outcome, that is, to find test cases whose behavior in-house is similar to that of fielded runs with the same outcome. This approach might be especially useful when multiple root causes lead to the same outcome (e.g., different failures leading to a system crash).

In Figure 2, this scenario lies below and to left of the previous one. In this case, developers collect data in the field to model previously unseen failures. To avoid affecting users, developers need to limit their data collection per instance and must use lighter-weight oracles than they could have used if operating entirely in-house. On the other hand, they can instrument many instances and can observe a wide range of platforms and usage profiles. Section V describes how our second technique, based on association tree classifiers, can be used to support this scenario.

3) Performance Modeling of Field Executions: In this scenario, developers want to investigate the causes of a performance problem believed to be due to system aging (e.g., memory leaks or improperly managed locks). Because these kinds of problems occur infrequently and may take a long time to manifest, developers instrument a large number of instances in batches—with each batch collecting data over increasingly longer time periods—hoping to increase the chances of observing the problem early. They can achieve this goal by lightly instrumenting deployed program instances running under user control and modeling the collected data to predict incidences of the performance problem. The models can then be attached to program instances deployed at beta test sites, where they will trigger deeper data collection when impending performance troubles are predicted (this scenario assumes that predictions can be made before the execution ends.)

This scenario lies at the bottom left of Figure 2. In this case, developers must be particularly careful to limit data collection overhead to avoid overly perturbing the instance’s performance. They must likewise use very simple oracles. However, they can instrument many instances and are likely to observe a wide range of platforms and usage profiles. One new aspect of this scenario is that, because each run can take a long time to complete, some incremental analysis might help limit overall costs. Section VI describes our third technique, based on adaptive sampling association tree classifiers, which can be used to support this scenario.

III. Experimental Subject and Data

As part of this research, we designed and conducted several empirical studies to guide the development of the techniques,

evaluate them, and improve our understanding of the issues underlying our proposed approaches. To perform controlled experiments, we used the same subject for all studies and targeted a version of the general classification problem involving a behavior that we can measure using an accurate oracle: passing and failing execution outcomes. Therefore, our executions have one of two possible labels: "pass" or "fail." In this section we introduce our experimental subject and data.

A. Experimental Subject

As a subject program for our studies, we used JABA (Java Architecture for Bytecode Analysis),[2] a framework for analyzing Java programs that consists of about 60,000 lines of code, 400 classes, and 3,000 methods. JABA performs complex control-flow and data-flow analyses of Java bytecode. For instance, it performs stack simulation (Java is stack based) to track the types of expressions manipulated by the program, computes definitions and uses of variables and their relations, and analyzes the interprocedural flow of exceptions.

We selected JABA as a subject because it is a good representative of real, complex software that may contain subtle faults. We considered 19 real faults extracted from JABA's CVS repository by a student in a different research group. The student inspected all CVS commits starting from January 2005, identified the first 19 bug fixes reported in the CVS logs, and distilled the related faults as source-code differences. We then selected the latest version of JABA as our golden copy of the software and generated 19 different versions by inserting one fault into the golden copy. In this way, we were able to use the golden copy as an accurate oracle. We also created nine versions of JABA containing multiple faults.

B. Execution Data and Labels

To build a training set for the JABA versions considered, we used the executions of the test cases in JABA's regression test suite. Because JABA is an analysis library, each test consists of a driver that uses JABA to perform one or more analyses on an input program. There are 7 such drivers and 101 input programs, divided into real programs (provided by users) and ad-hoc programs (developed to exercise a specific functionality). Thus, overall, there are 707 test cases. The entire test suite achieves about 60% statement coverage and 40% method coverage.

For each of the versions, we ran the complete regression test suite and collected (1) information about passing and failing test cases, and (2) various types of execution data. In particular, we collected statement counts, branch counts, call-edge counts, throw and catch counts, method counts, and various kinds of value spectra (e.g., relations between variable values at methods' entry and exit or maximum values of variables). The outcome of each version v and test case t was stored in binary form: "1" if the execution of t on v terminated and produced the correct output; "0" otherwise. Because the test drivers output, at the end of each test-case execution, an XML version of the graphs they build, we were able to identify

[2]http://www.cc.gatech.edu/aristotle/Tools/jaba.html

TABLE I

JABA VERSIONS USED IN THE STUDIES.

Version	Faults	Total failures	
1	1	0	(0%)
2	2	12	(1.7%)
3	3	0	(0%)
4	4	372	(52.6%)
5	5	138	(19.5%)
6	6	0	(0%)
7	**7**	144	(20.4%)
8	8	0	(0%)
9	9	86	(12.2%)
10	10	0	(0%)
11	11	69	(9.8%)
12	12	4	(0.6%)
13	13	38	(5.4%)
14	14	14	(2%)
15	15	0	(0%)
16	16	30	(4.2%)
17	17	105	(14.9%)
18	18	0	(0%)
19	19	21	(3%)
20	1 3	0	(0%)
21	2 17	113	(16%)
22	9 12	90	(12.7%)
23	12 13 19	60	(8.5%)
24	7 13 16 17	231	(32.7%)
25	5 7 11 19	207	(29.3%)
26	2 5 9 12 19	206	(29.1%)
27	9 12 13 16 17	200	(28.3%)
28	5 9 11 12 13 19	244	(34.5%)

failures of t for v by comparing the golden version's output to that produced by t when run on v. (We canonicalize the XML representation to eliminate spurious differences).

Table I summarizes the distribution of failures across the different versions. Each row in the table shows the version number (Version), the list of faults included in each version (Faults included) and the total number of failures (Total failures), both in absolute terms and in percentage. Versions one to 19 are single-fault versions. Versions 20 to 28 contain multiple faults, and column "Faults included" lists the single-fault version numbers whose faults were used to create this multi-fault version. For example, version 28 contains 6 faults: those taken from versions 5, 9, 11, 12, 13, and 19. Versions with a failure rate greater than 8% are highlighted in boldface. The meaning of this is explained in Section IV-B.1.

IV. CLASSIFICATION USING IN-HOUSE DATA COLLECTION

Our goal in this part of the work is to define a technique that can classify executions of deployed programs using models built from data collected in-house. As illustrated in the upper part of Figure II-A, the models would be built by collecting execution data in-house and using an accurate oracle to label these executions. The oracle could be of different kinds, such as a golden version of the program, an ad-hoc program, or a human oracle. This scenario makes sense when attaching oracles to deployed programs would be too demanding in

terms of resources (space, computational, or both) or in terms of setup costs, which is typically the case for accurate oracles.

As stated in Section III, we consider classification of remote executions as passing or failing executions. The other two aspects that we need to define within our approach are the machine-learning technique to use and the kind of execution data to consider. The family of learning techniques that we use to define our approach is tree-based classifiers. In terms of execution data, we consider different control- and value-related types of execution information and assess their predictive power using an empirical approach.

In the following sections, we first describe our approach based on tree-based classifiers. Then, we discuss the empirical studies that we performed to assess and refine the approach.

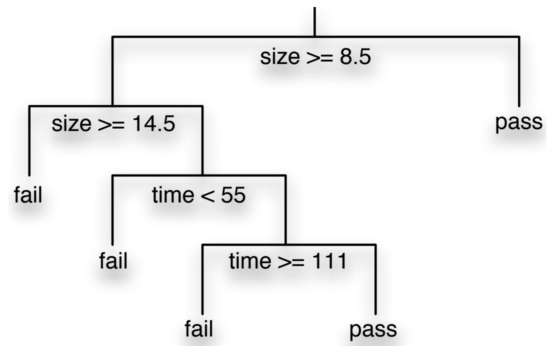

Fig. 3. Example of tree classifier for executions with two features (i.e., predictors), *size* and *time*, and two possible labels, "pass" and "fail."

A. Random Forests Classifiers

Tree-based classifiers are an especially popular class of learning techniques with several widely-used implementations, such as CART [19] and ID4 (see `http://www.rulequest.com`). These algorithms follow a recursive partitioning approach which subdivides the predictor-space into (hyper) rectangular regions, each of which is assigned a predicted outcome label ("pass" or "fail" in our case). The resulting models are trees in which each non-leaf node denotes a predicate involving one predictor, each edge represents the *true* or *false* branch of a predicate, and each leaf node represents a predicted outcome. Based on a training set of data, classification trees are built as follows:

1) For each predictor, partition the training set based on the observed ranges of the predictor data.
2) Evaluate each potential partition based on how well it separates failing from passing runs. This evaluation is often realized based on an entropy measure [19].
3) Select the range that creates the best partition and make it the root of the tree.
4) Add one edge to the root for each subset of the partition.
5) Repeat the process for each new edge. The process stops when further partitioning is impossible or undesirable.

To classify new observations, tree classifiers identify the region to which that observation belongs; the predicted outcome is the outcome label associated with that particular region. Specifically, for each execution we want to classify, we begin with the predicate at the root of the tree and follow the edge corresponding to the value of the corresponding predictor in the execution data. This process continues until a leaf is encountered. The outcome label found at the leaf is interpreted as the predicted outcome for the new program run.

For example, Figure 3 shows a hypothetical tree-classifier model that predicts the "pass"/"fail" outcome based on the value of the program running time and input size. The decision rules prescribed by the tree can be inferred from the figure. (While traversing the tree, by convention, we follow the left edge when the predicate is true and the right edge otherwise.) For instance, an execution with $Size < 8.5$ would be predicted as "pass", while if $(8.5 \leq Size < 14.5)$ AND $(Time < 55)$, the execution would be predicted as "fail."

The main reason why we chose tree-based classifiers over other classification approaches is that they create interpretable prediction models. However, because the model are built by greedy procedures, they can be quite sensitive to minor changes in the training data [20]. To address these problems, we use a generalization of tree-based classification, called random forests, as our classification technique. Random forests is an ensemble learning method that builds a robust tree-based classifier by integrating hundreds of different tree classifiers via a voting scheme. This approach maintains the power, flexibility and interpretability of tree-based classifiers, while greatly improving the robustness of the resulting model. Consider the case in which we have M predictors and a training set with N elements. We *grow* (i.e., incrementally create) each tree classifier as follows:

1) Sample N cases at random with replacement (*bootstrap sample*) from the original data. This sample will be the training set for growing the tree.
2) Specify a number $m << M$ such that, at each tree node, m variables are selected at random out of the M, and the best split on these m is used to split the node.[3]

The forest consists of a large set of trees (500 in our approach), each grown as described above. For prediction, new input is fed to the trees in the forest, each of which returns a predicted classification label. The most selected label is returned as the predicted label for the new input. In the case of a tie, one of the outcomes is arbitrarily chosen.

Random forests have many advantages. First, they efficiently handle large numbers of variables. Second, ensemble models are quite robust to outliers and noise. Finally, the random forests algorithms produce error and variable-importance estimates as a byproduct. We use the error estimates to study the accuracy of our classifiers and use the variable importance estimates to determine which predictors must be captured (or can be safely ignored) in the field in order to classify executions.

[3]A split is a division of the samples at the node into subsamples. The division is done using simple rules based on the m selected variables.

B. Experimental Refinement of the Technique

To evaluate and refine the initial definition of our classification technique, we applied our approach to the subject and data presented in Section III. In the study, we excluded versions with error rates below an arbitrarily chosen cutoff of 8%, which effectively removed only four versions from consideration (versions with zero failures would not be considered anyway). Admittedly, an 8% failure rate is much higher than what we would expect to see in practice, which may affect the generality of our results. Nevertheless, we felt we had to take this step for several reasons. In particular, we would have needed many more test cases (which would have been prohibitively expensive to generate) in order to reliably classify program versions with lower failure rates. There are special machine learning techniques that have been developed to handle this kind of situation and, as we discuss in Section VII, could be grafted onto our technique in future studies. However, integrating such techniques now would only make our initial analyses more difficult to interpret.

As discussed above, our technique could be instantiated in many ways, depending on the different types of execution data considered. Instead of simply picking a possible instance of the technique and studying its performance, we used an empirical approach to evaluate different possible instances. To this end, we designed and conducted a multi-part empirical study that explored three main research questions:

RQ1: Can we reliably classify program outcomes using execution data?

RQ2: If so, what kinds of execution data should we collect?

RQ3: Is all the data we collect actually needed to produce accurate and reliable classifications?

We addressed RQ1 by measuring classification error rates. We addressed RQ2 by examining the relationship between classification error rates and the type of execution data used in building classification models. We addressed RQ3 by examining the effect of predictor screening (i.e., collecting only a subset of the predictors) on classification error rates. In the following sections, we describe the design, methodology and results of our exploratory studies in detail.

1) Empirical Design: Initially, we considered only single-fault versions of our subject (see Table I). (The study involving multiple faults is discussed in Section IV-B.5.) For each program version and type of execution data collected, we fit a random forest of 500 classification trees using only predictors of that type. We then obtained the most important predictors by using the variable importance measures provided automatically by the random forest algorithm, and identified the subset of important predictors that resulted in the lowest error rate.

For each resulting classification model we computed an error estimate, called the *Out Of Bag* (*OOB*) errors estimate. To compute this quantity, the random forest algorithm constructs each tree using a different bootstrap sample from the original data. When selecting the bootstrap sample for the kth tree, only two-thirds of the elements in the training set are considered (i.e., these elements are in the bag). After building the kth tree, the one-third of the training set that was not used to build the tree (i.e., the OOB elements) is fed to the tree and classified. Given the classification for an element n obtained as just described, let j be the class that got most of the votes every time n was OOB. The OOB error is simply the proportion of times that j is not equal to the actual class of n (i.e., the proportion of times n is misclassified), averaged over all elements. This evaluation method has proven to be unbiased in many studies. More detailed information and an extensive discussion of this issue is provided in Breiman's original paper [20].

2) Study 1 – Research Question 1: The goal of this first study is to assess whether execution data can be used at all to predict the outcome of program runs. To do this, we selected one obvious type of execution data, statement counts (i.e., the number of times each basic block is executed for a given program run), and used it within our technique. We chose statement counts because they are a simple measure and capture diverse information that is likely to be related to various program failures. For each JABA version, there are approximately 12,000 non-zero statement counts, one for each executed basic block in the program. Following the methodology described in Section IV-A, we built a classification model of program behavior for each version of the subject program. We then evaluated those models by computing OOB error estimates and found that statement counts were nearly perfect predictors for this data set: almost every model had OOB error rates near zero. This result suggests that at least this one kind of execution data might be useful in predicting program execution outcomes.

Although statement counts were good predictors, capturing this data at user sites can be expensive. For our subjects, instrumentation overhead accounted for an increase around 15% in the total execution time. While this might be acceptable in some cases, it is still a considerable slowdown that may not be practical for many applications. Moreover, the amount of information collected, one integer per basic block, can add considerable memory and bandwidth overhead for large programs and large numbers of executions.

3) Study 2 – Research Question 2: In Study 2, we investigate whether other kinds of (more compact and cheaper to collect) execution data can also be used to reliably estimate execution outcomes. Using statement counts as a starting point, we investigated whether other data might yield similar prediction accuracy, but at a lower runtime cost. Note that, because statement counts contained almost perfect predictors, we did not consider richer execution data, such as data values or paths. Instead, we considered three additional kinds of data that require the collection of a smaller amount of information: throw counts, catch counts, and method counts.

Throw Counts and Catch Counts: Throw counts measure the number of times each throw statement is executed in a given run. Analogously, catch counts measure the number of times each catch block is executed. Each version of JABA has approximately 850 throw counts and 290 catch counts, but most of them are always zero (i.e., the corresponding throw and catch statements are never exercised). This is a typical situation for exception handling code, which is supposed to be invoked only in exceptional situations.

As with statement counts, we built and evaluated classi-

fication models using throw counts as predictors. We found that throw counts are excellent predictors for only one version (v17), with error rates well below 2%, but are very poor predictors for all other versions. Further examination of the fault in v17 provided a straightforward explanation of this result. Fault #17 causes a spurious exception to be thrown almost every time that the fault is executed and causes a failure. Therefore, that specific exception is an almost perfect predictor for this specific kind of failure. Most of the other throw counts refer to exceptions that are used as shortcuts to rollback some operations when JABA analyzes certain specific program constructs. In other words, those exceptions are used to control the flow of execution and are suitably handled by `catch` blocks in the code, so they are typically not an indicator of a failure.

The results that we obtained using catch count predictors were almost identical to those obtained using throw counts. Overall, it appears that, for the data considered, throw and catch counts do not by themselves provide wide predictive ability for different failures. Although this may be an artifact of the specific subject considered, we believe that the results will generalize to other subjects. Intuitively, we expect throw (and catch) counts to be very good predictors for some specific failures (e.g., in the trivial case of executions that terminate with fatal failures related to explicitly-thrown exceptions). We do not expect them to predict reliably other kinds of (more subtle) failures and to work well in general, which is consistent with what we have found in our study.

Method Counts: Method counts measure the number of times each method has been executed in a given run. For each version of JABA considered, there are approximately 3,000 method counts—one for each method in the program. As for statement counts, the random forest algorithm considered, among these 3,000, only the 1,240 non-zero method counts. The models built using method counts performed extremely well for all program versions. Like with statement counts, method counts led to models with OOB error rates near zero. Interestingly, these results are obtained from models that use only between two and seven method count predictors (for each program version). Therefore, method counts were as good predictors as statement counts, but had the advantage of being much less expensive to collect.

More generally, as these results suggest, there are several kinds of execution data that may be useful for classifying execution outcomes. In fact, in our preliminary investigations, we also considered several other kinds of data. For example, we considered branch counts and call-edge counts. Branch counts are the number of times each branch (i.e., method entries and outcomes of decision statements) is executed. Call-edge counts are the number of times each call edge in the program is executed, where a call edge is an edge between a call statement and the entry to the called method. Both branch counts and call-edge counts were as good predictors as statement or method counts.

Note that the execution data that we considered are not mutually independent. For example, method counts can be computed from call-edge counts and throw counts are a subset of statement counts. It is also worth noting that we initially considered value-based execution data and captured data about the values of specific program variables at various program points. However, we later discarded these data from our analysis because the compact and easy-to-gather count data defined above yielded almost perfect predictors. In particular, because method counts are excellent and fairly inexpensive to collect, we decided to consider only method counts for the rest of our investigation. (There is an additional reason to use method counts, which is related to the statistical validity of the results, as explained in Section IV-B.5.)

4) Study 3 – Research Question 3: The results of Study 2 show that our approach could build good predictors consisting of only a small number of method counts (between two and seven). This finding suggests that, at worst, our technique needs to instrument less than 130 of the 3,000 methods (assuming 7 different methods over 19 faulty versions) to perform an accurate classification. We use this result as the starting point for investigating our third research question.

One possible explanation for this result is that only these few counts contain the relevant "failure signal." If this is the case, choosing exactly the right predictors is crucial. Another possibility is that the signal for program failures is spread throughout the program, and that multiple counts carry essentially the same information (i.e., they form, in effect, an equivalence class). In this case, many different predictors may work equally well, making it less important to find exactly the right predictors. Moreover, if many different predictor subsets are essentially interchangeable, then lightweight instrumentation techniques are more likely to be widely applicable.

To investigate this issue, we randomly sampled a small percentage of the method counts and investigated the predictive power of this small subset. More precisely, we (1) randomly selected 1% (about 30) and 10% (about 300) of the method counts, (2) built a model based only on these counts, and (3) validated the model as described in Section IV-B.1. We repeated this experiment 100 times, selecting different 1% and 10% subsets of method counts every time. This approach is an effective way to discover if there are many sets of common predictors that are equally significant. Without random sampling, it is easy to be misled into identifying just a few important predictors, when in fact there are other predictors which have comparable predictive capabilities. Also, random sampling provides a way of assessing how easy (or difficult) it may be to find good predictors. For instance, if 1% subsamples return a good subset of predictors 90% of the time, the number of equally good predictors is very high. Conversely, if 10% subsets contain good predictors only 5% of the time, we would conclude that good predictors are not as easily obtained.

We found that the randomly selected 1% and 10% of method counts invariably contained a set of excellent predictors over 80% and 90% of the time, respectively. This result suggests that many different predictor subsets are equally capable of predicting passing and failing executions. This is especially interesting because most previous research has assumed that one should capture as much data as possible at the user site, possibly winnowing it during later post-processing. Although ours is still a preliminary result, it suggests that large amounts of execution data might be safely ignored, without hurting

prediction accuracy and greatly reducing runtime overhead on user resources. (We actually measured the overhead imposed by collecting such small subsets of execution data, and verified that it is almost negligible in most cases.)

5) Possible Threats to the Validity of the Studies:

Generality Issues: All the results presented so far are related to the prediction of the outcomes within single versions. That is, each model was trained and evaluated on the same version of the subject program. Although a classification approach that works on single versions is useful, an approach that can build models that work across versions is much more powerful and applicable. Intuitively, we can think of a model that works across versions (i.e., a model that can identify failures due to different faults) as a model that, to some extent, encodes the concept of "correct behavior" of the application. Conversely, a model that works on a single version and provides poor results on other versions is more likely to encode only the "wrong behavior" related to that specific fault.

Since one of our interests is in using the same models across versions, we also studied whether there were predictors that worked consistently well across all versions. We were able to find a common set of 11 excellent predictors for all of the programs versions that we studied. Classification using these predictors resulted in error rates below 7% for all versions of the data. Moreover, the models that achieved these results never included more than 5 of those 11 predictors.

Another threat to the generality of our results is that we considered only versions with a single fault (like most of the existing literature). Therefore, we performed a preliminary study in which we used our technique on the versions of JABA containing multiple faults (see Table I). In the study, we selected predictors that worked well for single-error versions and used them for versions with multiple errors. We found that, although there are instances in which these predictors did not perform well and produced error rates around 18%, the predictors worked well most of the time, with error rates below 2%. In Sections V and VI, we further discuss the use of our approach in the presence of multiple errors. In this first set of studies, our focus is mostly on investigating and defining techniques that can successfully classify program executions, rather than techniques specifically designed to recognize failures under different conditions.

Multiplicity Issues: When the number of predictors is much larger than the number of data points (test cases, in our case), it is possible to find good predictors purely by chance. When this phenomenon happens, predictors that work well on training data do not have a real relationship to the outcome, and therefore perform poorly on new data. If the predictors are heavily correlated, it becomes even more difficult to decide which predictors are the best and most useful for lightweight instrumentation. Inclusion of too many predictors may also have the effect of obscuring genuinely important relationships between predictors and outcome. This issue, which has been overlooked by many authors in this area, is a fundamental one because multiplicity issues can essentially mislead statistical analysis and classification.

Our first step to deal with multiplicity issues was to reduce the number of potential predictors by considering method counts, the execution data with the lowest number of entities. Furthermore, we conducted a simulation study to understand how having too many predictors may result in some predictors that have strong predictive powers purely by chance. The simulation was conducted as follows. We selected a version of the subject and treated the method counts for that version as a matrix (i.e., we arranged the counts column by column, with each row representing the counts for a particular test case). To create a randomly sampled data set, we then fixed the row totals (counts associated with each test case), and randomly permuted the values within each row. In other words, we shuffled the counts among methods, so as to obtain a set of counts that does not relate to any actual execution.

We repeated this process for each row in the data set to produce a new randomly sampled data set. With the data set so created, we randomly sampled 10% subsets of the column predictors. Our simulations of 100 iterations each showed that a random 10% draw *never* (for all practical purposes) produced a set of predictors able to classify executions as "pass"/"fail" in a satisfactory manner. Therefore, the probability of obtaining, purely by chance, good predictors from a 1% random subset of the predictors 80% of the time (which is what we observed in our study on the real data) is very slim. We can thus conclude that the results we obtained are due to an actual relation between the occurrence of a failure and the value of the method counts.

C. Summary Discussion

So far, our studies suggest that, for the subject and executions considered, we can (1) reliably classify program outcomes using execution data, (2) do it using different kinds of execution data, and (3) at least in principle, do it while ignoring a substantial percentage of potential data items. According to these results, we can use our technique based on random forests and on the collection of method counts to classify remote executions. This technique would be useful as long as we can build our classification models through in-house training under the developer supervision.

V. CLASSIFICATION USING LIGHTWEIGHT, IN-THE-FIELD DATA COLLECTION

In contrast to the previous technique, which used data collected in-house, our goal here is to define a technique that can classify executions of deployed programs using models built from data collected in the field. As illustrated in the middle panel of Figure II-A, the models would be built by collecting execution data in the field while using a lightweight, built-in oracle to label these executions. The oracle could be based on various mechanisms, such as assertion checking, monitoring of error handling code, or crash detection. This scenario refers to situations where we want to train the models in the field (e.g., because the number of configurations/parameters is large, and we want to focus on the ones actually used by the users) and more accurate oracles are too expensive or impossible (e.g., in the case of human oracles) to attach to a deployed program. In these cases, a classification technique needs to operate on

readily-collectible execution data collected in the field, use these data to train the models, and later classify executions according to the models. The initial data collection would typically involve a subset of software instances (e.g., instances used by beta testers), whereas the later classification could be performed on execution data coming from any instance and would not require the use of a built-in oracle.

As a specific instance of this general problem, we again consider classification of remote executions as passing or failing. The machine-learning technique that we use to define our approach is, in this case, a learning technique that we have invented called *association trees*. In terms of execution data, we consider only method counts, which proved effective when used with our first technique (see Section IV).

In the following sections, we first describe our approach based on association trees. Then, we discuss the empirical studies that we performed to assess and refine the approach.

A. Association Trees

Our first approach, based on random forests, created very accurate models for the system and test cases studied. It requires, however, that every program instance capture the same, large set of features. When the training-data collection is performed completely in house, and on sufficiently powerful computers, this will not be a problem. In other situations, however, the data collection and transmission overhead could be unacceptable. Consider the remote failure modeling scenario of Section II-B, for example, in which developers may wish to capture low-level execution data from fielded instances, hoping that the resulting classification models will allow for precisely locating potential failure causes. Capturing lots of low-level data increases overhead, which will eventually affect system performance in an unacceptable way.

In the random forest studies, we found that from a large (thousands) pool of potential predictors only a very small fraction (less than 1%) were important for classification. Unfortunately, the approach based on random forests must actually collect the data and conduct the analysis before it can determine which predictors are actually useful. These results suggest that an alternative approach—assuming one can be found—could eliminate a substantial amount of the data collection overhead, be it measured in data transmission bandwidth, runtime overhead, or code bloat. Such an approach would also save considerable data-analysis costs, which tend to grow polynomially with the number of potential predictors. All of these costs are substantial, especially when training data is captured in the field.

To address this problem, we first changed our instrumentation strategy. Instead of capturing each potential predictor in each execution, we capture only a small subset (less than 10%) of predictors in each instance. This sampling drastically reduces the data collection overhead, but creates a dataset in which nearly all of the entries are missing, which greatly reduces the performance of tree-based techniques; traditional tree-based classifiers, like random forests, do not work well when many predictors are missing. For example, one tree-based algorithm applied to this reduced data for one JABA version produced an error rate of 8.5%, compared to an error rate below 1% when applied to the complete data sets. To solve this problem, we developed a new classification technique, called *association trees*, that works well even when different instances capture different predictors.

The training set for the association trees algorithm is, as usual, a set of labeled execution data. Each data vector has one slot for each potential predictor. If a predictor is collected during a given run (i.e., the corresponding program entity is instrumented), its value is recorded in the vector. Otherwise, a special value (*NA*) is recorded, indicating that the predictor was not collected for that run. Also as usual, the algorithm's output is a model that predicts the outcome of a run based on the contents of an execution data vector. However, unlike for our tree-based technique, in this approach the models will predict one of {"pass", "fail", "unknown"}, where "unknown" means that the model is unable to make a prediction (because of the lack of data).

The association trees algorithm has three stages: (1) transform the predictors into items that are present or absent, (2) find association rules from these items, and (3) construct a classification model based on these association rules.

In the first stage, the algorithm transforms the predictors into items that are considered either present or absent, in two steps. First, it screens each predictor and checks that the Spearman rank correlation between the predictor and the observed outcomes exceeds a minimum threshold. (Spearman's rank correlation is similar to the traditional Pearson's correlation, except that the actual value of each variable is replaced with a rank: 1 for the largest value, 2 for the second largest value, and so on, which makes the correlation measure less sensitive to extreme values.) The goal of this step is to discard predictors whose values are not correlated with outcomes and, thus, are unlikely to be relevant for the classification.

Second, the algorithm splits the distribution of each remaining predictor into two parts, such that the split point is the point that minimizes the p-value of the t-test for outcome. Ideally, after the split, all runs with one outcome would have values above the split, while all runs with the other outcome would have values below it. If the p-value is below a maximum threshold of .0005, the algorithm creates two items: the first item is present in a given run if the predictor's value is below the split point; the second item is present if the value is above the split point. Neither item is present if the corresponding predictor was not being measured during the run. We also represent outcomes as separate items ("pass" and "fail" for the data used in this paper).

After the algorithm completes Stage 1, the original set of training data has been transformed into a set of observations, one per run, where each observation is the set of items considered present for that run. The goal of the second stage of the algorithm is to determine which groups of frequently occurring items are strongly associated with each outcome. To this end, it applies the well-known apriori data-mining algorithm [21], where it sets outcome as the item to predict. The apriori algorithm is used extensively by the data-mining community to efficiently find items that frequently occur together in a data set (e.g., to discover which items, such

as peanut butter and jelly, are frequently purchased together). The algorithm can then determine which of these frequently occurring sets of items are good predictors of the presence of another item of interest (e.g., if someone buys peanut butter and jelly, then they often buy bread as well). In our case, we want to know which sets of items are correlated with successful executions or failures. These sets of items, together with their correlations with outcomes are called *association rules*. Association rules are of the form A implies B. We call A the *antecedent*, and B the *consequent* of the rule. Each rule needs a minimum *support*: the antecedent must appear in a certain fraction of all observations for the rule to be considered potentially valid. Each rule also needs a minimum *confidence*: when the antecedent is present in an observation, the consequent must also be present in the observation for a predefined fraction of the observations (the value of this fraction represents the confidence). Typically, we set the confidence level to 1 if we assume outcomes are deterministic. The confidence level can be decreased if we wish to consider non-deterministic outcomes as well. As explained in Section V-B, in our empirical evaluation of the approach, we vary the required supports to experiment with different settings. However, because failures tend to occur far less often than successes, we typically set the support for rules predicting passing executions to be several times greater than the support for predicting failing ones. Finally, to reduce computation times, we choose to limit the length of association rules to three. Therefore, if four items must be present to perfectly predict an outcome, our algorithm will not be able to find the corresponding rule (in these studies).

The third and last stage of the algorithm performs outcome prediction using the association rules produced in the previous stage. Given a new run, the algorithm finds the rules that apply to it. If all applicable rules give the same outcome, it returns that outcome as the prediction. If there is disagreement, or there are no applicable rules, the algorithm returns a prediction of "unknown." We chose this unanimous voting scheme to be conservative. One might also decide to use majority voting or a weighted voting scheme (if the penalties for incorrectly predicting different outcomes are uneven).

B. Empirical Evaluation

1) Set-up: In this study, we use the data discussed in Section III. The instrumentation measured the 1,240 method-entry counts greater than zero as possible features. From this complete data set, we created simulated program instances that represented a hypothetical situation in which each instance is instrumented to collect measures for 100 features only. To do this, for each simulated instance, we randomly selected 100 features; the remaining ones correspond to features whose instrumentation was not activated and, thus, whose measures were not collected for that instance. We then applied the association tree algorithm to this data under various parameter settings. Note that the goal of these initial tests is simply to determine whether some points in the parameter space yield good classifications models. We leave a more exhaustive analysis of parameter effects and tradeoffs to later investigations. We list and discuss the parameters considered.

- *Test suite size*: we assigned a random sample of b test cases to each simulated instance. Therefore, each instance executed b test cases, producing 100 unique predictors for each test case run. For each program version, we executed 18,000 total test runs, with test suite sizes of b = 6, 12, or 24 across 3,000, 1,500, and 750 simulated instances, respectively. One half of the data was used as a training set. The rest was used for cross validation.
- *Support and confidence*: we used a minimum support of 1 for both failures and successes because we consider the outcomes to be deterministic. We set minimum confidences of .0066, .0033, and .0016 for rules predicting success, and .001 and .0005 for rules predicting failure.
- *Correlation thresholds*: In Stage 1 of the algorithm, we discarded predictors from consideration if they did not have a minimum Spearman rank correlation with outcomes of at least 0.4, 0.3, or 0.2.

Performance measures: For every run of the algorithm, we captured several performance measures:

- *Coverage*: Percentage of runs for which the model predicts either "pass" or "fail" (i.e., 1-percentage of "unknown").
- *Overall misclassification*: Percentage of runs whose outcome was incorrectly predicted.
- *False positives*: Percentage of runs predicted to be "pass" that instead failed.
- *False negatives*: Percentage of runs predicted to be "fail" that instead passed.

2) Results: We constructed about 90 different association tree models across the single- and multiple-fault versions of JABA used in the previous study. Figure 4 depicts the various performance metrics. For each metric, the figure shows the aggregated results for all versions (All Data), the results for single-fault versions only (1-Flt), and the results for multiple-fault versions only (N-Flt). Across all versions and settings, we found coverages ranging from 2% to 95% with a median of 63%. Coverage was substantially higher for single-fault versions (median of 74%) than for multiple-fault versions (median of 54%).

Overall misclassification ranged from 0% to 10%, with a median of 2%. There was little difference between single- and multiple-fault versions.

False positives had a median of 0% and a 75th percentile of 3%. Due to a few outliers, however, the distribution ranged from 0% to 100%. We found that all 7 of these outliers occurred on multiple-fault versions, with correlation thresholds of .3 or .4, and when coverage was less than 20%. Also, in each case, the accompanying false negative percentage was always 0%. In these cases, the high correlation threshold caused many predictors to be discarded. Few rules for passing runs were generated and, thus, all of the passing runs in the test data were predicted to be "unknown" or "fail."

False negatives, like false positives, were generally low. The median false negative percentage was 10%. The 75th percentile was also low, 22%. Again, there were some (3) outliers with 100% false negative percentage and 0% false

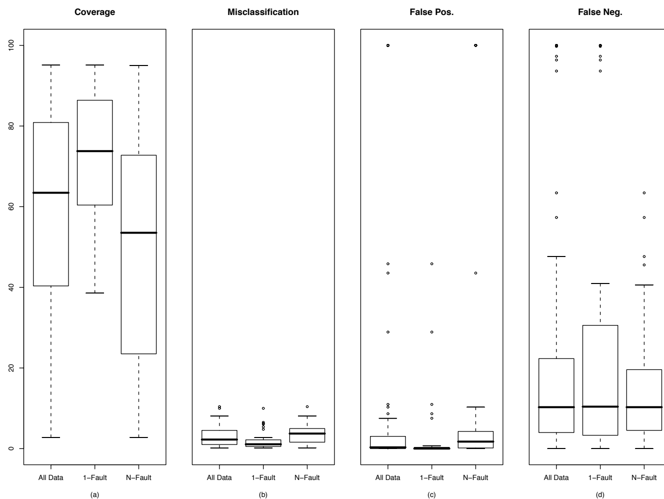

Fig. 4. Association tree results.

positive percentages. This time however, the outliers were all for runs of version 11, a single-fault version, and had 70% or more coverage.

Because one of the goals of this evaluation is to refine and tune our approach, we have also analyzed and made some tentative observations concerning the parameters used in building association trees.

Test suite size appeared uninfluential. We found results of various quality with every size. Given that we restricted the length of association rules to three and that we observed between 750 and 3000 instances, with each instance having roughly 1/12th of the possible predictors in it, we should expect to have good coverage of all potential items by chance.

Increasing support for passing runs strongly affected coverage, but much less effect for the failing runs. This situation occurs because, with a lower support for passing runs, we find more rules for predicting "pass" (the much more frequently appearing class), and can thus predict correctly in more cases. The main effect of increasing support for failing runs was to decrease false negatives and to increase false positives.

The coarsest tuning parameter was the minimum correlation needed between the predictors and the outcomes. When set too low (0.1), far too many rules were found, which can greatly slow down the algorithm and even make it run out of memory while trying to find rules. Conversely, when the minimum correlation was set too high, too few rules were found. In our experiment, settings of 0.2 or 0.3 tended to give good results, while settings of 0.1 or 0.4 tended to perform much worse.

C. Summary Discussion

Although still preliminary, our results suggest that our association-tree technique performed almost as well as our tree-based technique defined in Section IV, albeit with a few outliers. In terms of data collection costs per instance, however, this new approach is considerably less expensive. Our random-forests technique, like all classification techniques that we know, measures all possible predictors across all instances. (Even techniques that do sampling, such as the one in References [6] and [7], still collect data for each predictor, therefore missing opportunities for reducing instrumentation and data-transfer costs.) Conversely, our association trees approach instrumented only about 8% of the potential predictors in any given instance. Therefore, this second technique is likely to be applicable in cases where lightweight instrumentation is preferable (or necessary), and where a slightly less accurate classification is acceptable.

This new approach, although successful, has some issues that may limit its applicability in some contexts.

First, the technique requires all (sampled) data to be collected before modeling begins, but gives no help in determining how many instances and test cases are needed to build adequate models. In our studies, we arbitrarily chose to use a number of instances and executions (test cases) that goes from 3000 instances running 6 test cases each to 750 instances running 24 test cases each. We currently lack theoretical or heuristic support for deciding how much data to collect.

Second, the technique gives no help in selecting the modeling parameters. Thus, developers must rely on trial and error, which can be problematic in some situations. For example, in some cases we created good models with a correlation threshold of 0.1; in other cases, we ran out of memory on a machine with 1 GB of RAM using the same parameter value.

Third, the technique may under-sample useful predictors. This issue is related to the first problem, in that the algorithm does not help in determining the minimum number of runs needed to build good models. If there are very few useful predictors and not enough runs, then it is possible to miss important predictors (or combinations thereof).

Finally, the technique does not adapt easily to changes in the observed systems and in their usage; it must be rerun from scratch when changes occur. It would be useful and much more cost-effective if the models could be adapted incrementally.

VI. Classification Using Adaptive Sampling

As in the previous section, our goal here is to define a technique that can classify executions of deployed programs using models built from data collected in the field. The key difference is that here we also want our data collection strategy to be adaptive: information gleaned from early runs should help determine which data to collect during later runs. This approach can be used to reduce the process' time to completion and also reduce the total amount of data collected. All other aspects of the overall technique, such as oracle characteristics, classification tasks, type of data collected, and underlying assumptions, are the same as for the previous technique.

A. Adaptive Sampling

One limitation of the association tree technique described in Section V is that the algorithm treats all potential predictors as equally important all the time. That is, the algorithm selects which predictors to collect in a given instance by taking uniform random samples of all possible predictors. Because of this limitation, the association tree algorithm may take a considerably long time to complete and may fail to capture

important association rules in some cases. This may happen, for instance, if only a small percentage of the predictors are actually useful for building good classification models. It may also happen in the case of problems that manifest themselves only after the program has been running for some time (see the scenario about performance modeling of fielded executions discussed in Section II-B.3).

Suppose, for instance, that a given system fails only when method x and method y are each executed more than 64 times in the same run. Suppose further that there are many possible predictors, a small percentage of them are enabled on each instance, and there are relatively few failing runs. In this case, it is possible that methods x and y are rarely sampled together in the same run. Therefore, the failure cause will not be identified or will at least take a long time to be identified. Basically, by giving all predictors the same likelihood of being collected, the algorithm can spread its resources too thinly, leading to poor models.

In these situations, finding ways to rule out useless predictors early could greatly improve the algorithm's costs effectiveness. To tackle this problem, we created an incremental association tree algorithm called *adaptive sampling association trees*. This algorithm incrementally learns which predictors have demonstrated predictive power in the past and preferentially instruments them in future instances, while deemphasizing the other predictors. Important expected benefits of such an approach are that it should allow for (1) reducing the amount of data collection required, (2) eliminating the need to guess at many of the parameter settings, and (3) naturally adapting models over time (instead of requiring complete recalibration every time the system, its environment, or its usage patterns change).

To do adaptive sampling, our algorithm first associates a weight with each predictor. Initial weights can be set in many different ways. For example, they can be based on the developers' knowledge of interesting (i.e., problematic) modules or paths. In the experiments described below, we simply used a uniform weighting of 1 for each predictor.

When a new instance is ready to run, the algorithm queries a central server for the k predictors to be collected in the instance. The server selects without replacement the k features to be measured from the set of possible features. Unlike the basic association trees algorithm, which gives equal weights to all predictors, the adaptive sampling association trees algorithm sets the selection probability of each predictor to be the predictor's weight divided by the total weight of all predictors. Next, the measurement of the selected features is enabled in the instance so that, when it is executed, the resulting execution data are returned to a central collection site.

At this point, the algorithm tests whether the collected predictors are related to the outcome. For each predictor measured, the algorithm computes the Spearman rank correlation between the values observed for this predictor and the outcomes of all runs in which the predictor was collected. If the Spearman's rank correlation is above a minimum threshold, the algorithm increases the predictor's weight by one. If, over the universe of all possible inputs, the Spearman rank correlation of any useful predictor is above our threshold, it is easy to show that, asymptotically in the number of executions, the algorithm will sample it at a high rate. Similarly, if any predictor has a Spearman rank correlation below the threshold, it will eventually drop out of the set of predictors sampled.

Once the algorithm has collected data from a sufficient number of instances, it creates association rules following the same approach as the basic association trees algorithm. The key difference is that, by using a non-uniform sampling strategy, the algorithm is more likely to have heavily sampled useful predictors, while lightly sampling less useful ones. Consequently, with this new algorithm, we expect to find more accurate rules and fewer incorrect ones at any support level.

B. *Empirical Evaluation*

1) Set-up: We used the same data in this study that we used in the previous studies: method counts as execution features measured, and "pass" and "fail" as the possible behaviors. We then simulated 500 instances of each of the single- and multi-fault JABA versions (see Table I) as follows. For each instance, we randomly selected 100 features (i.e., method counts) to collect and k runs (i.e., test case executions), where k is a random number following a Poisson(12) distribution. Half of the runs were allocated to the training set, and the rest were used for cross validation. As discussed above, we assigned to all features an initial weight of 1. After each run, the algorithm increased by 1 the weights of features that were strongly correlated with the outcome (i.e., Spearman rank correlation above 0.3 in absolute value).

Overall, our training and our test sets comprised an average of 3,000 test cases, each collecting 100 predictors. Basic association trees, in contrast, executed 9,000 test cases with 100 predictors each. (Note that we actually considered 18,000 test runs for the basic association trees, but used only half of the data as a training set, as discussed in Section V-B). We measured the performance of each adaptive sampling association tree using the same four measures used in the previous study (see Section V-B).

2) Results: Using the adaptive sampling algorithm, we constructed models (i.e., association rules) for each of the JABA versions used in earlier studies. In the few cases where a given model performed poorly (i.e., produced a false positives rate over 20% or coverage below 80%), we allowed ourselves to change support levels for failure rules and/or for success rules. This strategy mimics the tuning process a developer could perform in practice. There was some variability in the number of iterations needed for the results to converge. This variability is present because the time to discover relevant features varies—if it takes a while to discover good features, then, obviously, more iterations of the process are needed before obtaining good results.

Figure 5 depicts the various performance metrics for the final models (the metrics shown and the notation are the same used in Figure 4). Across all models, coverage ranged from 67.3% to 99.6%. The median coverage was 88.3%. As the figure shows, coverage was substantially higher for single-fault versions (median of 91.25% for 1-Flt) than for multiple-fault versions (median of 79.86% for N-Flt).

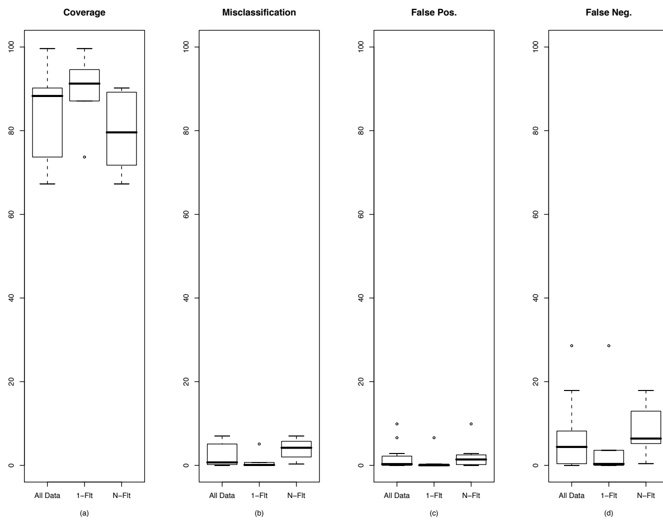

Fig. 5. Adaptive sampling association tree results.

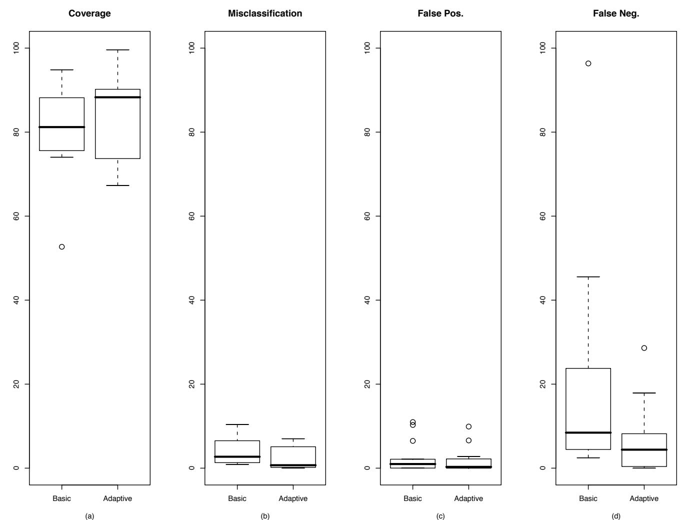

Fig. 6. Basic association trees vs. adaptive sampling association trees.

TABLE II
ADAPTIVE SAMPLING ASSOCIATION TREES USING STATEMENT COUNTS.

Version	Coverage	False.pos	False.neg	Misclass
v4	0.87	0.02	0.01	0.01
v5	0.97	0.00	0.00	0.00
v7	0.98	0.00	0.00	0.00
v9	0.89	0.00	0.00	0.00
v11	0.79	0.00	0.00	0.00
v17	0.98	0.00	0.00	0.00

Overall misclassification ranged from 0% to 7%, with a median of 0.7%. Misclassification was lower for single-fault versions than for multiple-fault versions (0.09% median versus 4.2% median), although both are quite low in practical terms.

False positive percentage ranged from 0% to about 10% due to a few outliers. The 3rd quartile, for instance, is 2.2% false positives. Also in this case, single-fault versions had fewer false positives than multiple-fault versions (0.02% median versus 1.4% median), but both were low in practical terms.

Finally, false negative percentages ranged from 0% to 28.6%. The median was 4%, and the 3rd quartile was 8.2%. Single-fault versions had fewer false negatives than multiple-fault versions (0.33% median versus 6.4% median).

To examine the scalability of this approach, we also applied it to the much more voluminous statement count data (about 12,000 possible features, as opposed to about 1,240 possible features in the case of non-zero method counts). We applied the technique to the 6 single-fault JABA versions using less than 450 instances and collecting less than 600 predictors (less than 5% of the total) per run. The results are shown in Table II. As the table shows, coverage is over 90% on average, and false positives, false negatives, and overall misclassification are close to zero in every case. These results suggest that adaptive sampling may scale nicely and, thus, allow for classification of the larger programs and much larger data sets that we would expect to see in practice.

C. Summary Discussion

Our results suggest that the adaptive sampling association trees approach can perform almost as well as the random forest approach used in Section IV, and as well or better than the basic association tree approach.

Figure 6 depicts the results obtained for the basic association trees side by side with those obtained for the adaptive sampling association trees. To perform this comparison as fairly as possible, we selected the “best” basic association trees results for each version. This is necessary because our basic association trees study included models built with non-optimal parameter settings. Note, however, that since we have four different performance measures, the definition of best is necessarily subjective. In this case, we have preferred models with higher coverage and lower false negatives because software developers are often more interested in identifying failing runs than passing runs. As shown in the figure, by sampling adaptively we generally achieved higher coverage and lower misclassification rates, with significantly less variability between versions.

In addition, in terms of data collection costs, our new approach is significantly cheaper than the previous two approaches. Random forests instrumented all possible 1,240 predictors over 707 test case runs. Association trees instrumented only 100 predictors, but used 9,000 runs. This approach, instead, instrumented only 100 predictors over 3,000 runs. Moreover, since the adaptive sampling approach often converged well before processing the execution data from all scheduled runs, this result is an upper bound on the number of runs needed by the algorithm. One important implication of these savings is that we were able to generate all desired models, whereas the basic association trees algorithm occasionally ran out of memory. Finally, adaptive sampling allowed us to successfully scale up to more voluminous, lower-level data (statement counts instead of method counts).

Although quite effective in our tests, adaptive sampling has some limitations as well. The key limitation we found is that the approach necessarily introduces a sequential de-

pendence among instrumented instances. That is, in its simplest implementation, adaptive sampling would not select the predictors for instance i and deploy it until (1) instance i-1 had returned its data and (2) predictor weights had been are updated (we call this dependence *lag-1 dependence*). There are many workarounds for this issue. For example, we could instrument instances in batches or loosen the dependences by considering lag-k rather than lag-1 approaches. Nevertheless, some sequential ordering would necessarily remain (or the technique would simply degenerate to basic association trees).

Such dependences might make adaptive sampling unacceptable in certain situations, such as cases where many instances are deployed at the same time and it is impossible or undesirable to update them in the field, and cases where observation periods are relatively short, but many runs must be instrumented. In the first case, adaptive sampling is impossible. In the second case, it might result in unacceptable slow-downs of the deployed instances. In both cases, thus, developers might prefer using the uniform sampling offered by basic association trees.

VII. Related Work

Several software engineering researchers have studied techniques for modeling and predicting program behavior. Researchers in other fields have also studied one or more of the general techniques underlying this problem. In this section we discuss some recent work in these areas.

Classifying Program Executions: Podgurski and colleagues [4], [12], [22]–[24] present a set of techniques for clustering program executions. They show that the resulting clusters help to partition execution profiles stemming from different failures. This work differs from ours in that their model construction requires the collection of all predictors for all executions and assumes that a program's failure status is either obvious or externally provided.

Bowring, Rehg, and Harrold [2] classify program executions using a technique based on Markov models. Their models consider only one specific feature of program executions: program branches. Our work considers a large set of features and assesses their usefulness in predicting program behaviors. Also, the models used by Bowring and colleagues need complete branch-profiling information, whereas our approaches can generally perform well with only minimal information.

Execution Profiling: Our work draws heavily on methods and techniques for run-time performance measurement, especially for what concerns program instrumentation issues. One way to instrument a program is to place probes (code snippets) at all relevant program points (e.g., basic block entries). As the programs runs, the probes that gets executed compute and record data about the ongoing execution. Obviously, this simple approach can generate enormous amounts of data and can substantially perturb the very performance developers are trying to observe. Consequently, researchers have explored various techniques to lower instrumentation overhead.

Arnold and Ryder [25] present an approach for reducing instrumentation costs through sampling. The approach is based on (1) having two versions of the code, one instrumented and one non-instrumented, and (2) switching between these two versions according to a sample frequency that is dynamically modifiable. They investigate the tradeoff between overhead and accuracy of the approach and show that it is possible to collect accurate profiling information with low overhead.

Traub and colleagues [26] present an approach based on using few probes and collecting data infrequently. In this approach, called *ephemeral instrumentation*, probes and groups of probes that work together are statically inserted into the host program. The probes cycle between enabled and disabled states. The approach can accurately estimate branch biases while adding only 1%–5% overhead to the program.

Miller and colleagues [27] have developed a dynamic runtime instrumentation system called ParaDyn and an associated API called DynInst. This system allows developers to change the location of probes and their functionality at run-time. This general mechanism can be used to implement many instrumentation strategies, such as ephemeral instrumentation.

Anderson and colleagues [28] present an approach to data collection alternative to instrumentation. While the program is running, they randomly interrupt it and capture the value of the program counter (or other available hardware registers). Using this information they estimate the percentage of times each instruction is executed. They claim that their technique can generate a reasonably accurate model with overheads of less than 5%. A chief disadvantage of this approach is that it is very limited in the kind of data it can gather.

There are numerous other approaches to instrumentation; among the most prominent are techniques such as ATOM [29], EEL [30] ETCH [31], and Mahler [32].

Machine Learning and Statistical Analysis: Our work is also closely related to machine learning techniques that distinguish program outcomes, such as passing runs from failing runs. In this context, one particularly important issue is dealing with data sets in which failures are rare. For example, if a system's failure rate is 0.1%, then a data set of 1M runs would be expected to contain only 100 failing runs. In these situations, it can be difficult to classify the rare outcome. Several general strategies have been proposed to deal with this problem. One approach, called boosting, involves multiphase classification techniques that place special emphasis on classifying the rare outcome accurately. For example, Joshi and colleagues [33] developed a two-phase approach in which they first learn a good set of overall classification rules, and then take a second pass through the data to learn rules that reduce misclassifications of the rare outcome. Similarly, Fan and colleagues [34] developed a family of multiphase classification techniques, called AdaCost and AdaBoost, in which each instance to be classified has its own misclassification penalty. Over time, the penalties for wrongly-classified training instances are increased, while those of correctly-predicted instances are decreased.

a) Partial Data Collection: Our research is also related to approaches that aim to infer properties of executions by collecting partial data, either through sampling or by collecting data at different granularities.

Liblit and colleagues [6], [7] use statistical techniques to sample execution data collected from real users and use the

collected data to perform fault localization. Their approach is related to our work, but is mostly targeted to supporting debugging and has, thus, a narrower focus. Furthermore, although their data collection approach is time efficient because it aggressively samples execution data, it may still introduce space-related issues: it must add instrumentation to measure all predictors (predicates, in their case) in all instances, which may lead to code bloat; it must also collect one item per predictor for each execution, which may result in a considerable amount of data being collected from each deployed instance. Our techniques based on association trees do not have these problems because they can build reliable models collecting only a small fraction of execution data from each instance.

Pavlopoulou and Young [11] developed an approach, called residual testing, for collecting program coverage information in program instances deployed in the field. This approach restricts the placement of instrumentation in fielded instances to locations not covered during in-house testing. This approach is related to ours because it also partially instruments deployed software. However, it has a very different focus and uses completely different techniques. In particular, it does not perform any kind of classification of program outcomes.

Elbaum and Diep [3] perform an extensive analysis of different profiling techniques for deployed software, including the one from Pavlopoulou and Young discussed above. To this end, they collect a number of execution data from deployed instances of a subject program and assess how such data can help quality-assurance activities. In particular, they investigate how the granularity and completeness of the data affect the effectiveness of the techniques that rely on such data. This work provides information that we could leverage within our work (e.g., the cost of collecting some kinds of data and the usefulness of such data for specific tasks) and, at the same time, could benefit from the results of our work (e.g., by using predicted outcomes to trigger data collection).

VIII. Conclusion and Future Work

In this paper, we have presented and studied three techniques—random forests, basic association trees, and adaptive sampling association trees—whose goal is to automatically classify program execution data as coming from program executions with specific outcomes. These techniques can support various analyses of deployed software that would be otherwise impractical or impossible. For instance, they can allow developers to gather detailed information about a wide variety of meaningful program behaviors. They can also allow programs to trigger targeted measurements only when specific failures are likely to occur.

The empirical evaluation and investigation of our techniques suffers, like all empirical studies, from various threats to validity. The main threats are that we studied only a single system, considered only two possible outcomes, exercised the system with a limited set of test cases, and focused only on versions with failure rates higher than 8%. Nevertheless, the system is sufficiently large and complex to have non-obvious behaviors, the faults are real, and the test suite includes tests (obtained from users) that represent real usages of the system. Therefore, we can use the results of our evaluation to draw some initial conclusions about the effectiveness and applicability of our techniques.

In our initial studies, we examined three fundamental questions about classification techniques. Our results showed that we could reliably classify binary program outcomes using various kinds of execution data. We were able to build accurate models for systems with single or multiple faults. We also found that models based on method counts were as accurate as those built from finer-grained data. Finally, we conducted several further analyses that suggested that (1) our results are unlikely to have occurred by chance and (2) many predictors were correlated (i.e., many predictors were irrelevant). An important implication of this latter finding is that the signal for failure may be spread through the program, rather than associated with a single predictor or small set thereof.

In general, all three techniques performed well with overall misclassification rates typically below 2%. The key differences between the techniques lie in (1) how much data must be collected and (2) whether the training phase is conducted in a batch or sequential fashion. These differences make each technique more or less applicable in different scenarios and contexts. For example, the random-forests technique requires for the training to be performed in-house, by using an accurate oracle and capturing all predictors in every program instance. The basic association trees algorithm randomly and uniformly instruments a small percentage of all potential predictors (8% in our study), but still allows reliable predictive models to be built. Our third technique, called adaptive-sampling association trees, was also almost as accurate as random forests, while improving on the basic association trees algorithm and being much cheaper than both.

In future work, we will continue our investigations in several directions. First, we will study our techniques in increasingly more realistic situations and on increasingly larger systems. Specifically, we will apply these techniques to modeling faults in several large, open source systems. We will also extend our investigation beyond binary classifications to assess whether our techniques can capture behaviors other than passing and failing (e.g., performance).

Second, we will investigate new weighting schemes for use with adaptive sampling association trees. We will explore weighting schemes that also take into account the runtime overhead imposed by the measurement and collection of each predictor. One possibility, for instance, is to reward good predictors that are on lightly executed paths more than good predictors on heavily executed paths.

Finally, we intend to explore the analysis of multiple, simultaneous data streams, as opposed to a single data stream as we do now. In particular, we want to collect distinguished data streams from multiple components in component-based or service-oriented systems, with the goal of not only detecting problems, but also associating them to specific components.

Acknowledgment

This work was supported in part by National Science Foundation awards CCF-0205118 to the National Institute of

Statistical Sciences (NISS), CCR-0098158 and CCR-0205265 to University of Maryland, and CCR-0205422, CCR-0306372, and CCF-0541080 to Georgia Tech. We used the `R` statistical computing software and the `randomForest` library, available at `http://cran.r-project.org/` to perform all statistical analyses. Jim Jones prepared and provided the 19 single-fault program versions. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, November 2002, pp. 2–8.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, July 2004, pp. 195–205.

[3] S. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[4] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-based methods for classifying software failures," in *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, November 2004, pp. 451–462.

[5] D. M. Hilbert and D. F. Redmiles, "Extracting usability information from user interface events," *ACM Computing Surveys*, vol. 32, no. 4, pp. 384–421, December 2000.

[6] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2003)*, June 2003, pp. 141–154.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[8] "Microsoft online crash analysis," 2007, http://oca.microsoft.com.

[9] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proceedings of the 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, September 2003, pp. 128–137.

[10] A. Orso and B. Kennedy, "Selective Capture and Replay of Program Executions," in *Proceeding of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, May 2005.

[11] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," in *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, May 1999, pp. 277–284.

[12] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, May 2003, pp. 465–474.

[13] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, 2004, pp. 45–54.

[14] A. Orso, J. A. Jones, and M. J. Harrold, "Visualization of program-execution data for deployed software," in *Proceedings of the ACM symposium on Software Visualization (SOFTVIS 2003)*, June 2003, pp. 67–76.

[15] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying classification techniques to remotely-collected program execution data," in *Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, September 2005, pp. 146–155.

[16] R. Duda, P. Hart, and D. Stork, *Pattern Classification*, Wiley, 2000.

[17] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, Springer, 2001.

[18] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*, Cambridge University Press, 2004.

[19] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[20] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[21] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th Conference on Very Large Databases*, 1994, pp. 487–499.

[22] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing failure: the distribution of program failures in a profile space," in *Proceedings of the 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2001)*, September 2001, pp. 246–255.

[23] W. Dickinson, D. Leon, and A. Podgursky, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, May 2001, pp. 339–348.

[24] D. Leon, A. Podgurski, and L. J. White, "Multivariate visualization in observation-based testing," in *Proceedings of the 22nd international conference on Software engineering (ICSE 2000)*, May 2000, pp. 116–125.

[25] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, 2001, pp. 168–179.

[26] O. Traub, S. Schechter, and M. Smith, "Ephemeral instrumentation for lightweight program profiling," June 2000, unpublished technical report, Harvard University.

[27] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tools," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, November 1995.

[28] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, November 1997.

[29] A. Srivastava and D. Wall, "Link-time optimization of address calculation on a 64-bit architecture," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1994)*, 1994.

[30] J. Larus and E. Schnarr, "Eel: Machine-independent executable editing," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1995)*, 1995.

[31] T. Romer, G. Voelker, A. Wolman, S. Wong, H. Levy, B. Chen, and B. Bershad, "Instrumentation and optimization of win32/intel executables using etch," in *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[32] D. Wall and M. Powell, "The mahler experience: Using an intermediate language as the machine description," in *Second International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1987)*, October 1987.

[33] M. V. Joshi, R. C. Agarwal, and V. Kumar, "Mining needle in a haystack: classifying rare classes via two-phase rule induction," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 30, no. 2, pp. 91–102, 2001.

[34] W. Fan, S. J. Stolfo, J. Zhang, and P. K. Chan, "AdaCost: misclassification cost-sensitive boosting," in *Proc. 16th International Conference on Machine Learning*, 1999, pp. 97–105.

Murali Haran graduated with a B.S. in Computer Science (with honors) from Carnegie Mellon University, Pittsburgh, Pennsylvania in 1997. In 2001 and 2003 he earned M.S. and Ph.D. degrees in Statistics from the University of Minnesota, Minneapolis. He was a postdoctoral fellow at the National Institute of Statistical Sciences , Research Triangle Park, North Carolina for 2004-2005. He has been Assistant Professor at the Department of Statistics, Pennsylvania State University, since 2004. His research interests include Markov chain Monte Carlo algorithms and hierarchical models for spatial data. He has had several interdisciplinary collaborations including projects with Plant Pathology, Geography and Meteorology at Penn State. He has also worked with computer scientists on statistical issues in software engineering.

Alan F. Karr is Director of the National Institute of Statistical Sciences (NISS), a position he has held since 2000. He is also Professor of Statistics & Operations Research and Biostatistics at the University of North Carolina at Chapel Hill (since 1993), as well as Associate Director of the Statistical and Applied Mathematical Sciences Institute (SAMSI). Before coming to North Carolina, he was Professor of Mathematical Sciences and Associate Dean of the School of Engineering at Johns Hopkins University. His research activities are cross-disciplinary collaborations involving statistics and such other fields as data confidentiality, data integration, data quality, software engineering, information technology, education statistics, transportation and social networks. He is the author of 3 books and more than 100 scientific papers. Karr is a Fellow of the American Statistical Association and the Institute of Mathematical Statistics, an elected member of the International Statistical Institute, and served as a member of the Army Science Board from 1990 to 1996.

Michael Last Biography text here.

Alessandro Orso received his M.S. degree in Electrical Engineering (1995) and his Ph.D. in Computer Science (1999) from Politecnico di Milano, Italy. He was a visiting researcher in the EECS Department of the University of Illinois at Chicago in 1999. From March 2000, he has been with the College of Computing at the Georgia Institute of Technology, first as a research faculty and then as an assistant professor. His area of research is software engineering, with emphasis on software testing and analysis. His interests include the development of techniques and tools for improving software reliability and trustworthiness, and the validation of such techniques on real systems.

Adam A. Porter earned his B.S. degree summa cum laude in Computer Science from the California State University at Dominguez Hills, Carson, California in 1986. In 1988 and 1991 he earned his M.S. and Ph.D. degrees from the University of California at Irvine. Currently an associate professor, he has been with the department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland since 1991. His current research interests include empirical methods for identifying and eliminating bottlenecks in industrial development processes, experimental evaluation of fundamental software engineering hypotheses, and development of tools that demonstrably improve the software development process.

Ashish Sanil Ashish Sanil obtained a M.S. and a Ph.D. in Statistics from Carnegie Mellon University. He also has a B.Sc.[Hons.] and a M.Sc. in Mathematics from the Indian Institute of Technology at Kharagpur. Ashish is curenly a Quantitative Analyst at Google, Inc. Previously, he worked at the Bristol-Myers Squibb Company as a Senior Research Biostatistician. Prior to that, he was a Research Statistician at the National Institiute of Statistical Sciences. His current research interests include applications of statistical prediction models, e.g. as applied to the software development process. He is also interested in applying Bayesian methodology for devising efficient adaptive experimental designs.

Sandro Fouche` Sandro Fouche is a doctoral candidate in Computer Science at the University of Maryland, College Park. His research interests include empirical validation of large-scale systems, systems for deploying and managing high-reliability computational services, and experimental evaluation of fundamental software engineering hypotheses. Sandro is a member of the ACM and IEEE.