Improving Observation-Based Testing with Database Incorporation Heuristics

Gregory M. Kapfhammer

Abstract—**Observation-based software testing techniques attempt to identify defect-revealing software execution profiles from a large collection of usage scenarios that have an unknown quality. The current observation-based testing research pioneered by Dickinson et al., Leon et al., and Podgurski et al. [4], [13], [16] employs unsupervised machine learning algorithms, such as cluster analysis, to isolate important or meaningful test cases. However, these existing techniques have not been applied to applications that interact with a database. In this paper, we briefly highlight a data collection infrastructure that was developed to effectively capture the execution of Java applications. Also, we evaluate the veracity of the assumptions that underly current observation-based testing methods. Furthermore, we describe some preliminary heuristics that attempt to incorporate an application's interaction with a database into the input space of the certain machine learning algorithms. Finally, we discuss the experiments that were conducted to evaluate the usage of cluster analysis for observation-based testing and the ability of our database incorporation heuristics to improve the operation of these algorithms.**

Index Terms—**observation-based testing, unsupervised machine learning, cluster analysis, database-driven application testing**

I. Introduction

OBSERVATION-BASED testing is a testing technique that uses software execution profiles to identify usage scenarios that are likely to cause a system to fail. After execution profiles have been gathered from the repeated execution of the software system, these profiles are analyzed with a variety of unsupervised machine learning algorithms. Dickinson et al, Leon et al., and Podgurski et al. have used cluster analysis and other multivariate data analysis techniques to select program executions that are likely to cause a software system to fail [4], [13], [16]. To date, none of the empirical analyses of the aforementioned techniques have been conducted on an application that frequently interacts with a database. Since an increasing number of software applications are using proven relational database technology to store information, it is important to develop an observation-based testing approach that is customized for these types of programs. After building a candidate application, developing a data collection infrastructure, and extending existing cluster analysis algorithms, we have conducted several empirical analyses to evaluate the theoretical foundations of observation-based testing. Furthermore, we have developed a number of heuristics that attempt to incorporate the behavior of a stand-alone application when it uses a database. Finally, we have examined the effectiveness of our proposed heuristics in several empirical analyses.

In Section II-A we provide a brief review of software testing terminology and a discussion of program spectra. Furthermore, Section II-B offers an overview of current observation-based software testing techniques. In Section II-C we review the concepts associated with cluster analysis and highlight the different distance metrics and linkage techniques that can be used to produce clusters in an input space. Also, Section III examines the data collection infrastructure that was developed to support the extraction of the input spaces for our unsupervised machine learning algorithms. Next, we explain the experimental framework that was employed during our examination of observation-based testing in Section IV. Specifically, Section IV-A reviews our candidate application and Section IV-B offers an overview of our heuristics for incorporating database interaction information into the cluster analysis process. Section IV-C describes an experiment that attempted to determine if it was possible to use cluster analysis to recognize similar usages of our candidate application. Moreover, Section IV-D discusses an experiment that was conducted to evaluate whether it was possible to effectively isolate anomalous program behavior by using cluster analysis. In Section IV-E we examine the impact that our proposed heuristics have on the quality of the resulting clusters of the input space. Finally, Section V draws conclusions and offers some promising avenues for future research.

Department of Computer Science
Allegheny College, Meadville PA, 16335, USA
Email: gkapfham@allegheny.edu

II. Preliminaries

A. Software Testing Review

Software testing techniques attempt to isolate defects in a given software application. Software testing that is not exhaustive can only reveal the existence of defects and cannot conclusively prove the correctness of a software system [15]. However, the combinatorial explosion of test inputs clearly shows that exhaustive testing is intractable. Figure 1 provides an example of a methodology that could be employed during the testing and analysis of an application that interacts with a database.

In this software testing methodology, a *database seeder* is used to populate the database with acceptable values. Next, a *test case selector* is used to chose the tests that will be most likely to reveal defects and/or produce a confidence in the system under test. In the observation-based testing paradigm, test cases are selected from a large collection of execution profiles of an unknown quality through the usage of cluster analysis algorithms. After test case descriptions have been produced, it is important to use a *test case generator* to create tests that can be executed against the system under test. These test cases can be used by a *test executor* to exercise the chosen application. The results from the test execution phase are normally provided to a *test adequacy evaluator* that attempts to analyze the "quality" of the test cases. Also, the test results can be examined by a *test minimizer* that discovers the "essence" of the test cases in an attempt to improve the isolation and repair of defects. Finally, the current test suite can be used by a *regression tester*

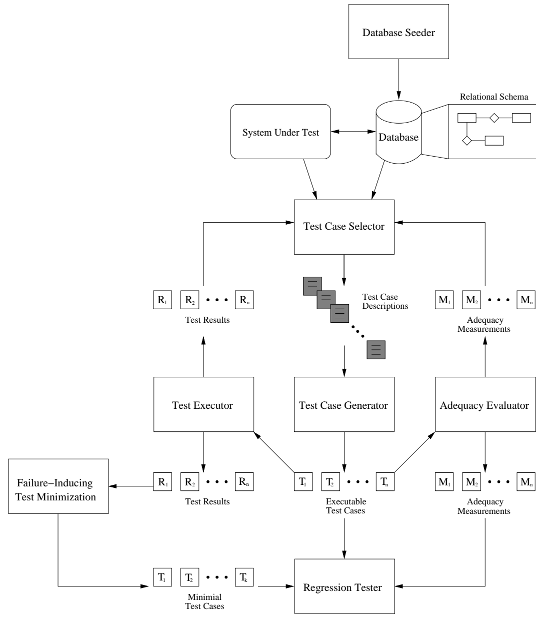

Fig. 1. A Methodology for Testing Database-Driven Applications.

that is responsible for assessing the impact that bug fixes and new functionality have on the quality of the existing application.

B. Observation-based Testing

According to Dickinson et al., observation-based testing is a software testing technique that "permits a useful subset of test cases to be selected from a large set of tests of unknown quality" [4]. Observation-based testing is distinctly different than synthetic-based testing approaches that rely upon the manual or automatic generation of test cases. However, it is important to note that observation-based testing can only be effectively employed when a significant portion of the application under test is available for execution. Current observation-based testing techniques are governed by two assumptions. First, this approach to software testing assumes that it is possible to adequately characterize the execution behavior of a software system through the usage of unsupervised machine learning algorithms like cluster analysis [16]. Second, Podgurski et. al have suggested that the unusual execution profiles revealed by cluster analysis (i.e. those profiles that are located in small or singleton clusters) are often related to application failure [16]. Dickinson et al. describe the application of cluster-based filtering algorithms that attempt to isolate execution profiles that exercise the software under test in a meaningful or unusual fashion [4].

Program spectra, or characterizations of program behavior, can be used for many different software testing and analysis purposes [8]. Different program spectra, such as control flow, data flow, variable values, and event sequences information can be used during observation-based testing analyses [4]. For example, path spectra include all of the execution paths that were recorded during the observation of a program. Current observation-based testing techniques coarsely characterize the behavior of a software system by analyzing the spectra of execution counts for the methods within a given application [4]. It is conceivable that the cluster analysis algorithms employed by observation-based testing could rely upon the program spectra of class usage counts. In [4] and [13], Dickinson et al. and Leon et al. collect program spectra information by using the GNU call coverage profiler, `gprof`. In [18], Steven et al. describe an application called `jRapture` that can effectively capture a number of different program spectra for an arbitrary Java application.

C. Cluster Analysis

Cluster analysis can be classified as a form of unsupervised machine learning. Han et al. note that clustering is technique that attempts to group data into classes where all of the objects in a given cluster are similar [7]. In this report, we will focus on hierarchical, agglomerative cluster analysis methods. These types of cluster analysis algorithms create a hierarchy of clustered objects that eventually belong to a single cluster that contains the complete input space [7]. Since hierarchical cluster analysis is a distance-based statistical analysis technique, it is often profitable to normalize the input space in an attempt to prevent high magnitude attributes from inappropriately influencing the cluster formation process [2], [7]. After the usage of input space normalization, the cluster analysis procedure computes the distance between the points in a data set [17]. Once the distance information has been computed, different linkage mechanisms are used to produce a hierarchical cluster tree known as a dendrogram. This dendrogram graphically depicts the iterative formation of clusters from existing items in the input space or the already created clusters [17]. Clusters are formed when horizontal lines are "grafted" onto the dendrogram at the desired level(s) of distance. Since hierarchical clustering approaches only make a single single pass over the input space, they are fast, but potentially unable to produce meaningful clusterings [7].

Traditionally, cluster analysis algorithms consider the $(m \times n)$ input matrix $\mathbf{X}$ as collection of m, $(1 \times n)$ row vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m}$. Equation 1 expresses the Euclidean distance between vectors $\mathbf{x_r}$ and $\mathbf{x_s}$, while Equation 2 describes the City Block distance between the same vectors. Other similarity metrics, such as Standardized Euclidean distance, Mahalanobis distance, and the Minkowski metric can also be used to compute distance information [2], [9], [11], [17]. The Minkowski metric is a generalization of the Euclidean and City Block distance formulations. Generally, these two distance measurements are most useful when they are applied to interval-scaled variables [7].

$$d^e(\mathbf{x_r}, \mathbf{x_s}) = (\mathbf{x_r} - \mathbf{x_s})(\mathbf{x_s} - \mathbf{x_r})^T \tag{1}$$

$$d^{cb}(\mathbf{x_r}, \mathbf{x_s}) = \sum_{j=1}^{n} \mid x_{rj} - x_{sj} \mid \tag{2}$$

The linkage algorithms employed by cluster analysis use information about the distance between items in a cluster to create new clusters. Essentially, a linkage technique is responsible for

measuring the distance between two specific clusters of objects in order to determine the clusters that will be produced during the current iteration of a hierarchical clustering algorithm [7]. There are several different approaches for computing the distance between two of the current clusters in an input space. Equation 3 through Equation 5 describe the single, complete, and average linkage mechanisms, respectively. For a more detailed review of other linkage techniques, such as centroid and ward linkage, please refer to [2] and [3]. In our description of these linkage mechanisms, we adapt the notation used in [2] and [7]. Thus, we define C_r to be the set of objects in cluster r and C_s to be the set of objects in cluster s. For convenience, we assume that $\mathbf{x_j}$ is a member of the singleton cluster C_j if it is an object that has not been assigned to a cluster. Moreover, we denote $d(\mathbf{x_r}, \mathbf{x_s})$ as the distance between $\mathbf{x_r}$ and $\mathbf{x_s}$ as calculated by an arbitrary distance metric.

$$d_{sng}(C_r, C_s) = min_{\mathbf{x_r} \in C_r, \mathbf{x_s} \in C_s} d(\mathbf{x_r}, \mathbf{x_s}) \quad (3)$$

$$d_{cmp}(C_r, C_s) = max_{\mathbf{x_r} \in C_r, \mathbf{x_s} \in C_s} d(\mathbf{x_r}, \mathbf{x_s}) \quad (4)$$

$$d_{avg}(C_r, C_s) = \frac{1}{(|C_r| \cdot |C_s|)} \sum_{\mathbf{x_r} \in C_r} \sum_{\mathbf{x_s} \in C_s} d(\mathbf{x_r}, \mathbf{x_s}) \quad (5)$$

It is possible for different combinations of distance and linkage measurements to produce clusterings of differing levels of quality. For example, a certain combination of distance and linkage measurement techniques might produce a clustering of several program execution profiles that more accurately captures the similarities or differences between the usages of the program. However, all observation-based testing techniques assume that cluster analysis will be applied to program execution profiles that are of an unknown quality. Thus, it is generally not possible to compare different combinations of distance and linkage measurements by analyzing the semantic meaning of the resulting clusters of program executions. However, it is possible to evaluate the quality of a specific clustering of an input space through the calculation of the cophenetic correlation coefficient, which we denote κ_c.

Suppose that $\mathbf{Y}$ is a vector storing the distances between objects in the input space. Furthermore, assume that $\mathbf{Z}$ is a vector that contains the distances between the clusters that were produced during the linkage phase of cluster analysis. Finally, we denote μ_y and μ_z as the average of the distances in $\mathbf{Y}$ and $\mathbf{Z}$, respectively. Equation 6 describes the calculation of κ_c, which attempts to measure the distortion in the resulting clusters [2]. From Equation 6, is it clear that $\kappa_c \in [0, 1]$. Traditionally, the resultant cluster of the input space is considered to be of a "high quality" when κ_c is very close to 1 [2]. Thus, higher values of κ_c indicate that the objects in the input space nicely fit into the clustering that is described by the resulting dendrogram.

$$\kappa_c = \frac{\sum_{i<j} (\mathbf{Y}_{ij} - \mu_y)(\mathbf{Z}_{ij} - \mu_z)}{\sqrt{\sum_{i<j} (\mathbf{Y}_{ij} - \mu_y)^2 (\mathbf{Z}_{ij} - \mu_z)^2}} \quad (6)$$

III. Data Collection Infrastructure

As stated in Section II-C, cluster analysis algorithms operate on an $(m \times n)$ input matrix $\mathbf{X}$. Each one of the m rows in the input space corresponds to the collection of program spectra for a given execution of a system under test. In this report, we assume that the application that is subject to the analyses required by observation-based testing is written in the Java programming language. In order to support the collection of program spectra from a Java-based application, we developed a data collection infrastructure. Our data collection tool relies upon aspect-oriented programming (AOP) structures to collect method and class call counts spectra. Our data collection infrastructure consists of approximately 1400 lines of well-documented Java code that uses the AspectJ programming language [12] and the log4j logging package [6] to record the behavior of an application.

A. Aspect-Oriented Programming

The aspect-oriented programming paradigm facilitates the implementation of functionality that bridges multiple classes in a software system. In this paradigm, it is possible to express an "aspect" that is composed of "pointcuts" that crosscut one or more entities in a given application [5]. In the AOP paradigm, it is possible for a single aspect to define a pointcut for all of the methods is a specific class. For any given pointcut, it is possible to specify *before*, *after*, and *around* advice [5], [12]. This advice can be used to include arbitrary computations at a specified pointcut in an application. Aspect-oriented programming solutions are often developed for applications that should contain functionality that is pertinent to many classes inside of a given software system. For example, it might be desirable to describe the exception handling behavior for a collection of Java classes as an aspect [14].

B. Data Collection Aspects

The AspectJ programming language can enable the production of a data collection framework for Java applications. Figure 2 shows a small portion of the AspectJ aspect called `Trace` that enables the observation of our candidate application. The first pointcut called `observedClasses` forces all sub-aspects of the `Trace` aspect to define the classes that should be subject to observation. For our purposes, it is important to ensure that *all* classes are subject to the observation provided by our infrastructure. The pointcut `executeMain` allows for inclusion of the data collection steps that must be taken before and after the execution of the `main()` method in our chosen application. Furthermore, the `currentConstructor` and `currentMethod` aspects enable us to collect the appropriate program spectra information before and after the execution of every constructor and method in the classes that have been selected for observation. For each of the defined pointcuts, we used the log4j logging framework to record the desired information in standard text files [6]. When the `Trace` aspect was "woven" into our candidate application using the AspectJ compiler, we were able to transparently extract information about the execution profiles of the program [12].

```
 abstract pointcut observedClasses();

 pointcut executeMain(): observedClasses() &&
    execution(public static void main(String[]));

 pointcut currentConstructor(): observed-
Classes() && execution(new(..));

 pointcut currentMethod(): observedClasses() && exe-
cution(* *(..)) &&
    !execution(public static void main(String[]));
```

Fig. 2. AspectJ Pointcuts in our Data Collection Infrastructure.

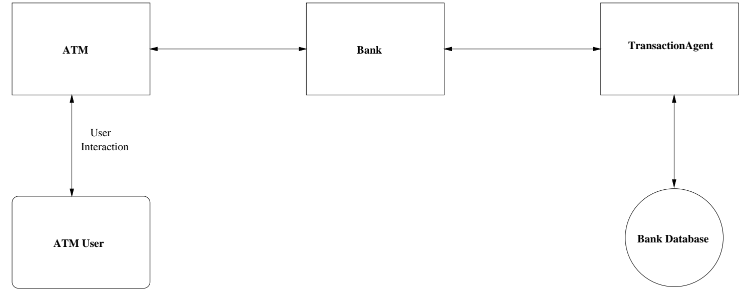

Fig. 3. High-Level Architecture of the Candidate Application.

IV. Experimental Framework

In the research discussed in this report, we were interested in evaluating the underlying assumptions of the observation-based approach to software testing. Furthermore, we wanted to determine if the incorporation of the database interaction information could improve our ability to use cluster analysis to isolate anomalous executions of a chosen software system. To this end, we developed two heuristics for incorporating the state of a database that is used by a Java application. Furthermore, we implemented a simple candidate application in the Java programming language. Next, we used the observation infrastructure described in Section III-B to collect the desired information about the behavior of our application. In the experiment described in Section IV-C, we evaluated the first assumption of observation-based testing. That is, we attempted to determine if cluster analysis was able to effectively group similar usages of our candidate application. The experiment discussed in Section IV-D examined the second assumption of observation-based testing; namely, that it is possible to isolate failure-inducing program executions by looking in small or singleton clusters of an input space. Finally, the experiment described in Section IV-E attempted to evaluate the impact of our database inclusion heuristics on the chosen cluster analysis algorithms.

A. Candidate Application

We developed a simple simulation of an ATM system that contains a graphical user interface. Our application is composed of approximately 4,000 lines of well-documented Java code. The ATM simulation was built and executed with the Java™ 2 Platform, Standard Edition version 1.4.0. The application interacts with a simple database that is managed by a free, open-source, relational database management system called mySQL [19]. Since the candidate program was written in Java, we used the MM mySQL Java Database Connectivity (JDBC) driver to access the information stored in the relational database [1]. The implementation and usage of our candidate application was conducted on a Debian GNU/Linux workstation with a 700 MHz Celeron processor and 256 MB of RAM. In previous research, Kapfhammer et al. have used this software system to address the challenges that occur during attempts to identify and understand problematic software components [10].

Figure 3 depicts the high-level architecture of our simple candidate application. The ATM user interface prompts the user to insert a "card" that describes the pertinent bank account information. Next, the user is prompted for his/her personal identification number (PIN). Finally, the user is able to conduct any number of transaction with his/her account. The currently supported transactions include the ability to deposit or withdraw money, transfer money from one account to another, and check the balance of an existing account. In the current implementation of our candidate application, we assume that all of the entities in the system are running in a single address space.

B. Database State Incorporation Heuristics

We configured the observation infrastructure described in Section III to collect method and class call count information. Each time a method is used, we increment the method call count for that method. Similarly, each invocation of a method inside of a specific class causes the incrementation of the class call count. We also augmented our observation infrastructure so that it could monitor the usage of the methods and classes that actually interact with the mySQL database. Specifically, we collected call count information for methods that had the potential to either *use* or *change* the database. For our purposes, we assumed that the categories of "using the database" and "changing the database" were mutually exclusive. The method `lockAccount` is an example of a method provided by the `TransactionAgent` that would be classified as an operation that changed the database. However, the method `checkBalance` would be considered a method that only uses the database. It is important to note that the collection of usage counts for classes and methods that interact with the database does not provide for the inclusion of information about the state of the database. However, we believe that the simple heuristic of including features that characterize a program's interaction with a database should be evaluated before more complicated database state features are examined.

C. Experiment One: Detecting Similar Execution Profiles

In our first experiment, we executed the candidate program ten times and collected the respective execution profiles. Table I describes the actions that were conducted as examples of normal usage of our candidate application. In this experiment, we created execution profiles such that a pair of executions were deemed to exercise roughly the same functionality. For example, in execution profiles 1 and 2 we simply started our candi-

date application and then immediately exited the program. Furthermore, execution profiles 7 and 8 both performed a withdraw on a specified bank account. However, execution profile 8 contained the inappropriate selection of a primary savings account that was not available to the holder of the ATM card.

Next, we performed cluster analysis with all combinations of distance metrics and linkage techniques. Specifically, we focused on using the Euclidean and City Block distance metrics and the single, complete, average, centroid, and ward linkage mechanisms. After executing all possible configurations of our cluster analysis algorithms, we examine the resulting dendrograms to determine if the similar executions were clustered together. Finally, we produced clusters of the input space with the dendrogram by specifying thresholds for the inconsistency coefficient that governs clustering. For our clustering algorithms, higher values of the inconsistency coefficient enable less similar objects to be clustered together.

Figure 4 provides a dendrogram that graphically depicts the iterative clustering of our input space when the City Block distance metric was combined with the single linkage mechanism. In this example, our input space consisted of class usage counts that were not augmented with our database inclusion heuristics. When this dendrogram was clustered with an inconsistency coefficient of .9, almost every pair of similar profiles was placed in a unique cluster. That is, the first three pairs of execution profiles were place in their respective clusters. However, execution profile 7 was clustered with profile 9 and execution profile 8 was clustered with profile 10. This result is meaningful because profiles 8 and 10 both change the database and include the inappropriate selection of a bank account while profiles 7 and 9 change the database after correctly selecting the desired account.

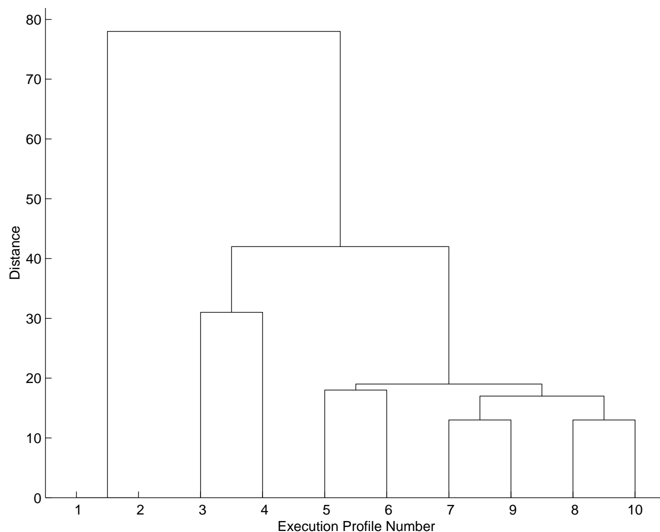

Fig. 4. Dendrogram that Illustrates the Clustering of Similar Execution Profiles.

Several combinations of distance metrics and linkage techniques did not produce accurate characterizations of our simple input space. When the Euclidean distance metric was combined with the single linkage technique and the class counts input space was clustered with an inconsistency coefficient of .9, only two clusters were formed. The first cluster contained execution profiles 1 and 2 and the second cluster contained the remainder of the profiles. When the information about database usage was incorporated into the cluster analysis process, the resulting cluster were the same. Yet, when the call counts for database changing methods were included in the input space, the first six execution profiles were clustered correctly and the final four profiles were collected in a single cluster. This result clearly indicates that the incorporation of our database state heuristics can lead to improved clustering results.

However, we were also interested in determining if usage of our database incorporation heuristics would improve or detract from the quality of clustering when the City Block distance metric was used. To this end, we incorporated all potential combinations of database change and usage information into the input space before executing our cluster analysis algorithms with the City Block distance metric and all of the available linkage techniques. For each of the possible combinations, the incorporation of database interaction information did not improve the clusterings (which, as previously noted, were already of a very high quality). Yet, it is important to note that the usage of our database inclusion heuristics never reduced the apparent quality of the clustering.

D. *Experiment Two: Isolating Anomalous Execution Profiles*

In our second experiment, we also executed our application ten times. However, we required that eight of the execution profiles must represent normal executions of the candidate application and that two of the executions must correspond to abnormal usage patterns. Therefore, during the first execution of the ATM simulation, we repeatedly input the name of our "ATM card" in an inappropriate fashion. While this behavior did not crash the ATM simulation, it did cause the ATM interface to lock in the expected fashion. During the third execution of the ATM simulation, we clicked on the "delete" key when we were prompted to input our PIN number. In this situation, our candidate application attempted to interpret the string "delete" as a number and this inappropriate interpretation eventually caused a `NumberFormatException` to be raised.

Figure 5 provides a dendrogram which clearly shows that the first and third execution profiles are highly dissimilar from the other executions of our candidate application. This dendrogram was the result of the application of the Euclidean distance metric and the single linkage technique on the class call counts input space. As expected, the third execution was considered to be more unusual than the first execution profile. When information about database changes was included in the input space, the resulting dendrogram isolates the abnormal executions in a less stark fashion. However, when database usage information was included in the cluster analysis, the unusual execution profiles were still readily apparent.

E. *Experiment Three: Evaluating Cluster Quality*

The experiments described in Section IV-C and Section IV-D provide anecdotal evidence that unsupervised machine learn-

Profile Number	Execution Steps	Notes
1	open → quit	Variable movement of mouse
2	open → quit	Variable movement of mouse, resized window
3	open → inputCard → quit	Input PIN correctly
4	open → inputCard → quit	Input PIN incorrectly
5	open → inputCard → checkBal → quit	Selected existing primary checking account
6	open → inputCard → checkBal → quit	Initially selected nonexistent secondary checking account
7	open → inputCard → withdraw → quit	Selected existing primary checking account
8	open → inputCard → withdraw → quit	Initially selected nonexistent primary savings account
9	open → inputCard → deposit → quit	Selected existing primary checking account
10	open → inputCard → deposit → quit	Initially selected nonexistent secondary checking account

TABLE I
DESCRIPTION OF PAIRS OF SIMILAR EXECUTION PROFILES.

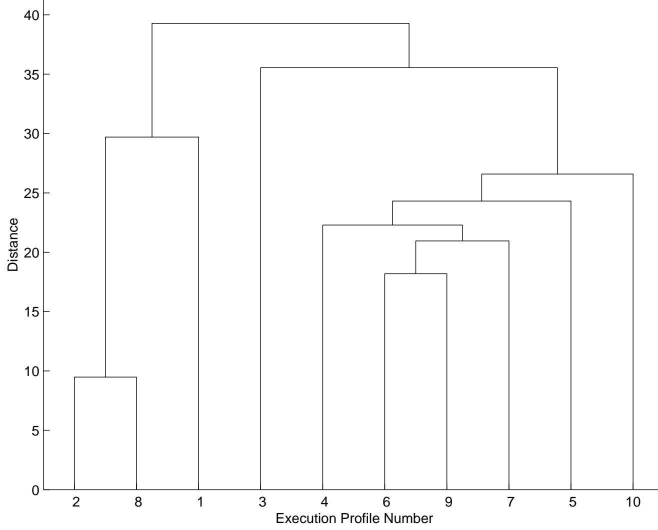

Fig. 5. Dendrogram that Illustrates the Clustering of Mixed Execution Profiles.

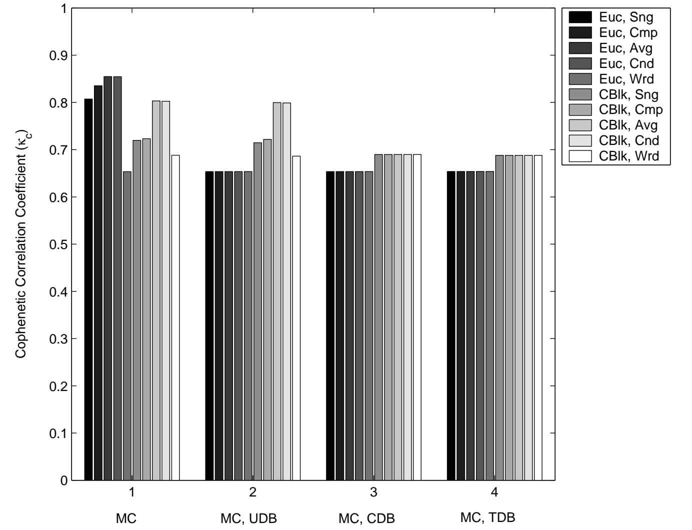

Fig. 6. κ_c for Different Cluster Analysis Configurations.

ing algorithms like cluster analysis can be used to understand the behavior of software systems. Moreover, these experiments show that our simplistic heuristics for incorporating information about database interaction are promising. However, observation-based testing efforts are normally conducted with systems that contain thousands of executions of unknown quality [4], [13]. Since the intent of each execution is almost always unknown, it is generally impossible to evaluate the effectiveness of cluster analysis by means of a detailed, manual cluster inspection. In our final experiment, we executed our candidate application thirty times and measured the cophenetic correlation coefficient for every combination of input space, distance metric, and linkage approach.

Figure 6 shows the resulting measure of cluster quality for varying configurations of the cluster analysis algorithm. Ten measurements of the cophenetic correlation coefficient were computed for different input spaces (represented by the groupings on the horizontal axis) and for different algorithm configurations (represented by individual bars in each horizontal axis grouping). We measured κ_c after using an input space that was composed of just method counts, method counts and database usage counts, method counts and database change counts, and method counts with all database interaction counts. When none of the information provided by our database incorporation heuristics was included in the input space, the values for κ_c varied somewhat significantly based upon choices of distance measures and linkage mechanisms. When database interaction information was incorporated into the input space, the overall quality of the resulting clusters decreased. However when database change information was included in the input space, the clustering algorithms appear to gain a measure of independence from their selected configuration. Similar, although slightly less pronounced patterns emerged when we measure the cophenetic correlation coefficient when the class count information was used as the basis for the input space.

F. Threats to Validity

The main threat to validity for our experiments is the limited number of case studies that were conducted. We only analyzed the applicability of observation-based testing and database in-

corporation heuristics for a single application of limited complexity. Each of the experiments were also limited by the size of the input spaces that were subject to cluster analysis. Also, the types of anomalous program behavior that were recorded in the second experiment did not occur because of problematic interaction with the relational database. Thus, our second experiment did not provide an exceptional testing ground for our heuristics that incorporate database interactions. While our final experiment measured cluster quality through the calculation of the cophenetic correlation coefficient, there is currently no direct evidence that high values for κ_c will actually improve observation-based testing efforts.

V. Conclusion and Future Work

THE observation-based approach to software testing attempts to isolate a small set of meaningful execution profiles from a large collection of unknown quality executions. Current observation-based testing techniques are not specifically tailored for the testing and analysis of applications that interact with databases. To this end, we have evaluated the underlying assumptions of observation-based testing and proposed two simple heuristics for incorporating database interaction into the testing effort. Our experimental analyses have shown that unsupervised machine learning algorithms like cluster analysis can effectively group program executions that exhibit normal behavior. Furthermore, we have shown that cluster analysis has the potential to isolate execution profiles that induce failure or exercise the application in an unusual fashion. More importantly, we have demonstrated that the inclusion of information about the interactions between a database-driven application and a database can improve the results of cluster analysis. We believe that our preliminary results indicate that current heuristics for measuring database interaction and new heuristics for incorporating database state have the potential to improve the observation-based testing for database-driven programs.

In future research, we plan on extending our analysis to new candidate applications. We believe that the incorporation of new software systems will only require minimal modifications to our data collection infrastructure. Furthermore, we are interested in changing our characterization of the input space to include new and different types of program spectra. Also, we feel that the usage of larger input spaces will enable us to better assess the effectiveness and scalability of our cluster analysis algorithms. Moreover, we plan on proposing new heuristics that actually incorporate the state of the database into the input space. For example, the inclusion of access counts for specific tables in a relational database could prove to be profitable. Alternatively, it might be possible to incorporate information about the primary key of the specific table row that was accessed during an interaction with the database. After cluster analysis, all observation-based testing techniques must sample the input space for unusual or failure-inducing program executions. In future work, we will implement several methods for the selection of execution profiles and empirically evaluate their effectiveness with different configurations of the input space and the clustering algorithms.

References

[1] MM mySQL JDBC drivers. 2001. http://mmmysql.sourceforge.net/.

[2] MathWorks Corporation. Statistics toolbox. 2001. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/stats.shtml.

[3] Edward E. Cureton and Ralph B. D'Agostino. *Factor Analysis: An Applied Approach*. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1983.

[4] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the 9th International Conference on the Foundations of Software Engineering*, pages 246–255. ACM Press, November 2001.

[5] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: An introduction. *Communications of the ACM*, 44(10):59–65, October 2001.

[6] Apache Software Foundation. Jakarta log4j. 2002. http://jakarta.apache.org/log4j/.

[7] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Fransisco, CA, 2000.

[8] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proceedings of the ACM/SIGSOFT Workshop on Program Analysis and Software Tools and Engineering*, pages 83–90. ACM Press, June 1998.

[9] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[10] Gregory M. Kapfhammer, C.C. Michael, Jennifer Haddox, and Ryan Colyer. An approach to identifying and understanding problematic COTS components. In *Proceedings of the 2nd International Software Assurance and Certification Conference*, Reston, Virginia, September 2000.

[11] Maurice Kendall. *Multivariate Analysis*. MacMillian Publishing Co, Inc., New York, NY, 1980.

[12] Gregor Kiczales, Erik Hillsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[13] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 116–125. ACM Press, November 2000.

[14] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427. ACM Press, 2000.

[15] Brian Marick. *The Craft of Software Testing*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[16] Andy Podgurski, Wassim Masri, Yolanda McCleese, and Francis G. Wolff. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, 8(3):263–283, July 1999.

[17] Helmuth Spath. *Cluster Analysis Algorithms for Data Reduction and Classification of Objects*. Ellis Horwood Limited, Chichester, West Sussex, England, 1980.

[18] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167, August 2000.

[19] Randy Jay Yarger, George Reese, and Tim King. *MySQL and mSQL*. O'Reilly, 1999.