Chapter 1

DATA MINING FOR SOFTWARE TESTING

Mark Last
Ben-Gurion University of the Negev

Abstract Software testing activities are usually planned by human experts, while test automation tools are limited to execution of pre-planned tests only. Evaluation of test outcomes is also associated with a considerable effort by software testers who may have imperfect knowledge of the requirements specification. Not surprisingly, this manual approach to software testing results in heavy losses to the world's economy. As demonstrated in this chapter, data mining algorithms can be efficiently used for automated modeling of tested systems. Induced data mining models can be utilized for recovering system requirements, identifying equivalence classes in system inputs, designing a minimal set of regression tests, and evaluating the correctness of software outputs.

Keywords: Automated Software Testing, Regression Testing, Input-Output Analysis, Info-Fuzzy Networks, Artificial Neural Networks

1. INTRODUCTION

A recent study by the National Institute of Standards & Technology (NIST, 2002) found that "the national annual costs of an inadequate infrastructure for software testing is estimated to range **from $22.2 to $59.5 billion**" (p. ES-3) which are about 0.6 percent of the US gross domestic product. This number does not include costs associated with catastrophic failures of mission-critical software (such as the Patriot Missile Defense System malfunction in 1991 and the $165 million Mars Polar Lander shutdown in 1999). According to another report, U.S. Department of Defense alone loses over **four billion dollars** a year due to software failures.

A program *fails* when it does not meet the requirements (Pfleeger, 2001). The purpose of testing a program is to discover faults that cause the system to fail rather than proving the program correctness (ibid). Some even argue that the program correctness can never be verified through software testing (DeMillo and Offlut, 1991). The reasons for that include such issues as the size

of the input domain, the number of possible paths through the program, and wrong or incomplete specifications. A successful test is a test that reveals a problem in software; tests that do not expose any faults are useless, since they hardly provide any indication that the program works properly (Kaner *et. al*., 1999). In developing a large system, the test of the entire application (*system testing*) is usually preceded by the stages of *unit testing* and *integration testing* (Pfleeger, 2001). The activities of testing a new system include *function testing, performance testing, acceptance testing*, and *installation testing. Functional (black-box) testing* is based solely on functional requirements of the tested system, while *structural (white-box) testing* is based on the code itself. The purpose of *regression testing* is to verify that the basic functionality of the software has not changed as a result of adding new features or correcting existing faults.

It is well-known that most modules in a large software system are usually quite simple and not prone to failure. The developers and the testers are interested to identify those few modules, which are complex enough to justify extensive testing. Module complexity can be evaluated using several measures, but the relationships between these measures and the likelihood of a module failure are still not fully understood. As shown by (Dick and Kandel, 2004), these relationships can be automatically induced by data mining methods from software metrics datasets that are becoming increasingly available in the private and public domain.

In a properly documented system, behavior requirements are covered by a series of *use-case scenarios* (Pfleeger, 2001). Use cases are further modeled by *Activity*, *Sequence*, and *Collaboration Diagrams*. Activity diagrams represent a slightly enhanced version of conventional flow charts, while sequence and collaboration diagrams apply mainly to object-oriented systems. The purpose of these requirement models is to completely specify the system actions and outputs in response to inputs obtained from the environment. Ideally, a minimal test set can be generated from a complete and up-to-date specification of functional requirements. Unfortunately, frequent changes make the original requirements documentation, even if once complete and accurate, hardly relevant to the new versions of software (El-Ramly *et. al*., 2002). To address this problem of "software requirements loss", El-Ramly et al. (2002) have developed an interaction-pattern mining method for the recovery of functional requirements as usage scenarios.

Each choice of input testing data is called a *test case*. The description of a test case usually includes pre-conditions for test case execution, actual inputs, and expected outputs (Jorgensen, 2002). A set of one or more test cases aimed at a single test object is called a *test suite*. Since the testing resources are always limited, each executed test case should have a reasonable probability of detecting a fault along with being non-redundant, effective, and of a proper

complexity (Kaner *et. al*., 1999). It should also make the program failure obvious to the tester who knows, or is supposed to know, the expected outputs of the system. Thus, selection of the tests and evaluation of their outputs are crucial for improving the quality of tested software with less cost.

If the requirements can be re-stated as logical relationships between inputs and outputs, functional test cases can be generated automatically by such techniques as cause-effect graphs (Pfleeger, 2001) and decision tables (Beizer, 1990; Jorgensen, 2002). To ensure effective design of new regression test cases, one has to recover (reverse engineer) the actual requirements of an existing system. In (Schroeder and Korel, 2000), several manual ways are proposed to determine input-output relationships in tested software. Thus, a tester can analyze system specifications, perform structural analysis of the system's source code, or observe the results of system execution. While available system specifications may be incomplete or outdated, especially in the case of a "legacy" application, and the code may be poorly structured, the *data* provided by the system execution remains the most reliable source of information on the actual functionality of an evolving system. The usage of various data mining methods for automated induction of input-output relationships from software execution data is demonstrated in (Vanmali *et. al*., 2002; Last and Kandel, 2003; Last *et. al*., 2003; Last *et. al*., 2004a; Saraph *et. al*., 2004; Last and Friedman, 2004).

2. MINING SOFTWARE METRICS DATABASES

Software metrics are the key tools in software quality management (Dick and Kandel, 2004). In the system development process, software metrics are collected at various points in the life cycle, and used to identify system modules that are potentially error-prone, so that extra development and maintenance effort can be directed at those modules. The Pareto Law seems to hold for most software systems implying that 80% of a system's bugs will be found in just 20% of the system's modules.

Each software metric quantifies some characteristic of a program. The simplest counting metric is the number of logical or physical lines of source code, but it is hard to find a general relationship between lines of code and defect rate (Hoppner *et. al*., 1999). Consequently, more sophisticated metrics of program complexity have been proposed (Kan, 2003). Thus, Halstead's software metrics take into account the number of operators and operands in a program. More complex metrics such as McCabe's cyclomatic complexity are based on the assumption that the program can be represented as a strongly connected graph with unique entry and exit points. While different metrics do measure different characteristics, the various metrics tend to be strongly correlated to each other and to the number of failures in a program (Dick and Kandel, 2004). Furthermore, there tend to be relatively few modules in any given system that

have a high degree of complexity. As a result, any database of software characteristics and defects will be heavily skewed towards simple modules with a small number of failures.

The data mining community has developed several approaches to overcome skewness in a dataset. The two most common are *cost-sensitive learning* and *resampling*. By associating unequal costs with different types of errors, a cost-sensitive learner can avoid certain kinds of errors, at the cost of making more errors of a different type. The resampling techniques used in (Dick and Kandel, 2004) are sometimes also referred to as *stratification*. The simplest approach is under-sampling, wherein a subset of majority-class examples are randomly selected for inclusion in the resampled dataset *without* replacement. This effectively thins out the majority class, making the dataset more homogeneous.

To identify clusters of similar software modules, Dick et al. (2004) conducted a series of fuzzy clustering experiments on three real-world software metrics datasets. Fuzzy clustering was carried out using the Fuzzy c-Means algorithm (Hoppner *et. al*., 1999) as implemented in MATLAB®6.0. The resulting fuzzy clusters were converted into "crisp" classes using the method of maximum membership and the C4.5 decision tree inducer was applied to the three datasets with examples being selected by a stratified sampling algorithm. The classes having unusually high metric values have been identified as "classes of interest" for stratification purposes. In general, "uninteresting classes" were undersampled in (Dick and Kandel, 2004) to as little as 25% of their original population, while "interesting classes" were oversampled by 100 or 200%. All of the resampling strategies produced an improvement in the overall accuracy as the sampling rate was decreased for uninteresting classes and increased for interesting classes. However, the optimal combination of undersampling and oversampling strategies appears to be specific to each dataset. According to (Dick and Kandel, 2004), the calibration of a data mining model using resampling techniques can be best carried out through the construction of a throw-away prototype of the actual system, which would lead to higher-quality software in general.

3. INTERACTION-PATTERN DISCOVERY IN SYSTEM USAGE DATA

As software systems get regularly maintained throughout their lifecycle, the documentation of their requirements often becomes obsolete or get lost. According to a recent US General Accounting Office report (US GAO, 2004), in many defense projects, "the requirements lacked the specific information necessary to understand the required functionality that was to be provided and did not describe how to determine quantitatively, through testing or other analysis, whether the systems would meet Army's respective needs." (p. 60). To ad-

dress the problem of "software requirements loss", El-Ramly et al. (2002) have developed an interaction-pattern mining method for the recovery of functional requirements as usage scenarios. Their interaction-pattern discovery algorithm, called IPM2, analyzes traces of the run-time system-user interaction to discover frequently recurring patterns; these patterns correspond to the functionality currently exercised by the system users, represented as *usage scenarios*.

The IPM2 algorithm is based on the state-transition model of the system interface behavior. Each state of the model corresponds to a distinct screen in the system user interface. A transition between system states is caused by a user action such as a sequence of cursor movements or keystrokes. A usage scenario is represented as a sequence of screen IDs. An ordered set of screen IDs occurring in a given sequence is called an *episode*. An episode is considered "interesting" based on some user-defined criteria (e.g., minimum support). An ordered set of screen IDs that exist in every interesting episode is called a "pattern." The IPM2 algorithm is looking for all maximal qualified patterns in a set of sequences (usage scenarios). In a case study presented in (El-Ramly *et. al*., 2002), 12 frequent patterns have been discovered in the run-time traces of a legacy system having 26 states (screens).

4. USING DATA MINING IN FUNCTIONAL TESTING

The architecture of the DM-based environment for automated black-box testing is shown in Figure $86-1$. The Random Tests Generator (RTG) Module obtains the list of system inputs, their types (discrete, continuous, etc.), and ranges from the Specification of System Inputs (SSI) Module. No information about system functionality is needed, since data mining algorithms can automatically reveal the functional requirements from a training set of randomly generated test cases. We have found empirically in (Last, Friedman, & Kandel, 2003) that a relatively small number of 1,000 training cases is sufficient to obtain a reasonably accurate model of the tested system. Systematic approaches to training set generation, based on the usage profile, may also be considered.

Data mining algorithms are trained on execution data, which includes the inputs provided by the RTG module and the corresponding outputs obtained from a legacy system by means of the Test Bed module. In our methodology we assume that the legacy system is stable, i.e. it has been in use for a sufficiently long period of time to eliminate nearly all errors. Models that can be induced by data mining algorithms from execution data include Artificial Neural Networks (Vanmali *et. al*., 2002), Single-Target Info-Fuzzy Networks (Last *et. al*., 2003), and Multi-Target Info-Fuzzy Networks (Last and Friedman, 2004). The following information can be usually derived from an induced data mining model:

A list of input attributes relevant to a single output (in a single-objective model) or to several outputs (in a multi-objective model). This list can be usually derived from the structure of the induced model. As shown in (Schroeder and Korel, 2000), the knowledge of input-output relationships can significantly reduce the amount of test cases.

Logical (if. . . then. . .) rules expressing the functional requirements as logical relationships between the selected input attributes and the system outputs. These rules can be used to determine the predicted (most probable) value of each output in a regression test case. Thus, an induced model can be used as an "automated oracle" for testing a later version of the system when no human expert with perfect knowledge of system requirements is available.

Discretization intervals of each continuous input attribute included in the model. In testing terms, each interval represents an "equivalence class", since for all values of a given interval the output values follow the same distribution.

A set of non-redundant test cases to be stored in the regression test library. The model structure can be converted into test cases representing non-redundant conjunctions of input values. For example, each terminal node in an info-fuzzy network represents a single test case.

Automated oracle. The induced DM model can be used to determine the predicted (most probable) value of the output in each test case. The information about the distribution of output values at the corresponding terminal nodes can also be used to evaluate the correctness of actual outputs produced by a new version of the tested system.

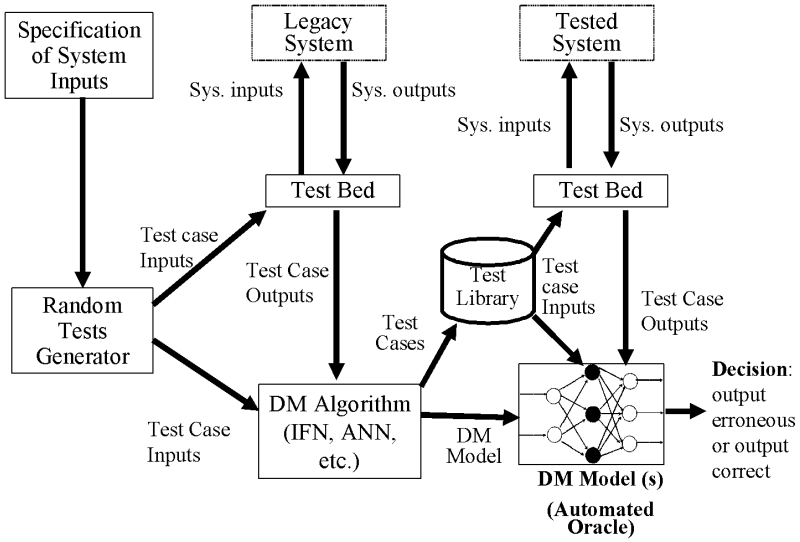

Figure 1.1. DM-Based Functional Testing

The role of each module in the DM-based testing environment is summarized below:

Specification of System Inputs (SSI). Basic data on each input variable includes variable name, type (discrete, continuous, nominal, etc.), and a list or a range of possible values. Such information is generally available from requirements management and test management tools (e.g., Rational RequisitePro®or TestDirector®).

Random Tests Generator (RTG). This module generates random combinations of values in the range of each input variable. Variable ranges are obtained from the SSI module (see above). The number of random test cases to generate depends on the complexity of the tested application. The generated test cases are used to train the DM module. If the predictive accuracy of the induced data mining model is not sufficient, the number of training cases can be increased.

Test Bed (TB). This module, sometimes called "test harness", feeds test cases generated by the RTG module to the software system, which may be a legacy version (when training a data mining algorithm) or a new, tested version (when using data mining models as automated oracles in regression testing).

DM Algorithm (DM). This module can be based on any classification or prediction algorithm that can induce an input-output model of the tested system from execution data. The following algorithms have already been used for this task: ANN – Artificial Neural Network (Saraph *et. al*., 2004), IFN - single-target Info-Fuzzy Network (Maimon and Last, 2000), and M-IFN - Multi-target Info-Fuzzy Network (Last and Friedman, 2004). The input to each DM algorithm includes test cases randomly generated by the RTG module and outputs produced by the legacy version for each random test case. The algorithms also use the descriptions of variables stored by the SSI module. A single-objective classification algorithm (such as IFN) is needed to run repeatedly for each system output, while a multi-objective algorithm (such as M-IFN) is run only once for all outputs. Actual test cases can be generated from the automatically detected equivalence classes by using a standard equivalence class testing technique (weak normal testing, strong normal testing, etc.). More details on equivalence class testing can be found in (Jorgensen, 2002).

In (Last *et. al*., 2003), we have applied the single-target IFN algorithm to the Unstructured Mesh Finite Element Solver (UMFES) which is a general finite element program coded in FORTRAN 77 using about 3,000 lines. It has four continuous inputs and five continuous outputs. The results in (Last *et. al*., 2003) have shown the effectiveness of a single-target IFN in discriminating between the correct and the faulty versions of tested software. Similar results have been obtained in (Last and Friedman, 2004) with the M-IFN algorithm.

Saraph et al. (2004) have presented a test case generation and reduction methodology based on a set of neural network algorithms described in (Setiono and Liu, 1995). A three-layer feed-forward network is built first using

the weight decay back-propagation algorithm. The weight decay mechanism assigns bigger magnitudes to important weights and smaller values to the unimportant weights. In the pruning phase these smaller weights are removed without affecting the predictive accuracy of the network. The inputs of the program are then ranked in the order of their importance by observing the order in which the links in the network are removed in the pruning phase. This ranking of features helps identify attributes contributing to the value of output. We can thus focus on the most significant inputs implied by the ranking and use that information for building the test cases. Test cases involving the attributes with higher ranking can be included and test cases involving attributes receiving lower ranks can be eliminated. The rule-extraction phase deduces rules refining the input-output relationships extracted by the pruning phase. It uses a clustering algorithm to build equivalence classes for continuous input attributes. The test cases are built by taking combinations of data values of the inputs. In (Saraph *et. al.*, 2004), the methodology has been applied to a multi-output business application, where it has automatically identified the most significant inputs affecting the application outputs. Identification of these relationships has caused a major reduction in the number of test cases as compared to exhaustive testing.

5. SUMMARY

In this chapter, we have presented emerging methodologies for applying data mining methods to automated testing of software systems. The advantages of data mining in the testing domain include:

The functional relationships can be induced automatically from execution data. Current methods of test design assume existence of detailed requirements, which may be unavailable, incomplete or unreliable in many legacy systems.

The data mining methods are applicable to modeling complex software programs.

The data mining methods can automatically produce a set of non-redundant test cases covering the most common functional relationships existing in software.

Issues for ongoing research include applying more data mining methods to software testing problems and application of the proposed methodology to large-scale software systems.

Acknowledgments

This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248.

References

Beizer, B., Software Testing Techniques. 2nd Edition (Thomson, 1990).

DeMillo, R.A. and Offlut, A.J., Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17(9): 900-910, 1991.

Dick, S. and Kandel, A. "Data Mining with Resampling in Software Metrics Databases." In M. Last, A. Kandel, and H. Bunke (Editors), Artificial Intelligence Methods in Software Testing, World Scientific, pp. 175-208, 2004.

Dick S., Meeks A., Last M., Bunke H., Kandel A., "Data Mining in Software Metric Databases", Fuzzy Sets and Systems, Vol. 145, Issue 1, pp. 81-110, July 2004.

El-Ramly M., Stroulia E., Sorenson P. From Run-time Behavior to Usage Scenarios: An Interaction-pattern Mining Approach, in Proceedings of KDD-2002, Edmonton, Canada (July 2002). ACM Press, 315 – 327.

Hoppner F., Klawonn F., Kruse R., Runkler T., Fuzzy Cluster Analysis: Methods for Classification, Data Analysis and Image Recognition, New York: John Wiley and Sons, Inc., 1999.

Jorgensen, P. C., Software Testing: A Craftsman's Approach. Second Edition, CRC Press, 2002.

Kan, S. H., Metrics and Models in Software Quality Engineering, 2nd Edition, Addison Wesley, 2003

Kaner, C., Falk, J., Nguyen, H.Q., Testing Computer Software, Wiley, 1999.

Last, M. and Friedman, M. "Black-Box Testing with Info-Fuzzy Networks", in M. Last, A. Kandel, and H. Bunke (Editors), "Artificial Intelligence Methods in Software Testing", World Scientific, pp. 21-50, 2004.

Last M., Friedman M., and Kandel A. "The Data Mining Approach to Automated Software Testing", Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003), pp. 388 - 396, Washington, DC, USA August 24 - 27, 2003.

Last M., Friedman M., and Kandel A., "Using Data Mining for Automated Software Testing", to appear in the International Journal of Software Engineering and Knowledge Engineering, Special Issue on Data Mining for Software Engineering and Knowledge Engineering, Vol. 15, No. 5, October 2004a.

Last, M. and Kandel, A. Automated Test Reduction Using an Info-Fuzzy Network, in Software Engineering with Computational Intelligence, ed. T.M. Khoshgoftaar (Kluwer Academic Publishers, 2003). pp. 235 – 258.

Last, M., Kandel, A., and Bunke, H. (Editors). "Artificial Intelligence Methods in Software Testing". World Scientific, Series in Machine Perception and Artificial Intelligence, Vol. 56, 2004b.

Maimon, O. and Last, M., Knowledge Discovery and Data Mining – The Info-Fuzzy Network (IFN) Methodology, (Kluwer Academic Publishers, Massive Computing, Boston, December 2000).

National Institute of Standards and Technology. "The Economic Impacts of Inadequate Infrastructure for Software Testing". Planning Report 02-3 (May 2002).

Pfleeger, S. L., Software Engineering: Theory and Practice. 2nd Edition, Prentice-Hall, 2001.

Saraph, P., Last, M., Kandel, A., "Test Set Generation and Reduction with Artificial Neural Networks", in M. Last, A. Kandel, and H. Bunke (Editors), "Artificial Intelligence Methods in Software Testing", World Scientific, pp. 101-132, 2004.

Schroeder P. J. and Korel, B. Black-Box Test Reduction Using Input-Output Analysis. In Proc. of ISSTA '00. pp 173-177, 2000.

Setiono R. and Liu, H. Understanding Neural Networks via Rule-extraction, IJCAI 1995.

United States General Accounting Office, Report to Congressional Requesters, DOD Business Systems Modernization, Billions Continue to Be Invested with Inadequate Management Oversight and Accountability, May 2004.

Vanmali M., Last M., Kandel A., Using a Neural Network in the Software Testing Process, International Journal of Intelligent Systems 2002, 17, 1: 45-62.