The data mining approach to automated software testing

Conference Paper · January 2003

DOI: 10.1145/956750.956795 · Source: DBLP

3 authors, including:

Mark Last
Ben-Gurion University of the Negev
193 PUBLICATIONS **2,699** CITATIONS
SEE PROFILE

Menahem Friedman
162 PUBLICATIONS **4,681** CITATIONS
SEE PROFILE

The Data Mining Approach to Automated Software Testing

Mark Last
Dept. of Information Systems Eng.
Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
mlast@bgumail.bgu.ac.il

Menahem Friedman
Department of Physics
Nuclear Research Center – Negev
Beer-Sheva, POB 9001, Israel
mlfrid@netvision.net.il

Abraham Kandel
Dept. of Comp. Science and Eng.
University of South Florida
Tampa, FL 33620, USA
kandel@csee.usf.edu

ABSTRACT
In today's industry, the design of software tests is mostly based on the testers' expertise, while test automation tools are limited to execution of pre-planned tests only. Evaluation of test outputs is also associated with a considerable effort by human testers who often have imperfect knowledge of the requirements specification. Not surprisingly, this manual approach to software testing results in heavy losses to the world's economy. The costs of the so-called "catastrophic" software failures (such as Mars Polar Lander shutdown in 1999) are even hard to measure. In this paper, we demonstrate the potential use of data mining algorithms for automated induction of functional requirements from execution data. The induced data mining models of tested software can be utilized for recovering missing and incomplete specifications, designing a minimal set of regression tests, and evaluating the correctness of software outputs when testing new, potentially flawed releases of the system. To study the feasibility of the proposed approach, we have applied a novel data mining algorithm called Info-Fuzzy Network (IFN) to execution data of a general-purpose code for solving partial differential equations. After being trained on a relatively small number of randomly generated input-output examples, the model constructed by the IFN algorithm has shown a clear capability to discriminate between correct and faulty versions of the program.

Categories and Subject Descriptors

D.2.5 [**Software**]: Testing and Debugging – *Testing Tools*; H.2.8 [**Database Management**]: Database Applications – *Data Mining*; I.2.6 [**Computing Methodologies**]: Artificial Intelligence – *Induction*.

General Terms

Algorithms, Experimentation, Verification.

Keywords
Automated Software Testing, Regression Testing, Input-Output Analysis, Info-Fuzzy Networks, Finite Element Solver.

SIGKDD '03, August 24-27, 2003, Washington, DC, USA.

1 MOTIVATION AND BACKGROUND
A recent study by the National Institute of Standards & Technology [32] 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) or 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 $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 do what it is required to do [33]. 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 demonstrated through software testing [7]. The reasons for that include such complexity issues as the size of the input domain, the number of possible paths through the program, and wrong or incomplete specifications. A successful test should reveal a problem in software; tests that do not expose any faults are useless, since they hardly provide any indication that the program works properly [17]. 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* [33]. The activities of system testing include *function testing, performance testing, acceptance testing*, and *installation testing.*

The ultimate goal of function testing is to verify that the system performs its functions as specified in the requirements and there are no undiscovered errors left. Since proving the code correctness is not feasible, especially for large software systems, the practical testing is limited to a series of experiments showing the program behavior in certain situations. Each choice of input testing data is called a *test case*. If the structure of the tested program itself is used to build a test case, this is called a *white-box* (or open-box) approach. Several white-box methods for automated generation of test cases are described in literature. For example, the technique of [7] uses mutation analysis to create test cases for unit and module testing. A test set is considered adequate if it causes all mutated (incorrect) versions of the program to fail. The idea of testing programs by injecting simulated faults into the code is further extended in [38]. Another paper [39] presents a family of strategies for automated generation of test cases from Boolean specifications. However, as indicated by [38], modern software systems are too large to be tested by the white-box approach as a single entity. White-box testing techniques can work only at the subsystem level. In function tests that are aimed at checking that a complex software system meets its specification, *black-box* (or closed box) test

cases are much more common. The actual outputs of a black-box test case are compared to expected outputs based on the tester's knowledge and understanding of the system requirements.

Since the testers have time for only a limited number of test cases, each test case should have a reasonable probability of detecting a fault along with being non-redundant, effective, and of a proper complexity [17]. It should also make program failures obvious to the tester who 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 functional requirements are current, clear, and complete, they can be used as a basis for designing black-box test cases. Assuming that requirements can be re-stated as logical relationships between inputs and outputs, test cases can be generated automatically by such techniques as cause-effect graphs (see [33]) and decision tables [2]. Another method for automatic generation of test vectors from functional relationships is described in [3].

To stay useful, any software system, whether it is an open source code, an off-the-shelf package or a custom-built application, has to undergo continual changes. Most common maintenance activities in software life-cycle include bug fixes, minor modifications, improvements of basic functionality, and addition of brand new features. The purpose of *regression testing* is to identify new faults that may have been unintentionally introduced into the basic features as a result of enhancing software functionality or correcting existing faults. According to [32], "as software evolves, regression testing becomes one of the most important and extensive forms of testing" (p. A-1).

A *regression test library* or *regression suite* is a set of test cases that are run automatically whenever a new version of the software is submitted for testing. Such a library should include a minimal number of tests that cover all possible aspects of system functionality. The standard way to design a minimal regression suite is to identify *equivalence classes* of every input and then use only one value from each edge (boundary) of every class. Even with a limited number of equivalence classes, this approach leads to an amount of test cases, which is exponential in the number of inputs. Besides, the manual process of identifying equivalence classes in a piece of software is subjective and inaccurate [17] and the equivalence class boundaries may change over time.

Ideally, a minimal test suite 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 [10]. To ensure effective design of new regression test cases, one has to recover (reverse engineer) the actual requirements of an existing system. In [36], several 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, and 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, execution data seems to be the most reliable source of information on the actual functionality of an evolving system.

In this paper, we extend the idea initially introduced by us in [21] that input-output analysis of execution data can be automated by the IFN (Info-Fuzzy Network) methodology of data mining [26] [22]. In [21] the proposed concept of IFN-based testing has been demonstrated on individual discrete outputs of a small business program. The current study evaluates the effectiveness of the IFN methodology on a complex expert-system application having multiple continuous outputs. This is also the first time that we deal with the question of determining the minimal number of training cases required to construct the IFN-based model of a given software system.

The rest of the paper is organized as follows. Section 2 provides the background on info-fuzzy networks and their representation capabilities. Section 3 presents the info-fuzzy method of input-output analysis and test case evaluation with a special emphasis on systems having multiple continuous outputs. Section 4 describes a detailed case study of the proposed methodology. Finally, Section 5 summarizes the paper with initial conclusions and directions for future research and applications.

2 INFO-FUZZY NETWORKS: AN OVERVIEW

2.1 Info-Fuzzy Network Structure

A comprehensive description of the info-fuzzy network (IFN) methodology for data-driven induction of predictive models is provided in [26]. Info-fuzzy network (see Figure 1) has an "oblivious" tree-like structure, where the same input attribute is used across all nodes of a given layer (level). The network components include the root node, a changeable number of hidden layers (one layer for each selected input), and the target (output) layer representing the possible output values. Each target node is associated with a value (class) in the domain of a target attribute. The target layer has no equivalent in decision trees, but a similar concept of *category nodes* is used in *decision graphs* [19]. If the IFN model is aimed at predicting the values of a continuous target attribute, the target nodes represent disjoint intervals in the attribute range. There is no limit as to the maximum number of output nodes in an info-fuzzy network. The sample network in Figure 1 has three output nodes denoted by numbers 1, 2, and 3.

A hidden layer No. l consists of nodes representing conjunctions of values of the first l input attributes, which is similar to the definition of an internal node in a standard decision tree. For continuous inputs, the values represent intervals identified automatically by the network construction algorithm. In Figure 1, we have two hidden layers (No. 1 and No. 2). Like in Oblivious Read-Once Decision Graphs [19], all nodes of a hidden IFN layer are labeled by the same feature. However, IFN extends the "read-once" restriction to continuous input attributes by allowing multi-way splits of a continuous domain at the same network level.

The final (terminal) nodes of the network represent non-redundant conjunctions of input values that produce distinct outputs. The five terminal nodes of Figure 1 include (1,1), (1,2), 2, (3,1), and (3,2). If the network is induced from execution data of a software system, each interconnection between a terminal and a target node represents a possible output of a test case. For example, the connection (1,1) → 1 in Figure 1 means that we expect the output value of 1 for a test case where both input variables are equal to 1. The connectionist nature of IFN resembles the structure of a multi-layer neural network. Accordingly, we characterize our model as a *network* and not as a *tree*.

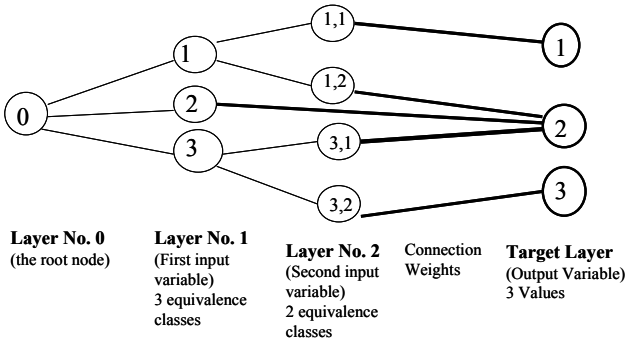

Figure 1 Info-Fuzzy Network - An Example

The idea of restricting the order of input attributes in graph-based representations of Boolean functions, called "function graphs", has proven to be very useful for automatic verification of logic design, especially in hardware. Bryant [5] has shown that each Boolean function has a unique (up to isomorphism) reduced function graph representation, while any other function graph denoting the same function contains more vertices. Moreover, all symmetric functions can be represented by graphs where the number of nodes grows at most as the square of the number of input attributes. As indicated by Kohavi [18], these properties of Boolean function graphs can be easily generalized to oblivious read-once decision graphs representing *k*-categorization functions. Consequently, the "read-once" structure of info-fuzzy networks makes them a natural modelling technique for testing complex software systems.

2.2 Network Induction Algorithm

Each output variable is represented by a separate info-fuzzy network. Thus, without loss of generality, we describe here an algorithm for constructing a network of a single output variable. The induction procedure starts with defining the target layer (one node for each target interval or class) and the "root" node representing an empty set of input attributes. The input attributes are selected incrementally to maximize a global decrease in the conditional entropy of the target attribute. Unlike CART™ [4], C4.5 [34], and EODG [19], the IFN algorithm is based on the pre-pruning approach: when no attribute causes a statistically significant decrease in the entropy, the network construction is stopped. In this paper, we focus on the selection of continuous input attributes, which present in our case study. The treatment of discrete input attributes by the IFN algorithm is covered elsewhere (e.g. see [22]).

The algorithm performs discretization of continuous input attributes "on-the-fly" by using an approach, which is similar to the information-theoretic heuristic of Fayyad and Irani [11]: recursively finding a binary partition of an input attribute that minimizes the conditional entropy of the target attribute. However, the stopping criterion we are using is different from [11]. Rather than searching for a *minimum description length* (minimum number of bits for encoding the training data), we make use of a standard statistical *likelihood-ratio test* [35]. The search for the best partition of a continuous attribute is *dynamic*: it is performed each time a candidate input attribute is considered for selection. Each hidden node in the network is associated with an interval of a discretized input attribute.

We calculate the estimated conditional mutual information between the partition of the interval S at the threshold Th and the target attribute T given the node z by the following formula (based on [6]):

$$MI\ (Th\ ; T\ /\ S\ ,\ z\ ) =$$

$$\sum_{t=0}^{M_T-1}\sum_{y=1}^{2} P(S_y; C_t; z) \bullet \log \frac{P(S_y; C_t / S, z)}{P(S_y / S, z) \bullet P(C_t / S, z)}$$

Where

$P(S_y / S, z)$ is an estimated conditional (a posteriori) probability of a sub-interval S_y, given the interval S and the node z;

$P(C_t / S, z)$ is an estimated conditional (a posteriori) probability of a value C_t of the target attribute T given the interval S and the node z;

$P(S_y; C_t / S, z)$ is an estimated joint probability of a value C_t of the target attribute T and a sub-interval S_y given the interval S and the node z; and

$P(S_y; C_t; z)$ is an estimated joint probability of a value C_t of the target attribute T, a sub-interval S_y, and the node z.

The statistical significance of splitting the interval S by the threshold Th at the node z is evaluated using the likelihood-ratio statistic (based on [35]):

$$G^2(Th; T / S, z) = 2\sum_{j=0}^{M_t-1}\sum_{y=1}^{2} N_{ij}(S_y, z) \bullet \ln \frac{N_t(S_y, z)}{P(C_t / S, z) \bullet E(S_y, z)}$$

Where

$N_t(S_y, z)$ is the number of occurrences of the target value C_t in sub-interval S_y and the node z;

$E(S_y, z)$ is the number of records in sub-interval S_y and the node z;

$P(C_t / S, z)$ is an estimated conditional (a posteriori) probability of the target value C_t given the interval S and the node z; and

$P(C_t / S, z) \bullet E(S_y, z)$ - an estimated number of occurrences of the target value C_t in sub-interval S_y and the node z under the assumption that the conditional probabilities of the target attribute values are identically distributed over each sub-interval.

The Likelihood-Ratio Test is a general-purpose method for testing the null hypothesis H_0 that two random variables are statistically independent. If H_0 holds, then the likelihood-ratio test statistic $G^2(Th; T / S, z)$ is distributed as chi-square with $NT(S, z) - 1$ degrees of freedom, where $NT(S, z)$ is the number of values of the target attribute in the interval S at node z. The default significance level (*p-value*) used by the IFN algorithm is 0.1%.

A new input attribute is selected to maximize the total significant decrease in the conditional entropy, as a result of splitting the nodes of the last layer. The nodes of a new hidden layer are defined for a Cartesian product of split nodes of the previous hidden layer and discretized intervals of the new input variable. If there is no candidate input variable significantly decreasing the conditional entropy of the output variable the network construction stops. In Figure 1, the first hidden layer has three nodes related to three intervals of the first input variable, but only

nodes 1 and 3 are split, since the conditional mutual information as a result of splitting node 2 proves to be statistically insignificant. For each split node of the first layer, the algorithm has created two nodes in the second layer, which represent the two intervals of the second input variable. None of the four nodes of the second layer are split, because they do not provide a significant decrease in the conditional entropy of the output.

A complete algorithm for constructing an info-fuzzy network from a set of continuous input attributes is presented in [22] and [26]. The IFN induction procedure is a greedy algorithm, which is not guaranteed to find *the* optimal ordering of input attributes. Though some functions are highly sensitive to this ordering, alternative orderings will still produce acceptable results in most cases [5]. This observation has been confirmed by our experiments in [22], where the models induced by the IFN algorithm from a set of benchmark datasets turned out to be nearly as accurate as the best known data mining models for those sets though IFN models contained less input attributes. A reasonably high predictive accuracy of IFN models is important if we intend to use them as "automated oracles" in regression testing, but expecting them to become "perfect predictors" of all outputs in complex software systems is certainly unrealistic. On the other hand, as shown in [21], the inherent compactness of these models can help us to recover the most dominant requirements from execution data and, consequently, to build a compact set of test cases. Another important property of the info-fuzzy algorithm is its stability with respect to training data [25], since as explained in the next section we can train it only on a small and randomly generated subset of possible input values.

3 INPUT-OUTPUT ANALYSIS WITH INFO-FUZZY NETWORKS

The architecture of the IFN-based environment for automated input-output analysis is shown in Figure 2. Random Tests Generator (RTG) obtains the list of system inputs and outputs along with their types (discrete, continuous, etc.) from System Specification. No information about the functional requirements is needed, since the IFN algorithm automatically reveals input-output relationships from randomly generated training cases. In our case study (see the next section), we explore the effect of the number of randomly generated training cases on the predictive accuracy of the IFN model. Systematic, non-random approaches to training set generation may also be considered.

The IFN algorithm is trained on inputs provided by RTG and outputs obtained from a legacy system by means of the Test Bed module. As indicated above, a separate IFN model is built for each output variable. The following information can be derived from each IFN model:

1. A set of input attributes relevant to the corresponding output.
2. Logical (if… then…) rules expressing the relationships between the selected input attributes and the corresponding output. The set of rules appearing at each terminal node represents the distribution of output values at that node (see [26]). These rules can be used to determine the predicted (most probable) value of the output in the corresponding test case.
3. Discretization intervals of each continuous input attribute included in the network. In testing terms, each interval represents an "equivalence class", since for all values of a given interval the output values conform to the same distribution.
4. A set of non-redundant test cases. The terminal nodes in the network are converted into test cases, each representing a non-redundant conjunction of input values / equivalence classes and the corresponding distribution of output values. Whenever the behavior of the tested application is characterized by some regular pattern, we expect the IFN-based number of test cases to be much smaller than the number of random cases used for training the network. This assumption is supported by the results of our case studies including the one presented in this paper.

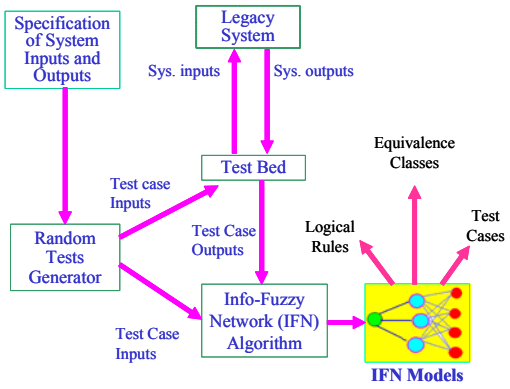

Figure 2 IFN-Based IO Analysis

A brief description of each module in the environment is provided below:

Legacy System (LS). This module represents a program, a component or a system to be tested in subsequent versions of the software. We assume here that this module is *data-driven*, i.e. it should have a well defined interface in terms of obtained inputs and returned outputs. Examples of data-driven software range from real-time controllers to business logic applications.

Specification of Application Inputs and Outputs (SAIO). Basic data on each input and output variable in the Legacy System interface 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 SAIO module (see above). The number of training cases to generate is determined by the user. The generated training cases are used by the Test Bed and the IFN modules.

Test Bed (TB). This module, sometimes called "test harness", feeds training cases generated by the RTG module to the Legacy System (LS). Execution of training cases can be performed by commercial test automation tools. The TB module obtains the LS outputs for each executed case and forwards them to the IFN module.

Info-Fuzzy Network Algorithm (IFN). The input to the IFN algorithm includes the training cases randomly generated by the RTG module and the outputs produced by the Legacy System for each test case. IFN also uses the descriptions of variables stored by the SAIO module. The IFN algorithm is run repeatedly to find a subset of input variables relevant to each output and the corresponding set of non-redundant test cases. Actual test cases are generated from the automatically detected equivalence classes by using an existing testing policy (e.g., one test for each side of every equivalence class).

In [21] we have applied the IFN algorithm to execution data of a small business application (Credit Approval), where the algorithm has chosen 165 representative test cases from a total of 11 million combinatorial tests. The Credit Approval application had 8 mostly discrete inputs and two outputs (one of them binary). It was based on well-defined business rules implemented in less than 300 lines of code. In the next section, we evaluate the proposed approach on a much more complex program, which, unlike Credit Approval, is characterized by real-valued inputs and outputs only.

4 CASE STUDY: FINITE ELEMENT SOLVER

4.1 The Finite Element Method

The finite element method was introduced in the late 1960's for solving problems in mechanical engineering [29][40] and quickly became a powerful tool for solving differential equations in mathematics, physics and engineering sciences [28][12][14]. The method consists of two stages. The first stage is finding a "functional", which is usually an integral that contains the input of the problem and an unknown function and is minimized by the solution of the differential equation. The existence of such a functional is a necessary condition for implementing the finite element method. The second stage is partitioning the domain over which the equation is solved into "elements", usually triangles. This process is called "triangulation" and it provides the "finite elements". Over each element the solution is approximated by a low degree polynomial (usually no more than fourth-order).

The coefficients of each polynomial are unknown but they are determined at the end of the minimization process. The union of the polynomials associated with all the finite elements provides an approximate solution to the problem. As in finite differences, a finer "finite element mesh" will provide a better approximation. The choice of quadratic or cubic polynomials generates a nonlinear approximation over each element and usually improves the accuracy of the approximate solution.

4.2 The Case Study

Consider solving the Laplace equation

$$\nabla^2 \phi = \phi_{xx} + \phi_{yy} = 0 \qquad (1)$$

over the unit square

$$D = \{(x, y) \mid 0 \le x \le 1, 0 \le y \le 1\} \qquad (2)$$

where the solution equals some given $f(x, y)$ on the square's boundary. This problem, i.e. finding $\phi(x, y)$ inside the square, has a unique solution. To find it we first define the functional

$$F = \iint_D (\phi_x^2 + \phi_y^2) dxdy \qquad (3)$$

where $\phi(x, y)$ is an arbitrary function which equals $f(x, y)$ on the square's boundary. This functional attains its minimum at $\phi = \phi_0$ where ϕ_0 is the exact solution. We now triangulate the square as shown in Figure 3.

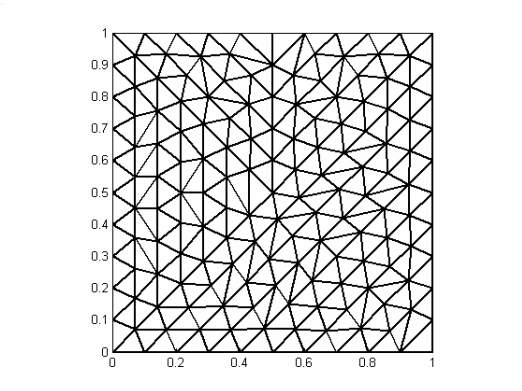

Figure 3 Triangulation of the domain

A low degree polynomial (say, linear) is chosen to approximate the solution over each triangle. The unknown coefficients of each polynomial are determined by minimizing the functional *F* (Eq. (3)) subject to the boundary conditions *f*:

$$\phi = f, \ on \ \ B \qquad (4)$$

4.3 The code

We applied an Unstructured Mesh Finite Element Solver (UMFES) which is a general finite element program for solving 2D elliptic partial differential equations (e.g., Laplace's equation) over an arbitrary bounded domain. UMFES was implemented in about **3,000 lines** of FORTRAN 77 code. The program consists of two parts. The *first part* obtains the domain over which the differential equation is solved as input, and triangulates it using fuzzy expert system's technique [13]. The input for this part is a sequence of points which defines the domain's boundary counterclockwise. It is assumed that the domain is simply connected, i.e. that it does not include holes. If it does, the user must first divide it into several simply connected subdomains and then provide their boundaries separately. The output of the code's first part is a sequence of triangles whose union is the original domain. The triangulation is carried out in such a way that two arbitrary triangles whose intersection is not an empty set, may have either one complete side in common or a single vertex. The approximate solution is calculated at the triangles' vertices which are called nodes. The nodes are numbered so that each two nodes whose numbers are close, are also geometrically close, 'as much as possible'. The fulfillment of this request guarantees that the coefficient matrix of the unknowns whose inverse is calculated at

the final stage of the numerical process, is such that its nonzero elements are located close to the main diagonal, which is advantageous for large matrices.

The triangulation rules which are based on a vast accumulation of knowledge and experience were chosen to minimize the process computational burden and to increase the accuracy of the approximate solution for a prefixed number of nodes. For example, given an initial domain, we first divide its boundary to small segments by extending the initial sequence of points. We have now obtained the boundary nodes and are ready to start the triangulation. Next, we are requested to find the best choice of a single secant that will divide the given domain into a couple of subdomains. The answer, which is a rule of thumb, is to draw a secant that will be completely within the domain, will provide two subdomains with 'similar' numbers of nodes on their boundary and will be "as short as possible". After drawing this 'optimal' secant, we treat each subdomain separately and repeat the process. The triangulation terminates when each subdomain is a triangle.

Other rules guarantee that each triangle will hardly ever possess an angle close to 0^0 or to180^0. This allows us to use fewer triangles and increases the solution's accuracy. Two types of triangulation are commonly used: (a) The triangles are 'close' to right-angled isosceles, and (b) The triangles are 'close' to equilateral. We may also adjust the density of the triangles within the domain. A triangulation with a variable density is demonstrated in Figure 4 where the density function is

$$\rho(x,y) = 1 + 2x^2 + 3y^2 \qquad (5)$$

Consequently, the triangulation is such that the least number of triangles per area unit is at (0, 0) and the largest occurs at (1, 1). The density function in Figure 3 is clearly $\rho(x,y) = const.$

The UMFES code can cope with any two-dimensional finite domain with a finite number of holes. Once the triangulation is completed, the code's *second part* applies the finite element method to solve any 2-D elliptic partial differential equation. The user now provides the input's second part which is the boundary conditions (B.C.), i.e. information about the solution over the domain's boundary.

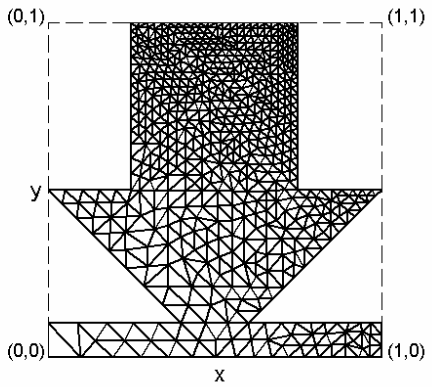

Figure 4 Triangulation Using a Variable Density Function

The code can treat 3 types of B.C. The first is Dirichlet type, where the solution ϕ is a specified function over some portion of the boundary. The second type is called homogeneous Neumann and states $\frac{\partial \phi}{\partial n} = 0$ over the second portion of the boundary. The third is a mixed type B.C., represented by the relation $\frac{\partial \phi}{\partial n} + \sigma\phi = \eta$ where σ and η are functions, given along a third portion of the boundary. The density of the finite elements is specified by the user and should be based on some preliminary knowledge of the solution's features such as the sub-domains where it varies slow or fast.

4.4 The Experiments

We have solved the case study of sub-section 4.2 replacing $f(x,y)$ by constants. Each problem was associated with Dirichlet B.C. defined by four constants (inputs), which specified the solution over the sides of the unit square. The four inputs were denoted as *Side1* ... *Side4* respectively. The five outputs were taken as the solution's values at the fixed points defined by the following coordinate-pairs:

$$(1/4,1/4),(3/4,1/4),(3/4,3/4),(1/4,3/4),(1/2,(1/2)$$

Accordingly, we referred to the outputs as *Out1*... *Out5*. The minimal distance from the boundary affects the predictive accuracy of the numerical solution. Thus, we can expect that the prediction for the last point (*Out5*) will be the least accurate one.

The B.C. conditions were taken randomly between -10 and +10, and we chose a sufficiently fine mesh of elements that guaranteed at least 1% accuracy at each output. Since the maximum and the minimum values of the solution are obtained on the boundary, the numerical values inside the domain must be between -10 and 10 just like the boundary (input) values. Thus, this typical problem was represented by a vector of 9 real-valued components: 4 inputs and 5 outputs.

Our first series of experiments was aimed at finding the minimal number of cases required to train the info-fuzzy network up to a reasonable accuracy. In this and all subsequent experiments we have partitioned the values of all target attributes into 20 intervals of equal frequency each, assuming this would be a sufficient precision for identifying the basic input-output relationships of the program. The algorithm was repeatedly trained on the number of cases varying between 50 and 5,000, producing five networks representing the five outputs in each run. All training cases were generated randomly in the input space of the problem. The predictions of every induced model have been compared to the actual output values in the training cases and in separate 500 validation cases using the Root Mean Square Error (RMSE). In Figure 5, we present the average RMSE over all five outputs as a function of the number of training records, while Figure 6 shows RMSE for each individual output.

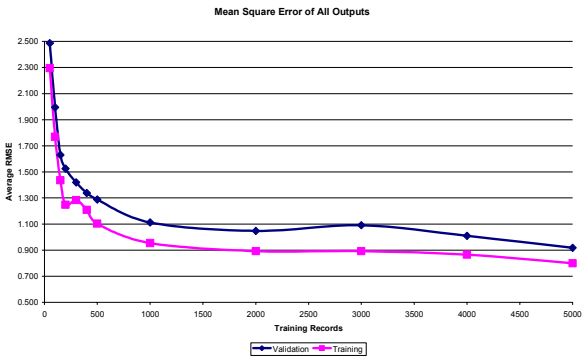

Figure 5 Average Mean Error over All Outputs

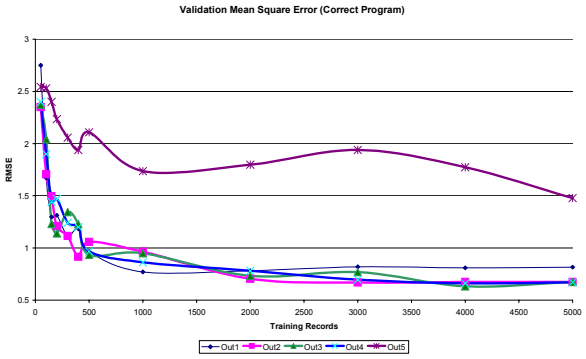

Figure 6 Mean Error for each Output

Both figures demonstrate a steep decrease in the mean error between 50 and 1,000 cases. Beyond 1,000 cases, the training error continues to decrease very slowly, while the validation error is relatively stable. These results suggest that **1,000 cases** only should be sufficient for training the info-fuzzy network on execution data of this program. Figure 6 confirms our earlier expectation that *Out5* behaves differently from the other outputs, which are closer to the boundaries. In fact, its mean error is almost two times higher than the error of any other output.

Table 1 shows the input attributes included in the model of every output after training the algorithm on 1,000 cases. The selected attributes are listed in the order of their appearance in the network. The same table also presents the size of each model (total number of nodes), the number of non-redundant test cases (number of terminal nodes), and the number of logical rules describing the induced input-output relationships. An immediate result is that the minimal test library based on the induced models would have only 386 test cases (one per each terminal node) vs. the original training set of 1,000 random cases. One can also see that only three input attributes out of four are sufficient for predicting the values of all outputs except for *Out5*, which is the hardest point to predict (see above). Also, the models are quite similar to each other in terms of number of nodes and number of rules. Consolidating the networks of multiple outputs into a single network is a subject of ongoing research.

Table 1 IFN Models Induced from 1,000 Cases

Output	Selected Input Attributes	Total Nodes	Terminal Nodes	Total Rules
Out1	Side4, Side1, Side2	79	67	264
Out2	Side2, Side1, Side4	86	74	276
Out3	Side3, Side2, Side4	90	77	275
Out4	Side3, Side4, Side1	102	91	279
Out5	Side1, Side3, Side4, Side2	102	77	318
Total		**459**	**386**	**1412**

The next step was to evaluate the capability of the IFN model to detect an error in a new, possibly faulty version of this program. We have generated four faulty versions of the original code, where the outputs were mutated *within the boundaries of the solution range* as follows:

1) error = 0.25*rand [-10, 10], where rand [-10, 10] is a random number in the range of the solution
2) error = 0.50*rand [-10, 10]
3) error = 0.75*rand [-10, 10]
4) rand [-10, 10]

Each faulty version was used to generate 500 validation cases. The validation RMSE of each model induced from 1,000 training cases is shown in Figure 7. We can see that even for the smallest injected error (1/4 of the solution range) there is a considerable difference of at least 30% between the RMSE of the outputs generated by the original program and the RMSE of the outputs generated by the faulty program. For all outputs, including the "hard to predict" output no. 5, the statistical significance level of this difference was found much higher than 0.001 (using the *F* test). These results show the effectiveness of IFN in discriminating between the correct and the faulty versions of tested software.

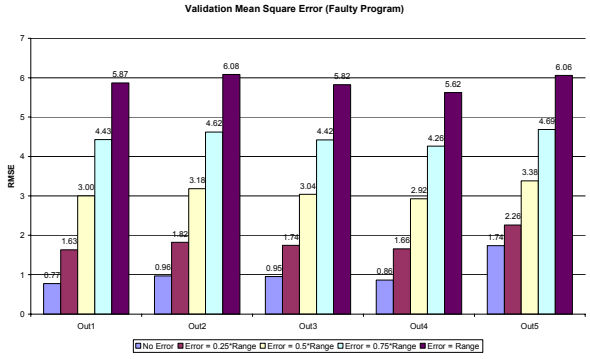

Figure 7 Mean Error for Faulty Programs

We have also studied the IFN capability to identify *individual cases* that produce incorrect output. In other words, we are interested to maximize the probability of catching an error (True Positive Rate), while minimizing the probability of mistaking correct outputs for erroneous (False Positive Rate). As a criterion for detecting an error, we have used the average of absolute differences between predicted and actual outputs. The ROC (Receiver-Operating-Characteristic) Curve for the four mutated versions of the original program is shown in Figure 8. We can see that for the smallest error (0.25 of the output range), we can get TP = 90% with FP slightly over 25%. However, for larger errors, the values of TP become very close to 100%, while keeping FP close to 0%.

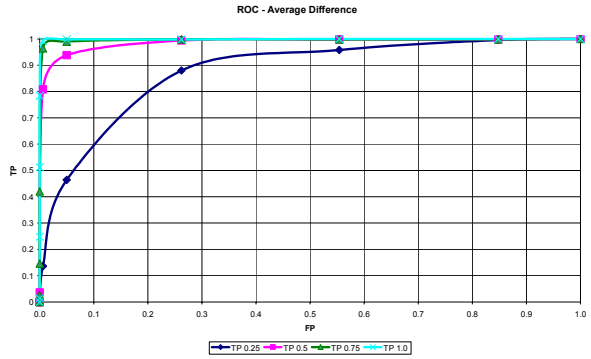

Figure 8 ROC Curve for Faulty Programs

5 SUMMARY AND CONCLUSIONS

In this research, we present and evaluate an emerging DM-based methodology for automated input-output analysis of data-driven software systems. As indicated in [36], such systems include embedded (real-time) applications, application program interfaces (API), and form-based web applications. The proposed method will address the following unsolved problems associated with regression testing of legacy systems:

- The method will automatically produce a set of non-redundant test cases covering the most common functional relationships existing in software (including the corresponding equivalence classes). Existing methods and tools for automated software testing do not provide this capability.
- The method will be applicable to testing complex software systems, since it does not depend on the analysis of system code like the white-box methods of automated test case selection.
- No significant human effort is required to use the method. Current methods and techniques of test case design assume manual analysis of either the requirements, or the code.
- Missing, outdated, or incomplete requirements are not an obstacle for the proposed methodology, since it learns the functional relationships automatically from the execution data itself. Current methods of test case generation assume existence of detailed requirements, which may be unavailable or incomplete in many legacy systems.

Issues for ongoing research include inducing a single model based on multiple outputs, test set generation and reduction, and application of the proposed methodology to large-scale software systems. Future experiments will also include evaluation of the method's capability to detect various types of errors injected in the code of the tested application.

Acknowledgement. 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.

6 REFERENCES

[1] Astra QuickTest from Mercury Interactive http://astratryandbuy.mercuryinteractive.com

[2] Beizer, B. Software Testing Techniques. 2nd Edition, Thomson, 1990.

[3] Blackburn, M.R., Busser, R.D., Fontaine, J.S. Automatic Generation of Test Vectors for SCR-Style Specifications. Proceedings of the 12th Annual Conference on Computer Assurance (Gaithersburg, Maryland, June 1997).

[4] Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, P.J. Classification and Regression Trees. Wadsworth, 1984.

[5] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, C-35-8, 677-691, 1986.

[6] Cover, T. M., and Thomas, J.A. Elements of Information Theory. Wiley, 1991.

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

[8] Dustin, E., Rashka, J., Paul, J. Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley, 1999.

[9] Elbaum, S., Malishevsky, A. G., Rothermel, G. Prioritizing Test Cases for Regression Testing. Proc. of ISSTA '00, 102-112, 2000.

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

[11] Fayyad, U., and Irani, K. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. Proc. Thirteenth Int'l Joint Conference on Artificial Intelligence (San Mateo, CA, 1993), 1022-1027.

[12] Friedman, M., Rosenfeld, Y., Rabinovitch A., and Thieberger, R. Finite Element Methods for Solving the Two Dimensional Schrödinger Equation. J. of Computational Physics, 26, 2, 1978.

[13] Friedman, M., Schneider, M., Kandel, A. FIDES – Fuzzy Intelligent Differential Equation Solver. Avignon, 1987.

[14] Friedman, M., Strauss, M., Amendt, P., London R.A., and Glinsky, M.E. Two-Dimensional Rayleigh Model For Bubble Evolution in Soft Tissue. Physics of Fluids, 14, 5, 2002.

[15] Hamlet, D. What Can We Learn by Testing a Program? Proc. of ISSTA 98, 50-52, 1998.

[16] Hildebrandt, R., Zeller, A. Simplifying Failure-Inducing Input. Proc. of ISSTA '00, 135-145, 2000.

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

[18] Kohavi R. Bottom-Up Induction of Oblivious Read-Once Decision Graphs. Proceedings of the ECML-94, European Conference on Machine Learning (Catania, Italy, April 6-8, 1994), 154-169.

[19] Kohavi R., and Li, C-H. Oblivious Decision Trees, Graphs, and Top-Down Pruning. Proc. of International Joint Conference on Artificial Intelligence (IJCAI), 1071-1077, 1995.

[20] Last M., and Kandel, A. Automated Quality Assurance of Continuous Data. NATO Advanced Research Workshop on Systematic Organisation of Information in Fuzzy Systems (Vila Real, Portugal, October 25-27), 2001.

[21] Last M., and Kandel, A. Automated Test Reduction Using an Info-Fuzzy Network. to appear in Annals of Software Engineering, Special Volume on Computational Intelligence in Software Engineering, 2003 .

[22] Last M., and Maimon, O. A Compact and Accurate Model for Classification. to appear in IEEE Transactions on Knowledge and Data Engineering, 2003.

[23] Last, M. Online Classification of Nonstationary Data Streams. Intelligent Data Analysis, 6, 2, 129-147, 2002.

[24] Last, M., Kandel, A., Maimon, O. Information-Theoretic Algorithm for Feature Selection. Pattern Recognition Letters, 22 (6-7), 799-811, 2001.

[25] Last, M., Maimon, O., Minkov, E. Improving Stability of Decision Trees. International Journal of Pattern Recognition and Artificial Intelligence, 16, 2, 145-159, 2002.

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

[27] Mayrhauser, A. von, Anderson, C.W., Chen, T., Mraz, R., Gideon, C.A. On the Promise of Neural Networks to Support Software Testing. In W. Pedrycz and J.F. Peters (eds.). Computational Intelligence in Software Engineering. World Scientific, 3-32, 1998.

[28] McDonald B.H., and Wexler, A. Finite Element Solution of Unbounded Field Problems. IEEE Transactions on Microwave Theory and Techniques, MTT-20, 12, 1972.

[29] Mikhlin, S.G. Variational Methods in Mathematical Physics. Oxford, Pergamon Press, 1965.

[30] Minium, E.W., Clarke, R.B., Coladarci, T. Elements of Statistical Reasoning. Wiley, New York, 1999.

[31] Nahamias, S. Production and Operations Analysis. 2nd ed., Irwin, 1993

[32] National Institute of Standards & Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3, May 2002.

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

[34] Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann, 1993.

[35] Rao, C.R., and Toutenburg, H. Linear Models: Least Squares and Alternatives. Springer-Verlag, 1995.

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

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

[38] Voas J. M., and McGraw, G. Software Fault Injection: Inoculating Programs against Errors. Wiley, 1998.

[39] Weyuker, E., Goradia, T., and Singh, A. Automatically Generating Test Data from a Boolean Specification. IEEE Transactions on Software Engineering, 20, 5, 353-363, 1994.

[40] Zienkiewicz, O.C. The Finite Element Method in Engineering Science. London, McGraw-Hill, 1971.