Electronics and Communications in Japan, Part 2, Vol. 90, No. 12, 2007
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J89-D, No. 8, August 2006, pp. 1673–1683

An Overview and Case Study of a Statistical Regression Testing Method for Software Maintenance

Tomohiko Takagi,[1] Zengo Furukawa,[2] and Toshinori Yamasaki[2]

[1]Graduate School of Engineering, Kagawa University, Takamatsu, 761-0396 Japan

[2]Faculty of Engineering, Kagawa University, Takamatsu, 761-0396 Japan

SUMMARY

We propose a statistical regression testing method for evaluating the reliability of software as part of the software maintenance process. Maintenance procedures take up more than half the time of the software development process; in addition, software reliability is an important factor in determining the dependability of a product. Regression tests are performed in order to conserve or improve software reliability as part of the software maintenance process. However, existing systematic testing methods based on regression tests are not necessarily appropriate for evaluating software reliability. The statistical regression testing method is a means for compensating for the flaws of such existing methods. In this method a model of how the user makes use of the software is defined by means of a Markov chain; this is known as the *usage model*, and then test cases are generated at random according to a probability distribution based on this usage model. In this paper we perform experiments applying the proposed method to a small-scale client server program and demonstrate that the proposed method can be implemented. In addition, we clarify the effects and issues that may be anticipated when applying the method and establish how it may be used in practice.
 Electron Comm Jpn Pt 2, 90(12): 23–34, 2007; Published online in Wiley InterScience (www.interscience.wiley.com). DOI 10.1002/ecjb.20414

Key words: statistical tests; regression tests; automated tests; Markov chains; software reliability evaluation.

1. Introduction

Software reliability is an important factor in determining the dependability of a product. This is not just true of standard software systems; in embedded systems, some 34% of failures are due to software failure [1]. Software testing methods are well known as a means for improving the software reliability and are widely used from the modeling process to the software maintenance process. In particular, software maintenance is a process that takes up approximately 65% of the whole time needed for software development [2] and demands continue to be made to deal with new failures found in a product after its release as well as to adapt to changes in the surrounding environment. In software maintenance, regression tests are indispensable as a procedure for preserving or improving reliability. Regression tests are mainly used to confirm that changes to software do not have any unforeseen side effects on the existing functionality. Since corrections to a piece of software can be performed in a variety of different ways, the type of failures that can arise during the software maintenance process are not restricted to a particular type in general. Consequently, the safest method is to re-conduct all the test cases that have been performed up to then but when this is difficult due to costs, the set of test cases that are applied

can be refined down by performing code dependency analysis and risk analysis [3].

Recently, the automation of regression tests using testing frameworks such as xUnit and other record/playback tools has become standard. By creating a set of test cases within the framework of a testing tool, the person in charge of the tests can reduce the associated work when maintenance is performed repeatedly (in other words when the same tests are repeated). It is common to design test cases by hand in a systematic method in order to cover source code and specifications according to specific test criteria. Coverage tests and partition tests are representative of the systematic methods. These methods have the advantage of focusing on locations that have a high probability of containing a hidden failure but cannot be said to always be appropriate as a means of evaluating software reliability [4]. This is because in general systematic testing methods make use of the number of remaining bugs in software as the measure of its reliability. From the user's perspective, however, the frequency with which a bug becomes apparent during actual usage is of more significance than the number of remaining bugs.

Therefore, in this paper we propose the introduction of statistical testing methods [5–7] as a means for alleviating the defects of such systematic methods in regression tests. We refer to the proposed method as *statistical regression testing*. Statistical testing methods are a type of random function testing whose main purpose is to evaluate software reliability considering the actual usage environment. Therefore, we define a model of how the user will use the software that we term the *usage model*; we define this model as a Markov chain and then generate test cases at random according to a probability distribution over this model. A result of this approach is that it allows the discovery of failures that will easily become apparent in the actual user environment (in other words those failures that have a relatively large impact on the software reliability) to be prioritized.

When generating test cases at random, while it is simple to test data, it is difficult to generate the precise expected output [8]. One approach to resolving this problem that has been considered [9] is to apply the test data that are generated at random to a version of the software that has already been released in order to generate the expected output. In this paper we refer to the software used in generating the expected output as the *base software*. Since the base software has been used by actual users for a certain period of time, the person in charge of the tests can obtain highly accurate information regarding the reliability of the base software. Consequently, it is possible to obtain reliable expected output from those functions of the base software that are known to be reliable. In addition, if the base software records an execution log, then this execution log can be used to construct an accurate usage model that reflects the manner in which users actually make use of the software. In this way the base software is useful when conducting statistical tests. Since base software usually exists when performing regression tests in the software maintenance process, we draw attention in this paper to the fact that the introduction of statistical tests is not particularly difficult and may be anticipated to be useful.

The objective of this paper is to demonstrate that the statistical regression testing method can be implemented and to clarify the anticipated effects and issues with using it as a means to establishing the method for use in practice. First, in Section 2 we present an overview of the statistical regression testing method. Next in Section 3 we describe the procedure used and results of experiments applying the method. Finally we conduct a discussion in Section 4.

2. The Statistical Regression Testing Method

In this section as an overview of the statistical regression testing method we describe the preconditions necessary for the application of this method and the procedure by which it is applied.

2.1. Preconditions for applying the method

In order to apply statistical regression tests the following five preconditions must be satisfied.

- Existence of base software

Base software is needed for the generation of expected outputs. In addition, in constructing the usage model, it is desirable that an execution log be obtained from the base software. Since it is anticipated that the statistical regression testing method will be used as part of the software maintenance process, it is probable that one or more versions of the software that could serve as the base software exist.

- Software output can be observed

The statistical regression testing method presupposes automation of the testing procedures. In order to automate the generation of test cases and to determine the outcome of tests automatically, a test tool must be able to acquire and interpret the outputs from the base software and the software under test.

- Knowledge regarding the reliability of the base software can be acquired

Reliable expected outputs can be obtained from those features of the base software that are known to be reliable. It is necessary to distinguish latent failures in the base software in order to avoid generating incorrect expected

outputs. If the base software has already been released and used for some time in its intended environment, then it should be simple to obtain highly accurate information regarding which features there are failures in.

- Functionality is not changed substantially

During the software maintenance process, it is difficult to obtain expected outputs from the base software for features that are changed in the external specifications. If necessary it is possible to create specific programs to test features that have been changed but this may involve significant costs. The greatest anticipated reductions in costs due to the use of the statistical regression testing method will be in those cases where changes are limited to internal specifications such as refactoring, tuning, or a change of platform.

- Systematic testing has been completed

Since the objective of the statistical regression testing method is to evaluate software reliability, it is normal to conduct it in the final stages of the software maintenance process. In addition, because the statistical regression testing method is not necessarily suitable for testing abnormal states and rare cases, it cannot be used as a replacement for coverage tests and partition tests.

2.2. Procedure

Figure 1 shows an overview of the statistical regression testing method in the form of a data flow diagram (DFD). The statistical regression testing method consists of the following steps (1) to (4).

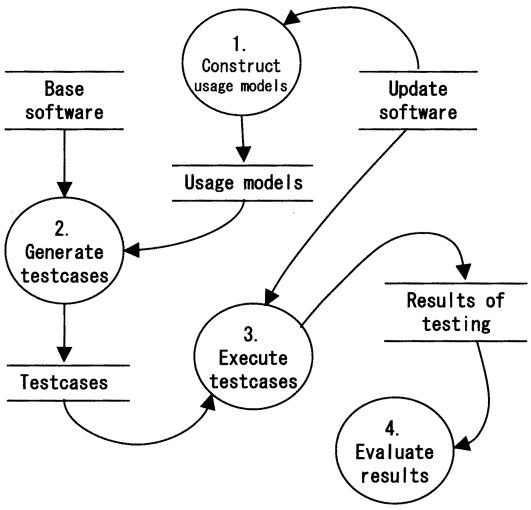


Fig. 1. Overview of statistical regression testing in a DFD.

(1) Constructing the usage model

The *usage model* is defined as a Markov chain based on the specifications for how the software is used. First a sequence of operations (including erroneous operations) on the software under test, the update software, is formulated as a state machine diagram [10]. In the state machine diagram we represent specific operations performed by the user as events and the steps of the behavior of the user (or the software) as states. We can treat input values for the operations as event parameters. Then we define all the transition probabilities and distributions of event parameter values on the state machine diagram. The methods we can use to solve these can be categorized into the following three types [7].

- Methods based on usage data from the intended environment. We can use an execution log for the program, raw data regarding the application domain, and work procedures. In order to accurately evaluate software reliability, this approach ought to be selected. In statistical regression testing we particularly recommend the use of an execution log from base software.
- Methods based on developers' predictions and expectations. We consider how a typical experienced user would operate the software.
- Assigning a uniform transition probability to all the outgoing transitions for each state. This method maximizes the entropy of the usage model.

In order to make the usage model easily reusable we can convert all the transition probabilities into constraints for simultaneous equations or for mathematical programming [11]. For example, we can represent the fact that *the transition probability P_a for transition a is twice the transition probability P_b for transition b* as $P_a - 2P_b = 0$. However, in the software maintenance process it is necessary to investigate how changes to functionality will influence the existing usage model. In general there will be a limited effect on the usage model [12] and it will not often be necessary to perform widespread corrections to it.

(2) Generation of test cases

The test case is the sequence of pairs of test data for the update software and associated expected output that correspond to a path from an initial pseudo state to a final pseudo state over the usage model. The test data consist of an event and event parameter value(s) that are generated stochastically from the usage model. If multiple usage models are being operated in parallel, we interleave the test data that each of them generate respectively (i.e., we fuse them into a single sequence). The expected output, on the

other hand, are generated by applying the test data to the base software. We prepare enough test cases to satisfy the stopping criterion for the statistical regression testing assuming no failures are detected in the update software (we describe the stopping criterion in more detail below).

If there is a difference in the external specifications of the update software and the base software, then a special procedure is needed in order to generate test cases. Here we let O_{updt} and O_{base} be the set of operations that can be performed on the update software and the base software, respectively. If $(O_{base} - O_{updt}) \cup (O_{updt} - O_{base}) \neq \phi$ then when generating expected outputs we can either ignore the operations that correspond to these or analyze them by replacing them with a minimal set of operations that connect to the nearest $O_{base} \cap O_{updt}$. In addition, if we let F_{updt} and F_{base} be the sets of functions of the external specifications of the update software and the base software, respectively, then we cannot obtain expected outputs from the base software for those relating to $F_{updt} - F_{base}$. In order to deal with this situation the following main four methods exist.

- Create a program to generate the expected outputs. It is important that this program uses a different algorithm to the update software.
- Create a program to verify the constraints. Here constraints refer to universal conditions that the update software must satisfy and that can be defined in the object constraint language (OCL [10]).
- Assume that there are no failures and do not verify the outputs of the update software.
- Have the expected output be designed by hand by the person in charge of the tests. Since the number of test cases will be extremely large, this will frequently be difficult from a cost perspective.

On the other hand, we must eliminate those outputs that are derived from $F_{base} - F_{updt}$ from the expected outputs.

(3) Executing the test cases

We apply the test cases to the update software. Since the number of test cases is extremely large, it is important that this step is completely automated using a test tool. We process all the test cases or proceed to the following step at the point where a failure is detected.

(4) Evaluation of test results

Finally we create a test model and evaluate the test results. The test model is a Markov chain that is defined by mapping the execution results of the test cases onto the state machine diagram created in (1). If failures were found as a result of executing the test cases, then we add a failure state that represents that a failure has been detected to the test model, and we create a transition from the state that caused the failure to the failure state. In addition, if the failure was minor, we create a transition to the originating state, while if the failure was fatal, we create a transition to the final pseudo state. If no failure is detected, then the test model is approximately equivalent to the usage model according to the law of large numbers. The objective of creating the test model is not only to assist the understanding of the person in charge of the tests by abstracting and visualizing the execution results of the test cases but also to enable the evaluation of the stopping criterion for the statistical regression testing.

Measures used for stopping criterion [6, 13] may be broadly categorized into the following two types.

- Software reliability

The following measures have been proposed: the average number of transitions between failure states, the number of failures detected per test case, the distance between the usage model and the test model according to the Kullback discriminant.

- Coverage

Various coverage criteria such as covering all states in the usage model, covering all transitions, or other path coverage criteria have been considered. These are intended to complement measures of software reliability.

Since no universal thresholds exist for these measures, it is necessary to investigate such parameters for each project in turn based on user demands and industry or corporate criteria.

If no failures are detected, then fundamentally the stopping criterion can be achieved. The fact that the stopping criterion is reached will be one piece of evidence that the update software is at a level where it is ready for shipping. If failures were detected, we perform debugging and the associated systematic tests and then return to stage (3). If the debugging applies to a large scope, then we may need to start again from stage (1). In general, when the software is not particularly reliable, it will often not be possible simply to make do with local amendments to the program; in some cases it may even be necessary to revise the specifications. Consequently, the software reliability will aid in determining the subsequent iteration process.

3. A Case Study

We performed experiments applying the statistical regression testing method that we have proposed in the previous section in order to confirm that it actually functions in practice. In this section we describe this process and the results.

3.1. Clarifying the objectives of the applied experiments

The objective of the statistical regression testing method is to evaluate the software reliability. Therefore, first we will clarify the factors on which software reliability depends.

In the statistical regression testing method, the software reliability is determined by the frequency with which functionality that contains failures is used. This approach belongs to what is known as the *data domain model* [14] in the field of software reliability engineering. In this case the accuracy of the software reliability measurement will depend on the following two points.

- Usage model accuracy; in other words, whether or not the usage model actually reflects the manner in which the update software is used in practice.
- Failure detection accuracy; in other words, to what extent the test tool can detect failures from the output of the update software.

Regarding the accuracy of the usage model, it is possible to obtain suitable results by constructing the usage model on the basis of data obtained from the actual usage environment (see Section 2.2); in particular, there are a number of automated methods that have been proposed previously that make use of a program execution log. For example, Takagi and Furukawa [15] present a method whereby a probe is inserted to record the usage history when the source code is generated from software specifications. In addition, Shukla and colleagues [16] inserted a probe into the Java API in order to construct a usage model at the component level. Kallepalli and Tian [17] constructed a usage model for a Web system by using the access log of a Web server. Using one of these methods, it is theoretically possible to obtain a highly accurate usage model with low manual costs. This is why in this paper we recommend obtaining an execution log from the base software. Since an execution log is the most objective and direct means, it is difficult to evaluate adequacy of a usage model in terms of accuracy to a greater level than this will provide. Therefore, we consider that issues relating to the accuracy of the usage model are already resolved and do not focus on them specifically in these experiments.

From the above discussion it follows that in order to confirm that the statistical regression testing method is functioning correctly, it suffices to clarify the accuracy with which failures are detected. To put this another way, we must confirm that the expected outputs that were automatically generated are effective in detecting failures. In addition, we must estimate the required costs for conducting the statistical regression testing if we are to fully investigate its potential for being realized in practice.

3.2. Overview of the applied experiments

First we create the base software and then we create the update software by changing the features of the former and adding failures to it. Next we create a usage model based on the software specifications and the authors' predictions and we assume that this accurately reflects the usage environment. After this we construct a test tool that will automate the execution of the statistical regression testing and then attempt to detect the failures that we have added to the software. Since in this way we can obtain the failure detection rate (the proportion of failures that were detected to all the failures), we can investigate the reason why we failed to detect a failure in those cases where a failure was not detected. From the above discussion we clarify the failure detection precision. If the test cases generated on the basis of the assumed usage model are able to detect all the latent failures in the functionality that we actually executed, then we can conclude that *given this usage model, we have implemented a highly accurate reliability evaluation framework*. In addition, by constructing a test tool we can obtain an estimate of the amount of development required to create it (the number of lines of source code) and the time needed for testing (the time required to generate and execute the test cases).

In the following we first describe the creation of the base software and update software in Sections 3.3 and 3.4. These correspond to the prior conditions of these applied experiments. Then in Section 3.5 we explain the procedure by which the statistical regression tests are executed and the results step by step. A discussion of these is then conducted in Section 4.

3.3. Receptionist system

We decided to use the problem of a receptionist system for a drinks wholesaler as the subject for our applied experiments. A written description of the problem is given in Fig. 2. This problem is a slight variation on the inventory control system proposed in Ref. 18.

In this paper we implement the receptionist system as a small-scale client server program with approximately 4000 lines of source code. We used the Java programming language [19] for the implementation. In order to make the problem general in these experiments, we did not have the client and server communicate via a physical network but rather had them run in a standalone environment. The runtime environment was PC (CPU: Pentium III 933 MHz; RAM: 504 MB) running Windows XP.

Prior to the experiments we first prepared two versions of the receptionist system.

- Old version receptionist system

Every day the warehouse of a certain drinks wholesaler receives deliveries of containers that contain bottled alcoholic beverages. A single container may contain multiple brands as long as the volume of the container is not surpassed. The wholesaler deals in approximately 100 different drink brands. The warehouse employee receives containers and stores them in the warehouse as they arrive and passes the shipment form for the container to the receptionist. When an order to ship is received from the receptionist, the oldest product will be shipped first. We assume that the goods in the containers are never repackaged or moved for storage in a different place; containers that are empty are immediately taken out of the warehouse and no damage occurs to the beverages either while being transported or in storage.

Every day the receptionist receives some 10 or so delivery requests from retail stores that have contracts with the wholesaler. Each time he receives the request, he issues a delivery order form to the warehouse employee. We assume that all delivery requests are made by delivery request form and that each such request is limited to a single brand. If the amount stored in the warehouse is insufficient, the retail store that made the request will be contacted. At the same time a stock replenishment request form is filled out and an order for the relevant product is placed with the beverage producer. Then once the levels of the product in the warehouse have been replenished, a delivery order is passed to the warehouse employee. In addition, information regarding containers that are anticipated to become empty is provided.

The slips used by the receptionist consist of the following contents:

- Shipment form: container ID, arrival day, list of {order number, product name, quantity}
- Delivery request form: order number, time and date of order, product name, quantity, destination name
- Delivery order form: order number, product name, destination name, list of {container ID, quantity}
- Stock replenishment request form: order number, time and date of order, product name, quantity

We are required to create a computer program that will automate the work of the receptionist (a receptionist system). Since this task has some unrealistic parts to it, we omit a processing of the input data for errors.

Fig. 2. The receptionist system problem for a drinks wholesaler (reproduced with slight changes from Ref. 18).

This initial version of the software was given the minimum necessary functionality to fulfill the task specifications; we used this software as the base software. Here we assumed that all functions had reached a sufficient level of reliability due to their being used over a long period of time.

- New version receptionist system

This consisted of the old version of the receptionist system to which a new feature was added. The new feature consisted of the ability to inform the warehouse employee when a container is expected to become empty; this was implemented by adding a *dispatch empty container mark* to the delivery order form.

3.4. Addition of failures

It was necessary to also intentionally add failures to the new version receptionist system. Here based on mutation analysis [20] we decided to use the five mutation types enumerated below. In this way we were able to mechanically generate various different bugs (failures that are expressed in various aspects) that could arise during the software maintenance process.

- Relational operator replacement (ROR)

This mutation type involves replacing relational operators. In these experiments we limited these to inclusion bugs such as writing >= instead of >.

- Arithmetic operator replacement (AOR)

This mutation type involves replacing one arithmetic operator with another one.

- Logical operator replacement (LOR)

This mutation type involves replacing one logical operator with another one.

- Constant replacement (CRP)

This mutation type involves replacing an integer type constant with another. For instance, given the constant C, we change it to $C - 1$, 0, or $C + 1$.

- Statement deletion (SDL)

This mutation type involves the deletion of arbitrary non-declare statements.

Based on the above we changed the new version receptionist system program in one location. In general we refer to the amended program as the *mutant program* and the amended statement as the *mutant*. In these applied experiments we created a simple system that assists in the generation of the mutant. However, it was necessary to avoid obtaining mutants that were equivalent to the original

program or to other mutant programs. We created multiple mutant programs and took a total of 100 of these that did not result in compiler errors (20 programs for each mutation type) and used these as the update software for testing.

3.5. Applying the statistical regression tests

We applied the statistical regression testing method to the new version receptionist system (in other words to each of the 100 mutant programs). In this section we describe this process and the results.

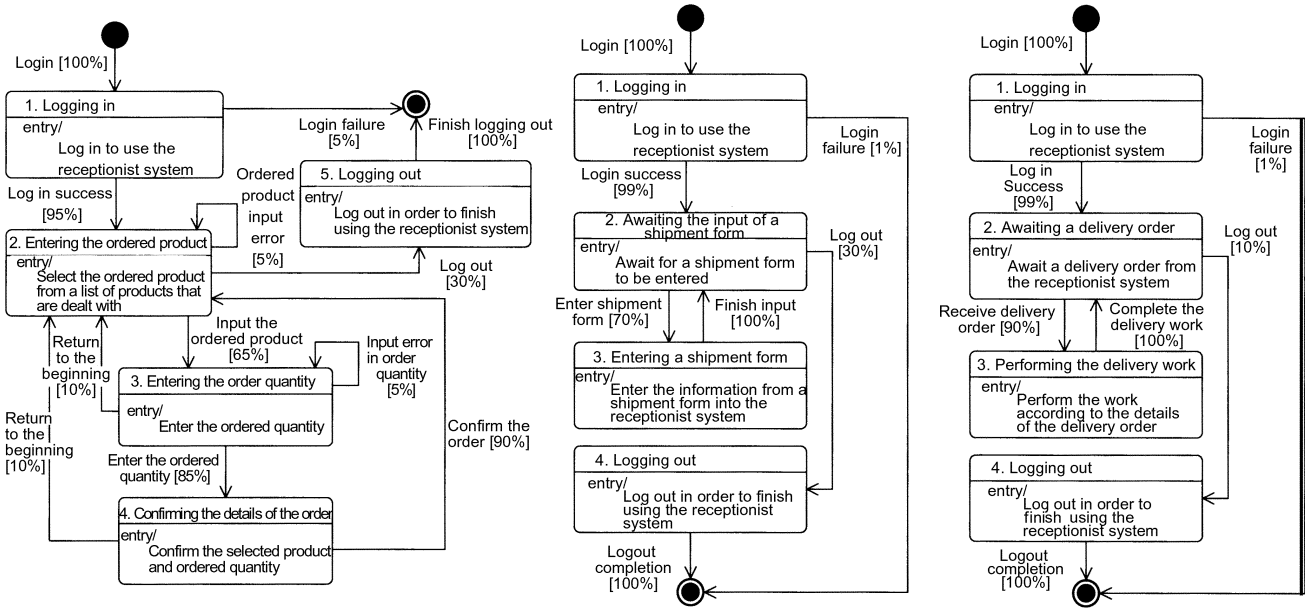
(1) Constructing the usage model

The users of the receptionist system are the retail stores and the warehouse employees. The warehouse employees can be divided into stock receipt employees and order shipment employees depending on the type of work they are assigned. Figure 3 shows usage models created for each of these three different types of user. The solid black circles in this figure represent initial pseudo states and the solid black circles surrounded by circles are final pseudo states. The entry/domain in each state box represents the entry actions appropriate for the state. All the transitions are labeled with event names and transition probabilities. The transition probabilities and the distributions for event parameter values are determined on the basis of predictions; in these experiments we assume that these accurately reflect the usage environment. We note that all the usage models that we have created operate in parallel.

(2) Generating test cases

Based on Fig. 3, we created a program in Java that generates test cases (the *test case generator*). This program consisted of approximately 1000 lines of source code. The test case generator applied to the old version receptionist system interleaving test data that were generated by each of the usage models in parallel. Since the old version and new ones display the main behavior using the standard output, we take the contents of the standard output that results from test data applied to the system to be the corresponding expected output. We set the test case generator so as to guarantee 1.5 seconds of waiting time following the application of the test data that has the contest of standard output in order to be sure of obtaining such contents. Regarding the stopping conditions we took a measure that resembles the average number of transitions between failure states and stopped the testing procedure when the number of transitions that could be taken from starting the testing without a failure occurring exceeded 10,000.

Following the above procedures we created 871 test cases that consisted of a text file of approximately 3 MB. These took approximately 210 minutes to generate. We present an example of a test case in Fig. 4. There are two pairs of test data and expected outputs in Fig. 4. The first one consists of the input from a retail store for an order quantity of 101 and it shows that the client program performs confirmation of the order details. Then the second one consists of the input from the retail store performing confirmation of the order and shows that the client program



(a) Retail store usage model (b) Stock receipt employee usage model (c) Order shipment employee usage model

Fig. 3. Usage models for the receptionist system.

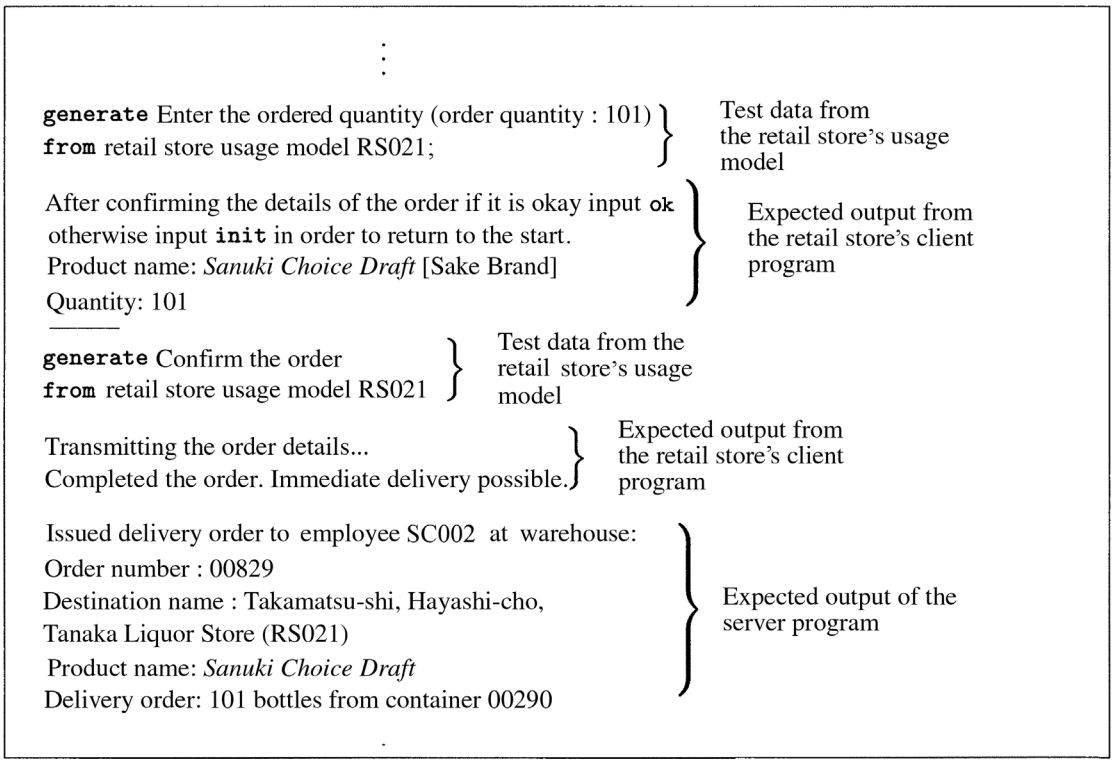


Fig. 4. Example test cases.

provides notification that the order has been completed and at the same time the server program makes a delivery order to the warehouse employee on the basis of the order details.

(3) Executing the test cases

We created a program in Java (the *test driver*) to interpret and execute the test cases that have been generated. This consisted of approximately 1000 lines of source code. The test driver applies the test data in the test case to the new version receptionist system. Then it obtains the contents of the standard output of the new version receptionist system in real time and compares this with the corresponding expected output in the test case. In addition, it is possible to generate the expected output of new functionality (here the empty container notification feature) by implementing it in the test driver using an algorithm different from that used in the update software. The test driver judges that a failure has been detected when either the contents of the standard output do not match the expected output or there has been no response within a fixed period (in these experiments we took this to be 10 seconds).

(4) Evaluating the test results

The test driver generates the test results as a comma separated value file (CSV). The test results include information such as the number of transitions taken and the failures detected. For example, Fig. 5 shows the progress of tests for a given piece of update software edited using a graphics tool. The figure shows that a failure was discovered as the 302nd test case was executed (as the 3308th transition was executed). Since this failure occurred in

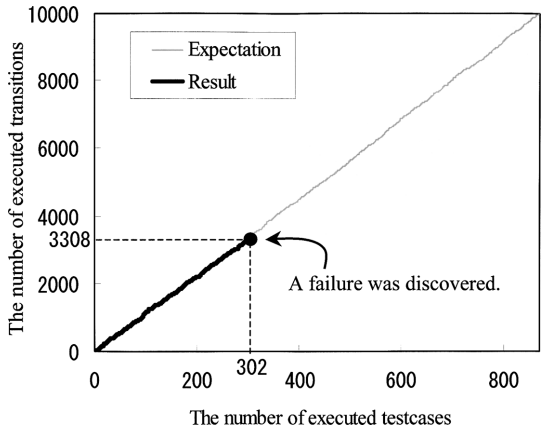


Fig. 5. Test progress described in terms of the relationship between the number of test cases executed and the number of transitions taken.

Table 1. Summary of a detected failure

Test data	Generation of the confirm the order event by the retail store's usage model.
Expected output	Currently stocks of the ordered product are insufficient. However, a delivery is expected in the near future so do not create a stock replenishment request form.
Test output	Because stocks of the ordered product are currently insufficient, create a new stock replenishment request form.
Details of the failure	The new version receptionist system may result in excess stock being procured since it is unable to recognize that a delivery is already planned.

response to the 222nd test case generated by the retail store's usage model, we can evaluate this in terms of deployment time in the usage environment by letting a be the average number of accesses from the retail store to the server (i.e., accesses/time unit) and then take the time to the failure to be $222/a$ (time unit). The time taken to detect the failure was 6 minutes and 36 seconds. An overview of the details of the failure that was detected is given in Table 1.

We were successful in discovering the failures in 94% of all the pieces of update software. The detection rate broken down by mutation type was as follows: ROR : 95%, AOR : 100%, LOR : 100%, CRP : 100%, SDL : 75%. In addition, the coverage once all the test cases had been exhausted was 79% according to the C_0 criterion and we covered large portions of the normal system. The details of the failures that we were unable to detect are as follows:

- Failures in rare cases; for example, failures that would only be apparent if the quantity of a product ordered by the retail store matched the amount held by the wholesaler exactly.
- Failures in abnormal systems; for example, failures would only become apparent when network communications were cut due to an external factor.

In all cases in which failures went undetected, this was due to their not being executed by the test cases. Therefore, we have confirmed that all failures that resulted in some form of change in the output contents when a test case was executed could be detected by the test tool.

The average amount of time required for the execution of test case before detecting a failure was 21 seconds; in addition, the time taken to exhaust all test cases when a failure could not be detected was approximately 27 minutes. We were able to detect a failure in 93% of the pieces of update software within one minute.

4. Discussion

4.1. Failure detection precision

The scale of the software used in these experiments was small and a large number of the failures that were inserted were quite elementary but we were able to automatically detect 94% of the failures. The reason for missing the remaining 6% was that we were unable to generate test cases that executed rare cases or abnormal system states. Therefore, we may conclude that since we were able to detect all the failures in those functions that were executed, given the assumption of the usage model in Fig. 3, we have created a framework that allows a highly accurate reliability evaluation.

If the rare cases and abnormal system states can be distinguished in advance, then it is possible to provide coverage for them by adding them to the usage model. The example of an abnormal state given in Section 3.5 (4) could for instance be covered by considering the event that the program is forcefully terminated as it is establishing communications. However, since in general the occurrence of rare cases and abnormal states in the usage environment is low, their appearance in test cases will also be low and we cannot expect to sufficiently test them. In a random testing method Beizer [8] stated that in order to test such functionality, test cases should be created in inverse proportion to the usage probability. In addition, in existing research into statistical testing methods a method has been proposed for getting coverage of the complete program; this involves generating test cases while altering the probability distribution of the usage model [21]. However, since in such an approach the usage model will not reflect the usage environment, it is difficult to use it to evaluate the software reliability in the usage environment. To avoid this problem Walton and colleagues [7] presented an approach that constructed a specific usage model for a set consisting only of functions that are used infrequently but that require testing. However, even using any of these approaches as long as we continue to generate test cases at random, there is no guarantee that we will obtain test cases that are effective for detecting specific failures. Consequently, it is difficult to both provide comprehensive coverage in the detection of failures and at the same time provide a complete evaluation of the software reliability. Regarding this point, we may conclude that statistical testing methods and systematic testing methods are mutually complementary.

4.2. Time taken for testing

In these applied experiments the time required to generate the test cases was more than 7.8 times the time taken to execute the test cases; we therefore saw that the

generation of the test cases required a large amount of time. The reason for this was that in order to ensure that we obtained the output from the base software when the test case generator applied test data to it, we set the waiting time to a long interval. If some sort of return code could be used to signify that the output has completed, it would enable us to shorten the time taken. The time taken for testing can be estimated from the number of transitions n required to satisfy the stopping criterion and the average time taken to generate (execute) one transition t. Assuming that during the software maintenance process there is no large variation in t, then the testing time is proportional to n. In these applied experiments the time taken to generate the test cases was $1.26n$ seconds and the time taken to execute them was $0.16n$ seconds. As is clear from above, the time taken for testing is determined more by the required level of reliability, the structure of the usage model, and the time required to wait for output than the scale of the update software as such. As a result of conducting multiple experiments applying statistical testing methods in the automobile, aeronautics, and communications fields, Guen and Thelin [22] concluded that the time taken for testing depends on the complexity and field to which software is to be applied; the position presented in this paper does not contradict theirs. We note that even if a long testing time is required, then since fundamentally the test tool functions automatically, the testing can be scheduled for evenings and nonwork days so as not to impede other development work.

It is difficult to make a general comparison between statistical regression testing methods and standard systematic testing methods from the perspective of cost effectiveness. It is not after all the objective of this paper to argue the pros and cons of these approaches to reach a conclusive preference. This is because it is important to select the appropriate testing method for the given testing objectives and development conditions; in addition, results from experience suggest that it is effective to make use of various testing methods according to Ref. 23.

4.3. Test tool creation costs

In the applied experiments we created two test tools in order to automate the statistical regression testing method: a test case generator and a test driver. The structures of these programs resemble one another and we foresee that large portions of them can be reused in the form of libraries and code patterns. However, since the total amount of source code was approximately 2000 lines, it could appear that an excessive cost was associated with the creation of these test tools. Therefore in future work we intend to construct a model compiler [24] to generate the test tools from the usage model. A model compiler is a computer aided software engineering (CASE) tool that supports Executable UML; it can create a complete program from the software specifications. The following three requirements are necessary for a model compiler to be able to support the statistical regression testing method:

- It must support an action language that can formally describe the operations of the usage model;
- It must have code generation rules that can generate the test case generator and test driver from the usage model;
- It must have functionality that allows the usage model and test model to be displayed diagrammatically and to analyze data from the tests.

We note that the test support tool JUMBL [25] can be used for statistical regression testing. This consists of a collection of Java class libraries and command line programs. The person in charge of testing can generate an executable test script by using JUMBL to mark up test commands within the usage model. Full-scale applied experiments are apparently yet to be completed but a benefit of this tool is that it is flexible to use having no dependency on the applied field or the development environment.

The model compiler-based approach differs most from JUMBL in that it aims to completely generate the test tool by treating the usage model as a platform independent model (PIM) within a model-driven architecture (MDA) and then using code generation rules specialized to this application field and execution environment. In this way we anticipate a reduction in the amount of effort required to create the test tools.

4.4. Debugging support

DMET [26] proposed by Matsushita and colleagues is one debugging supporting method that could be applied to statistical regression testing. According to DMET, a failure found by the statistical regression testing method implies that there is a high probability that the cause (the bug) exists in the delta source code between the base software and the update software. Therefore, by selectively investigating the delta source code, we may anticipate a reduction in the amount of time taken to discover a bug. If we can also test against software versions that fall between the base software and the update software, then it may be possible to further constrain the set of questionable source code to be checked.

4.5. Possibility of applications to embedded systems

The statistical regression testing method cannot necessarily be applied to embedded software. The reason for this is that it is difficult to obtain an execution log from the base software on the actual device. Considering the results

of applied experiments in Ref. 22, since there is a tendency for embedded software to have a large number of states and transitions in the usage model per line of source code, it may be difficult both from a cost and an accuracy perspective to construct the usage model manually. In addition, while it is possible to generate and execute the test cases on a simulator that runs on a PC, it is difficult to do this on the actual devices.

5. Conclusions

We have proposed a statistical regression test method that can be used both to evaluate the software reliability and to detect failures that will have a relatively large influence on the software reliability during the maintenance process. This method requires a large number of test cases that reflect the usage characteristics of actual users; however, given the usage model and the base software, an unlimited number of test cases can then be generated. The procedure from the generation of test cases to the generation of test results is all performed automatically using a dedicated test tool. Finally, by assessing the test results the person in charge of the tests is not only able to confirm which failures were detected but also to use the results to help determine the next round of the development process and also to use them as an indicator that the software has reached a level suitable for shipping.

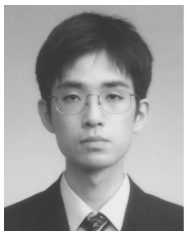
In results of the experiments applying the statistical regression test method to small-scale software in this paper confirmed we were able to evaluate the software reliability and to discover failures in a realistic amount of time. In future work it will be necessary to conduct experiments applying the method to larger scale and more complex software at an actual development site over a certain period of time and to investigate whether or not a significant reduction in the number of failures reported by actual users of the finished product is seen in order to verify the efficacy of the method. In addition, since we have seen that the amount of time needed to create test tools is by no means negligible, we intend to create a model compiler for generating test tools from the usage model.

REFERENCES

1. Ministry of Economy Trade and Industry, Commerce Information Policy Department, Information Policy Unit, Information Processing Promotion Division, 2005 Annual Embedded Software Industry Current Conditions Survey Report. (in Japanese)
2. CASE 1988–89, Sentry Market Research, p 13–14, Westborough, MA, 1989.
3. Binder RV. Testing object-oriented systems: Models patterns and tools. Addison–Wesley; 1999.
4. Hamlet D, Taylor R. Partition testing does not inspire confidence. IEEE Trans Softw Eng 1990;16:1402–1411.
5. Whittaker JA, Poore JH. Markov analysis of software specifications. ACM Trans Softw Eng Methodol 1993;2:93–106.
6. Whittaker JA, Thomason MG. A Markov chain model for statistical software testing. IEEE Trans Softw Eng 1994;20:812–824.
7. Walton GH, Poore JH, Trammell CJ. Statistical testing of software based on a usage model. Softw Pract Exp 1995;25:97–108.
8. Beizer B. Software testing techniques, 2nd ed. Van Nostrand Reinhold; 1990.
9. Takagi T, Furukawa Z, Yamasaki T. Generating expected output for statistical regression testing. Tech Rep Inf Process Soc Japan 2005, No. 75, p 97–102. (in Japanese)
10. Object Management Group, Unified Modeling Language, http://www.uml.org/
11. Poore JH, Walton GH, Whittaker JA. A constraint-based approach to the representation of software usage models. Inf Softw Technol 2000;42:825–833.
12. Musa JD. Operational profiles in software reliability engineering. IEEE Softw 1993;10:14–32.
13. Sayre K, Poore JH. Stopping criteria for statistical testing. Inf Softw Technol 2000;42:851–857.
14. Yamada S. Software reliability models. Nikka Giren; 1994. (in Japanese)
15. Takagi T, Furukawa Z. Constructing a usage model for statistical testing with source code generation methods. Proc 11th Asia-Pacific Software Engineering Conference, p 448–454, Busan, Korea, 2004.
16. Shukla R, Carrington D, Strooper P. Systematic operational profile development for software components. Proc 11th Asia-Pacific Software Engineering Conference, p 528–537, Busan, Korea, 2004.
17. Kallepalli C, Tian J. Usage measurement for statistical web testing and reliability analysis. Proc 7th International Software Metrics Symposium, p 148–158, London, 2001.
18. Nimura Y, Amamiya M, Yamazaki T, Fuchi K. Designs for common problems based on a new programming paradigm. Mag Inf Process Soc Japan 1985;26:458–459. (in Japanese)
19. Sun Microsystems. Sun Developer Network: Java 2 Platform Standard Edition, http://java.sun.com/
20. DeMillo RA, Offutt AJ. Constraint-based automatic test data generation. IEEE Trans Softw Eng 1991;17:900–910.
21. Fosse PT, Waeselynck H. STATEMATE applied to statistical software testing. Proc International Sym-

posium on Software Testing and Analysis, p 99–109, 1993.
22. Guen HL, Thelin T. Practical experiences with statistical usage testing. Proc 11th Annual International Workshop on Software Technology and Engineering Practice, p 87–93, 2003.
23. Kaner C, Bach J, Pettichord B. Lessons learned in software testing. John Wiley & Sons; 2002.
24. Mellor SJ, Balcer MJ. Executable UML: A foundation for model-driven architecture. Addison–Wesley; 2002.
25. Prowell SJ. JUMBL: A tool for model-based statistical testing. Proc 36th Hawaii International Conference on System Sciences, p 337c, 2003.
26. Matsushita M, Teraguchi M, Inoue K. Program-delta oriented debugging supporting method DMET. Trans IEICE 2004;J87-D-I:815–823. (in Japanese)

AUTHORS (from left to right)



Tomohiko Takagi (student member) graduated from the Faculty of Engineering at Kagawa University in 2002, completed the master's program at the Graduate School of Engineering in 2004, and enrolled in the advanced doctoral program. His research interests are in software engineering, particularly software testing methods. He is a member of the Information Processing Society of Japan.

Zengo Furukawa (member) graduated from the Faculty of Engineering at Kyushu University in 1975, completed the master's program at the Graduate School of Engineering in 1977, and joined the Hitachi Systems Research and Development Center. In 1986 he became a research associate at Kyushu University, a lecturer in 1990, and an associate professor at the Information Processing Education Center in 1992. Since 1998 he has been a professor in the Faculty of Engineering at Kagawa University. His research interests are in software engineering, in particular, software testing methods, distributed systems, and the management of Internet systems. He holds a D.Eng. degree, and is a member of the Information Processing Society of Japan, Japan Society for Software Science and Technology, ACM, IEEE-CS, and ISOC.

Toshinori Yamasaki (member) graduated from the Faculty of Engineering at Osaka University in 1966, completed the master's program at the Graduate School of Engineering in 1968, and became a research associate at Kansai University. In 1970 he became a research associate in the Faculty of Education at Kagawa University and subsequently a professor. He was concurrently the Head Master of the adjunct Takamatsu Junior High School. In 1993 he was a visiting researcher at the Information Science Laboratory of Berne University, Switzerland. Since 1998 he has been a professor in the Faculty of Engineering at Kagawa University. His research interests are in educational engineering, pattern measurement, and recognition. He holds a D.Eng. degree, and is a member of the Information Processing Society of Japan, Society for Information and Systems in Education, Physical Society of Japan, and Society for Systems Control and Information.