Sept. 2006, Vol.21, No.5, pp.697–707 J. Comput. Sci. & Technol.

# Software Reliability Experimentation and Control

Kai-Yuan Cai (蔡开元)

*Department of Automatic Control, Beijing University of Aeronautics and Astronautics, Beijing 100083, P.R. China*

E-mail: kycai@buaa.edu.cn

Received April 2, 2006; revised July 10, 2006.

**Abstract** This paper classifies software researches as theoretical researches, experimental researches, and engineering researches, and is mainly concerned with the experimental researches with focus on software reliability experimentation and control. The state-of-the-art of experimental or empirical studies is reviewed. A new experimentation methodology is proposed, which is largely theory discovering oriented. Several unexpected results of experimental studies are presented to justify the importance of software reliability experimentation and control. Finally, a few topics that deserve future investigation are identified.

**Keywords** software reliability, software experimentation, software reliability control, adaptive testing

## 1 Introduction

Software researches can roughly be classified as theory inventing, theory discovering, theory testing, and theory customerizing (applicablizing). This is shown in Fig.1, the term "theory" should be widely interpreted. It refers to an argument, an assertion, a principle, a model, or a hypothesis of software techniques, software processes or/and software systems/products. Theory inventing means that some kind of theories is created by researchers with intuitive judgments to characterize, describe, or interpret imaginary scenarios or observed phenomena. This is mainly the focus of theoretical researches. A famous example is the Turing machine[1]. Theory discovering means some kind of theories is discovered from observed data or evidence to model or interpret the underlying principles of the observed data or evidence. The controversial software science of Halstead is an example of discovered theory[2]. Theory testing means that some kind of theories is tested against observed data or evidence under certain circumstances or scenarios. The claim that the number of defects remaining in a game-playing software system is less than 20 is a theory that should be validated before the software is released. Theory discovering and theory testing are mainly the focus of the experimental researches of software. Finally, theory customerizing means some kind of theories is customerized to the software processes or systems of concern and the applicability of the theories is examined. The observation that software fault tree analysis techniques can efficiently identify causes of critical software faults and effectively improve design of flight control software is a customerized theory with respect to flight control software. This is mainly the focus of engineering researches.

Software reliability researches are devoted to software reliability improvement and assurance and can be tracked back to the work of Hudson in 1967[3], which models the behavior of software defect introduction and removal as a birth and death Poisson process. Large amount of software reliability research has been carried out, which can be classified as theoretical researches, experimental researches, and engineering researches accordingly. By software reliability experimentation we mean experimental researches of software reliability in which software experiments are conducted. According to Basili[4], an experiment is a form of empirical study where the researcher has control over some of the conditions in which the study takes place and control over the independent variables being studied; an operation carried out under controlled conditions in order to test a hypothesis against observation. With the undoubted and growing importance of software reliability problems in various application areas, a few questions arise naturally: what is the state-of-the-art of software reliability researches in general, and software reliability experimentation in particular? How should software reliability experiments be conducted? How can software experimentation help to evaluate new software reliability techniques?

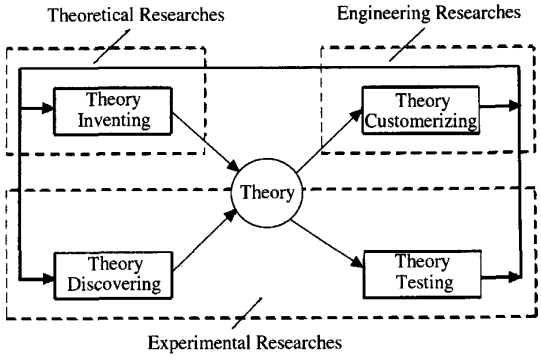


Fig.1. Classification of software researches.

This paper is aimed to address the above questions. More specifically, the aim of this paper is four-fold. First, this paper will review the current status of software relia-

Regular Paper
This work was supported by the National Natural Science Foundation of China (Grant Nos.60233020, 60474006 and 60473067).

bility researches with focus on software reliability experimentation. This will be done in Section 2. Second, this paper will propose a methodology for software reliability experimentation, which will be presented in Section 3 to guide how software reliability experiments should be conducted. Section 4 will present a few experimental results generated by the software reliability experimentation methodology. The third aim of this paper is to review a new software reliability technique for software reliability assessment, which is obtained by treating software reliability assessment as an adaptive control problem. This will be accomplished in Section 5 with preliminary experimental results. Finally, this paper will identify a few topics of research potential for software reliability experimentation and control. This will be demonstrated in Section 6. Concluding remarks are contained in Section 7.

## 2 Model-Based Studies and Empirical Studies

### 2.1 Brief History

The systematic researches on software reliability started with the recognition of the so-called software crisis in the late 1960s, of which software reliability problem was one of the major causes. In 1970s, the central themes of software reliability researches were software reliability modeling and software fault-tolerance. A large number of software reliability models were proposed[2,5−10]. These models are aimed to quantitatively evaluate software reliability and/or estimate the number of defects remaining in the software of concern. On the other hand, several approaches to software fault-tolerance were proposed, including the N-version programming[11] and the recovery block programming[12]. In this period the notion of software reliability was mainly interpreted as a quantitative index that measures the probability of failure-free operations of software systems. In other words, software reliability was interpreted in a narrow sense. The field of software reliability largely resembles that of hardware reliability.

In 1980s, researchers begun to examine which and what techniques can help to improve and assure software reliability in engineering practice[13−17]. Consequently, several important consensuses were reached. First, although certain software reliability techniques are useful and can help to improve and assure software reliability, most of available software reliability techniques fail to deliver their promises. Significantly better software reliability techniques are required. Second, besides being closely related to hardware reliability researches, the field of software reliability should be interpreted in a wide sense to cover disparate techniques in various research areas such as concurrent computing, model checking, real-time scheduling, intelligent decision-making, and so on. This is reflected in diverse topics covered in annual international symposium on software reliability engineering.

In 1990s, safety-critical software became a major theme in software reliability researches. How to guarantee the reliability of safety-critical software is a grand challenge to human intelligence. Since then, new kinds of applications including component-based software and Web software emerge as research topics for software reliability[18,19]. However it should be noted that among the various research topics, the mainstream techniques and applications can no longer be identified easily.

### 2.2 Model-Based Studies

Existing researches on software reliability can be divided into two parts: model-based studies and empirical studies. This subsection reviews model-based studies, whereas empirical studies are left to the next subsection. Model-based studies can be treated as synonymous with theory inventing to a large extent. However we stick to the former terminology since the notion of models is extensively adopted in software reliability researches.

A model-based study begins with making a set of assumptions for software reliability. Roughly speaking, it comprises the following steps:

1) A set of assumptions is made to abstract the software process or system under study to a certain level; ideally, the assumptions should fit engineering practice and be mathematically tractable.

2) A mathematical model is obtained, which can be analytic, stochastic, or formal.

3) The mathematical model is investigated to reveal the properties of interest and/or to derive measures of interest in a mathematically rigorous manner; consequently, theoretical results are obtained.

4) The theoretical results are tested or validated against some forms of evidence including observed data collected from real software projects or simulation settings.

Consider the Goel-Okumoto NHPP software reliability growth model as an example. It is based on the following assumptions[8]:

1) The software of concern is tested under anticipated operating environment.

2) For any finite set of time instants $t_1 < t_2 < \cdots < t_k$, the numbers of software failures, $f_1, f_2, \ldots, f_k$, observed in the time intervals $(0, t_1), (t_1, t_2), \ldots, (t_{k-1}, t_k)$, are independent.

3) Every software defect has an equal chance of being detected.

4) The cumulative number of software failures observed up to time $t$, denoted by $M(t)$, follows a Poisson distribution with the mean $m(t)$ such that the number of software failures observed in the micro-time-interval $(t, t + \Delta t)$ is proportional to the interval length and to the number of remaining software defects at time $t$.

5) $m(t)$ is a bounded, non-increasing function with $m(0) = 0$ and $m(t) \to a$ as $t \to \infty$, where $a$ is the total number of software failures eventually observed.

With the above assumptions it follows that $M(t)$ is a non-homogeneous Poisson process such that $\Pr\{M(t) =$

$k\} = \frac{(m(t))^k}{k!}\exp\{-m(t)\}$; $k \geqslant 0$, where $m(t) = a(1 - e^{-bt})$ with $b$ being the proportionality constant. This is a simplifying mathematical model for software failure processes and the reliability measures of interest can be derived. Further, suppose that $f_1, f_2, \ldots, f_k$ are given, then the likelihood function:

$$\Pr\{M(t_1) = f_1, M(t_2) - M(t_1) = f_2, \ldots, M(t_k) - M(t_{k-1}) = f_k\}$$

can be mathematically determined. The parameters $a$ and $b$ can be estimated according to the maximum likelihood method. The number of failures observed in the next time interval $(t_k, t_{k+1})$ can be predicted. The prediction accuracy against the actual number $f_{k+1}$ can be used to test if the Goel-Okumoto NHPP model fits the actual software reliability process.

The underlying philosophy of model-based studies is that a mathematically tractable model exists for the problem under investigation. Model is the central object of concern, and the mathematical analysis of the properties of the obtained model plays a key role. Besides various software reliability models[20,21] and others[22], model-based testing[23] and software model checking[24] also fall into the scope of model-based studies. The advantages of model-based studies lie in that the vast and rich library of existing mathematical tools can be exploited and the mathematical analysis of the given model may generate useful hints for developing more powerful models. Remember that in-depth researches for software reliability problems should eventually be based on or lead to appropriate mathematical models. However the disadvantages of model-based studies are also obvious. The underlying mathematical assumptions are not easy to be validated in engineering practice which is often much more complicated than what a mathematically tractable model may describe. The derived mathematical properties may or may not be useful for software process improvements.

### 2.3 Empirical Studies

Although empirical studies are reported for software reliability from time to time in scattered journals and conference proceedings[15,25–27], they are not the majority of software reliability researches. This contrasts with the growing awareness and widely accepted importance of empirical studies in software engineering in general[28–33], which are motivated by the fact that more often than not, various proposals for software process improvements and system optimizations were made without being backed by hard evidence or objective evaluation techniques for their effectiveness or advantages[34]. An empirical study is an act or operation on practical software processes or artifacts, in which data of interest are generated, collected, analyzed, and interpreted with the purpose of discovering, testing, customerizing, and/or improving certain known or unknown theories. In this broad sense empirical studies cover experimental researches as well as engineering researches. They can be observational, historical, and controlled[35].

Unlike model-based studies which are centered around mathematical models, an empirical study focuses on practical software processes and/or systems and the corresponding data of interest. The underlying philosophy is that theories should begin and end with practical subjects and do not make sense in isolation. A general procedure for an empirical study is as follows.

1) Select factors or theories whose effects are of interest to be examined.
2) Select subject software processes or systems on which the effects of the factors or theories are to be examined.
3) Build up the practical scenarios or experimental platforms from which data of interest are to be generated and/or collected.
4) Analyze the collected data to discover possible theories or examine hypothesized theories.

In a recent empirical study of regression testing[36], four factors (Test Suite Granularity, Test Input Grouping, Regression Testing Technique, Interactions) were considered and four hypotheses were examined as follows.

*H*1 (*Test Suite Granularity*): Test suite granularity does not have a significant impact on the costs and benefits of regression testing techniques.

*H*2 (*Test Input Grouping*): Test input grouping does not have a significant impact on the costs and benefits of regression testing techniques.

*H*3 (*Technique*): Regression testing techniques do not perform significantly differently in terms of the selected costs and benefits measures.

*H*4 (*Interactions*): Test suite granularity and test input grouping effects across regression testing techniques and programs do not significantly differ.

Four regression testing techniques, namely retest-all, regression test selection, test suite reduction, and test case prioritization were then applied to two subject programs: Emp-Server and Bash with each comprising nine consecutive versions. The testing data were collected and analyzed to examine the four hypotheses.

The main advantage of empirical studies is that theories of concern can be validated or invalidated by hard evidence with respect to practical software processes and/or products if the empirical studies are repeatable and realistic. This renders valid theories to be useful for software process improvements and system optimization. However, theories may be mis-validated or mis-invalidated if the collected data are not repeatable or realistic in some sense, and thus may deliver misleading guidelines for software process improvements and system optimization. This is sensitive since a large number of factors may play a role in an empirical study. It is not easy to distinguish important factors from unimportant factors. Further, human are often involved in data generation processes and the repeatability of the collected data

is at risk. Empirical studies can validate or invalidate theories, but cannot prove theories. The disadvantages of empirical studies are obvious. An additional disadvantage of empirical studies is that they may incur high costs and are time-consuming in building experimental platforms and generating/collecting data of interest.

### 2.4 Current Status

Rather than addressing the state-of-the-art of model-based studies and empirical studies in general, this subsection focuses on the current status of experimental software reliability studies. Accompanying the widespread experimentation in software engineering, the importance of experimental software reliability researches should not be doubted. With the recognition that theoretical researches that often fail to deliver their promises to software reliability improvements and assurance, more importance should be attached to experimental researches. Unfortunately, the amount of experimental researches for software reliability is limited. A recent review of testing technique experiments concludes that our current testing technique knowledge is very limited[37]. This is certainly unsatisfactory since experimental researches are playing a growing part in software engineering in general.

More specifically, existing experimental researches for software reliability are weak in several aspects as follows.

1) The amount of software reliability experiments is very limited. This is particularly true in comparison with the amount of model-based studies in software reliability and that of empirical studies in the general setting of software engineering.

2) The subjects selected for experiments are most concerned with software processes rather than software products or systems. This can be justified by the terminology of empirical software engineering. It is understood that empirical software engineering is the subdiscipline of software engineering that investigates software engineering techniques by using empirical methods: case study, inquiry, and experiment[38], whereas software engineering means application of a systematic, disciplined, quantifiable approach to development, operation and maintenance of software[39]. However software reliability refers to the reliability of software systems rather than the reliability of software processes. The subjects selected for experiments should achieve a balance between software processes and software systems.

3) The theories examined in experiments are often shallow, inaccurate, non-rigorous, non-systematic, or non-scientific. For example, the following statements were treated as theories for investigation[38]: when applying software engineering techniques team composition (e.g., team size) has an effect on software development; software engineering techniques of one paradigm (e.g., object-oriented paradigm) differ in their efficiency and effectiveness; design techniques with coarse-grained structuring concepts are superior to those without coarse-grained structuring concepts when developing complex application systems with respect to costs, quality, and maintenance.

4) Research methodology is ill-defined[40]; the repeatability of experiments is questionable[41]; the reporting guidelines are diverse[42].

5) Research infrastructure for experiments is non-standardized with specific purpose, and little attention has been paid to this topic[43,44].

6) Experiments are mainly concerned with theory customerizing and theory testing, and theory discovering is largely ignored, although there is exception[45].

7) The gap between theoretical researches and engineering researches/practice is obviously wide. Few results obtained in theoretical researches can help to improve software processes or to optimize software systems. On the other hand, few mathematical models were extracted from engineering researches/practice for theoretical researches.

## 3 Methodology for Software Reliability Experimentation

As indicated in Subsection 2.3, an important weakness of existing empirical studies is that the repeatability and realism are not guaranteed. Toy subject processes or products rather than real applications are often employed. The collected data may not be statistically repeatable. For example, in the empirical study of regression testing reported in [36], the subject programs contained only nine versions (refer to Subsection 2.3). This looked like a sample that comprises only nine data points. The resulting datasets could hardly make statistical sense. Another weakness of existing empirical studies is that they mainly focus on theory testing and theory customerizing. The corresponding research methodology adopted is not systematic or theory discovering oriented.

This section proposes a statistical methodology for software reliability experimentation which is appropriate for theory discovering. It consists of the following six steps.

Step 1. Experiment Planning: This step determines the purposes of the experimental study (which can be theory discovering or theory testing) and the subject software processes or systems. Typical purposes for an experimental study of software testing ① include:

1) compare the defect detection capabilities of random testing and partitioning testing for GUI applications;

①Conventionally, the role of software testing is treated as two-fold. First software testing can be conducted for reliability improvements by detecting and removing software defects. Second, it can also be conducted for reliability assessment which judges by freezing the code of the software under test whether the software has achieved the given quantitative reliability objective. However in an experimental study, software testing serves as a vehicle for discovering unknown theories of software or testing known theories, and this defines a third but unconventional fold of the role of software testing.

2) analyze the capability of random testing for detecting correlated defects;

3) assess the capabilities of distinct test suites for detecting defects;

4) analyze the invocation behavior of distinct components inside a software system.

The subject software for testing should be chosen carefully to make the software testing experiment realistically. First, the subject software should be a commercial application that was developed by professionals and put into field operation already. Applications developed by students should be avoided. Second, the size of the subject software should be reasonably large to contain tens of thousands of lines of codes. Small applications with a few thousands of lines of codes should be avoided unless they are exclusively chosen for the purpose of program analysis. Third, the subject software should demonstrate modern software technology such as middleware technology, component-based technology, web technology, and Linux technology. Applications developed by using out-dated technology such as Fortran language should be avoided.

Fig.2 shows the architecture of the subject software, namely Intra3D, that was employed in several experimental studies of ours. Intra3D is a GUI application that comprises two different libraries of C++ classes and COM classes. The former contains more than 30,000 lines of codes, whereas the latter contains more than 70,000 lines of codes. Fig.3 is the interface of the application, TextureMapping, which is composed of C++ classes of Intra3D.

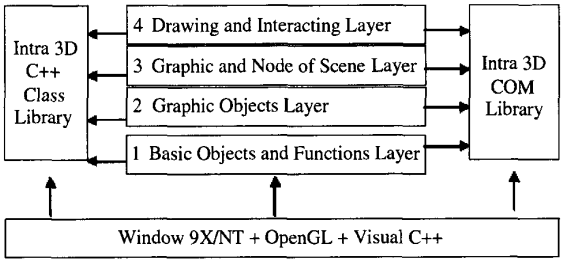


Fig.2. Architecture of Intra3D.

Step 2. Experiment Platform Set-Up: Setting-up an experiment platform to automate the experimental process is essential for human invention to be avoided during the testing process. This will ensure that a large number of trails of software testing can be conducted in a time-efficient manner and massive data which make a statistical sense can be collected. Three components are indispensable for an appropriate experiment platform. The first component is the test suite of interest. For example, in the experimental studies with the application TextureMapping, the employed test suite comprises 2000 test cases, each being a finite sequence of actions applied to TextureMapping. The second component is an execution unit such as WinRunner which selects and applies test cases to the software under test. The third component is that used for analysis and processing of testing data. Fig.4 shows the interface of the experiment platform for TextureMapping.

Fig.3. Interface of TextureMapping.

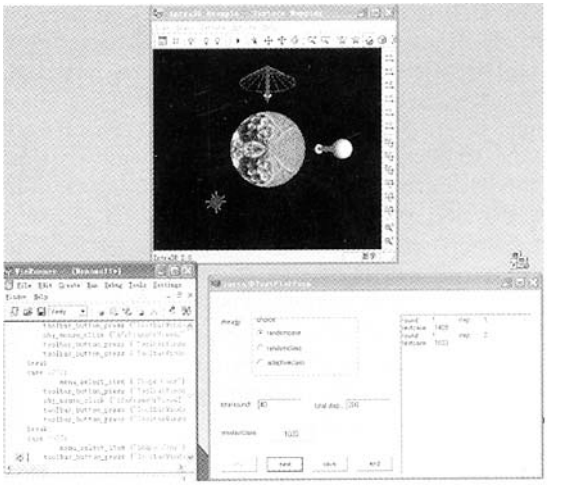

Fig.4. Experiment platform for TextureMapping.

Step 3. Speculative Experiment: this step conducts a number of experimental trials for a speculative purpose. At the beginning of the experimental study, the theory of interest may be obscure. The corresponding experimental plan may not be well described and is tentative to a large extent. For example, in order to test the theory that the invocation behavior among objects of an object-oriented software system follows a Markovian process, it is necessary to present an accurate definition for software states, to choose the mathematical method that can do hypothesis testing for the Markovian property, to select a testing strategy, and to determine the number of trials of software testing that should be conducted. However these may not be well done before the subject software is subjected to testing. A pragmatic solution is that software states are tentatively defined, and software testing

strategies and hypothesis testing methods are tentatively selected too. After a modest amount of software testing, phenomena of interest may emerge. This may lead to hypothesized theories that are unknown at the beginning of the experimental study. The purpose of speculative experiment is to identify theories that will be tested further in details.

Step 4. Specialized Experiment: This step conducts a large amount of experiments and collects massive data that make a statistical sense to test the theory of interest identified in the speculative experiment. For example, in an experimental study with the application TextureMapping, the total 32 distinct classes of the application were divided into 5 disjoint parts, each defining a distinct software states. The speculative experiment revealed that the corresponding software state transitions might follow a non-Markovian process. In order to test this observation or theory, 40 trials of software testing were conducted in the specialized experiment and the state transition behaviors were recorded for the 40 trials.

Step 5. Preliminary Analysis: This step analyzes the data collected in the specialized experiment from an intuitive and phenomenological perspective. It calculates statistics of interest and performs the corresponding statistical hypothesis testing. This leads to conclusions that need further systematic and in-depth analysis. For example, in an experimental study with the application TextureMapping, two important conclusions were drawn in the preliminary analysis of the testing data. First, the software state transitions do not fit a Markovian process. Second, certain law governs the software state transitions. Preliminary analysis takes place in a mathematical or statistical context. The corresponding physics backgrounds or interpretations are not involved.

Step 6. Systematic Summarization: This step summarizes the conclusions drawn in the preliminary analysis and interprets them in terms of physics or software languages. It reveals the physics roots that lead to the conclusions, discusses their relationships to related works, and proposes new mathematical models or pose new mathematical problems. For example, why do software state transitions fail to follow a Markovian process? Is this caused by the software requirement specification (software functionality) or software defects? Is this an Intra3D specific property or a widely valid one? Why is the Markovian assumption extensively made in computer software performance evaluation and reliability modeling? How can the non-Markovian properties be mathematically formulated?

The novelty of the proposed methodology lies in the following observations. First, a key point of the methodology is to build up the practical scenarios or experimental platforms that can avoid human intervention and automate the experiment process with the purpose that the generated or collected data can make statistical sense. Second, this methodology distinguishes between speculative experiment and specialized experiment. This reflects the intrinsic nature of theory discovering. Third, the statistical repeatability is enforced in Steps 2 and 4.

## 4 Phenomenological Implications

The methodology presented in Section 3 was applied in several experimental studies of software testing and revealed unexpected phenomena which were inconsistent with common belief in software engineering.

### 4.1 Fall of the NHPP Assumption

The NHPP assumption, which means that the number of software failure observed up to time $t$, denoted by $M(t)$, follows a non-homogeneous Poisson process, is popular in software reliability modeling[46-48]. The Goel-Okumot model presented in Subsection 2.2 is an example. This assumption implies that the mean and the variance of $M(t)$ are equal. In an experimental study with the Space program ②, 40 trials of testing were conducted for the program. At beginning of each trial, the program contained 38 known defects, of which 36 could be detected by the given test suite which comprised 13,498 distinct test cases. Upon a failure being observed, one and only failure-causing defect was removed from the program. The trial stopped upon the 36 detectable defects being removed. Fig.5 shows the behavior of the estimates of the mean and the variance of $M(t)$. This invalidates the NHPP assumption.

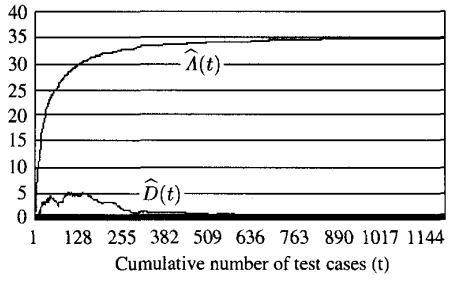


Fig.5. Estimates of the mean and the variance, denoted $\hat{\Lambda}(t)$ and $\hat{D}(t)$, respectively, of $M(t)$.

### 4.2 Fall of the Markovian Assumption

The Markovian assumption is widely adopted in software performance evaluation as well as in software reliability modeling[51,52]. It states that the transitions among software states during software operation follow a Markov process. In order to validate this assumption, an experimental study was conducted with the application TextureMapping, which included 40 trials of software testing. In each trial 200 test cases were selected from the test suite (comprising 2000 test cases) at random and applied to the application. The invocation

②The Space program was widely employed as the subject program for software testing research[49,50].

behavior of 32 distinct classes was recorded. These 32 classes were then divided into several parts, each defining a distinct software state. In this way the transition behavior of software states was available for 40 trials of software testing. The $\chi^2$ method was selected to test the Markovian assumption. Fig.6 shows the test results for 8 different definitions of software states, which have different numbers of software states: 5, 32, 33, 36, 40, and 64. The hypothesis that the Markovian assumption holds is accepted if the relative value (of the vertical axis) is negative. The experimental results reject the hypothesis.

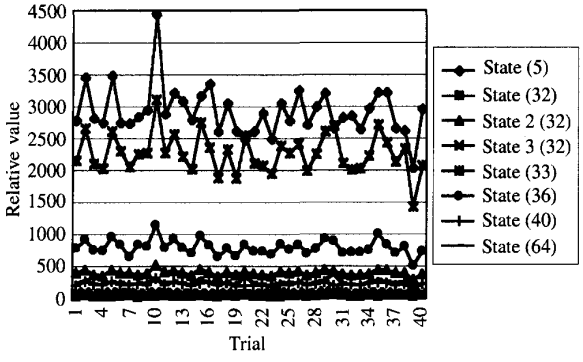


Fig.6. $\chi^2$ hypothesis test results for the Markovian assumption.

### 4.3 Large Number Theorem Revisited

The large number theorem is fundamental is probability theory. It states that the frequency of occurrence of an event approaches a certain value as the sample size increases into infinity. It is interesting to examine if the large number theorem still holds in the network environment. To this end, an experimental study was conducted for mutation testing of the Space program[53]. 36 mutants of the Space program were created, each comprising a defect, and subjected to testing. In each test a mutant was selected accordingly and the testing result (success or failure) was recorded. Before testing, the usage rate of the CPU of a laboratory server (connected to the Internet) every 500ms for 10,000 times and saved the sampled data in a file. These data were then used to select a mutant for testing in each test. Suppose the usage rate was 90%, then the reminder of 90/36 was 18. In this way mutant 18 of the Space program was selected and subjected to testing. 10,000 tests were performed in total. Let $f(n)$ be the number of failures observed in the first $n$ tests and $\hat{f}(n) = f(n)/n$. Fig.7 shows the behavior of $\hat{f}(n)$, which does not seems to fluctuate around a certain value. The large number theorem should be re-examined in the Internet context.

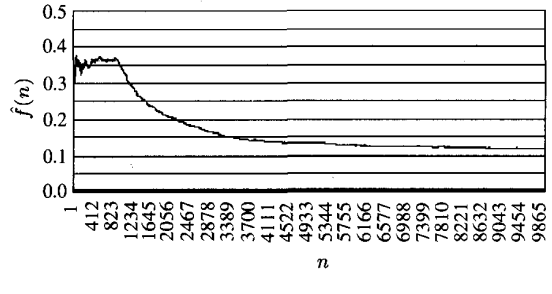


Fig.7. Behavior of $\hat{f}(n)$ for the Space program with server CPU usage rate based mutant selection scheme.

### 4.4 Single-Action-Multiple-Transition Automata

In order to identify the root causes for the non-Markovian behavior of the software state transitions, an interesting property of the software state transitions draws our attention. Suppose that object $A$ in an object-oriented software system invokes method $B$ provided by another object during software operation. However method $B$ may in turn invoke method $C$ who will further invoke method $D$. Then method $D$ returns to method $C$ who will in turn return to method $B$. Eventually, method $B$ returns to object $A$ and the operation triggered by the invocation of object $A$ terminates. The single invocation action leads to a sequence of transitions among methods and objects. This contrasts with conventional models of automata. Consider the following definition[1].

**Definition 4.1.** *A finite automaton is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$, *where*

1) $Q$ *is a finite set called the states,*
2) $\Sigma$ *is a finite set called the alphabet,*
3) $\delta : Q \times \Sigma \to Q$ *is the transition function,*
4) $q_0 \in Q$ *is the start state, and*
5) $F \subseteq Q$ *is the set of accept states.*

Note that an implicit assumption in the above definition or transition function is that a single action triggers one and only one state transition. It actually defines a class of single-action-single-transition (SAST) automata. The following definition, which is first proposed in this paper, may be more appropriate for characterizing software behavior.

**Definition 4.2.** *A finite single-action-multiple-transition (SAMT) automaton is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$, *where*

1) $Q$ *is a finite set called the states,*
2) $\Sigma$ *is a finite set called the alphabet,*
3) $\delta : Q \times \Sigma \to Q^*$ *is the single-action-multiple-transition transition function, with* $Q^*$ *being the Kleene star of* $Q$,
4) $q_0 \in Q$ *is the start state, and*
5) $F \subseteq Q$ *is the set of accept states.*

SAMT automata have not been tackled in the literature. Besides SAST automata and SAMT automata, multiple-action-single-transition (MAST) automata and multiple-action-multiple-transition (MAMT) automata can also be defined in a similar manner. While SAST automata can be viewed as the stochastic counterpart to Markov processes, the stochastic counterpart to SAMT automata may no longer be Markov processes. An interesting problem is whether SAMT Turing machines are equivalent to standard Turing machines in some sense.

## 5 Experimentation for Software Reliability Control

### 5.1 Software Reliability Control

Software reliability control refers to various activities that take place in software processes and software system operations for software reliability improvement and/or assurance. Software fault tolerance is aimed to control software operation behavior in the presence of software faults. Software FMEA and FTA analyze the effects of software failure modes and identify the causes of software failures. This may lead to better software design solutions that avoid catastrophic failures. In this sense software FMEA and FTA serve as a control activity in the software development process. Software testing detects and removes various defects before the software under test is released and is an effective paradigm for software improvement and assurance. However it is a grand challenge to formulate the feedback mechanisms in software processes and software system operations, to put the software reliability control problem into the context of feedback control principles and theory, and to quantify the gains and costs of software reliability control activities. Software reliability control should be upgraded to make the behavior of software reliable, repeatable, and predictable. New techniques or theories for software reliability control should be subjected to experimentation for their effectiveness and efficiency.

### 5.2 Experiments for Adaptive Testing

Software testing is arguably the least understood part in software development[54] and lacks theoretical foundation. In order to alleviate this problem to some extent, it is shown that software testing can be treated as feedback control problem[55]. The software under test serves as the controlled object, and the testing strategy serves as the corresponding controller. The software under test and the testing strategy constitute a closed-loop feedback system, and the control theory can be adapted to synthesize the required testing strategy in accordance with the given (quantitative) goal. In particular, adaptive testing refers to the software testing counterpart to adaptive control. Fig.8 shows a typical diagram of adaptive testing. The software under test is modeled as a controlled Markov chain, $H_t$ denotes the history of software testing up to time $t$, $A_{t+1}$ the action (test case) selected at time $t+1$, and $Z_{t+1}$ the corresponding output at time $t+1$. Adaptive testing can be performed for software reliability improvement[55] as well as for software reliability assessment[56]. It is important to evaluate if the adaptive testing strategies can outperform conventional testing strategies that are not based on adaptive control theories.

The adaptive testing (AT) strategy proposed in [56] deals with this problem: how to test software with a given number of tests such that the resulting testing data can lead to best or most trustworthy software reliability estimate. In order to validate the effectiveness of the adaptive testing strategy, four experiments were conducted for the Space program in comparison with the random testing (RT) strategy and the operational profile based random testing (ORT) strategy.

*Experiment* I. In this experiment, test suite 1 was used (testing scenarios 1 to 25) and $x_0 = 500$.

*Experiment* II. In this experiment, test suite 2 was used (testing scenarios 26 to 50) and $x_0 = 500$.

*Experiment* III. In this experiment, test suite 1 was used (testing scenarios 1 to 25) and $x_0 = 3000$.

*Experiment* IV. In this experiment, test suite 2 was used (testing scenarios 26 to 50) and $x_0 = 3000$.

Here $x_0$ represents the number of tests that are applied to a trial of software testing, and a testing scenario was described by the program variant (mutant of the Space program), the test suite, the operational profile, the defect detection rate, and the corresponding software reliability. Tables 1 and 2 show some of the experimental results, where *SD* denotes the standard deviation of the software reliability estimate. The testing strategy AT substantially outperformed the testing strategies RT and ORT in terms of *SD*.

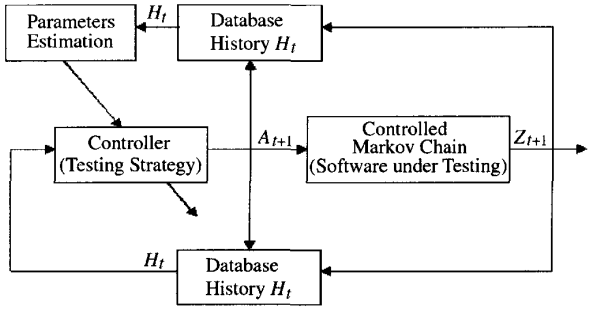


Fig.8. Diagram of adaptive testing.

## 6 Example Topics of Research Potential

### 6.1 General Questions

Halstead might be the first person who carried out systematic researches to examine the nature of software with the belief that there exist physics-like laws that obey each piece of software[2]. His belief is now widely rejected in the field of software since the proposed software science formulae are not valid. Unlike physical systems whose behavior may demonstrate full repeatability in some sense, software processes or systems may be featured with partial repeatability of their behavior[45]. Several fundamental questions concerning the nature of software can be raised and deserve further investigation in the future.

1) Question for Galileo: are scientific research methodologies applicable to software researches? Galilo pioneered modern scientific researches by introducing objective and systematic research methodologies which can be described as follows[57]: Observing phenomena → Presenting hypotheses → Inferring with mathematical

**Table 1.** Performance Comparison of the Testing Strategy AT with the Testing Strategy RT in Terms of $SD$

| Experiment | $\chi_{AT/RT}(SD)$ | $\chi_{RT/AT}(SD)$ | $\chi_{AT/RT}(SD)/25$ | $\chi_{RT/AT}(SD)/25$ | $\lvert \chi_{AT/RT}(SD) - \chi_{RT/AT}(SD) \rvert /25$ |
|---|---|---|---|---|---|
| I | 20 | 5 | 80% | 20% | 60% |
| II | 18 | 7 | 72% | 28% | 44% |
| III | 19 | 6 | 76% | 24% | 52% |
| IV | 17 | 8 | 68% | 32% | 36% |

Note: $\chi_{AT/RT}(SD)$ = number of testing scenarios for which the testing strategy AT outperformed the testing strategy RT in terms of $SD$;
$\chi_{RT/AT}(SD)$ = number of testing scenarios for which the testing strategy RT outperformed the testing strategy AT in terms of SD.

**Table 2.** Performance Comparison of the Testing Strategy AT with the Testing Strategy ORT in Terms of $SD$

| Experiment | $\chi_{AT/ORT}(SD)$ | $\chi_{ORT/AT}(SD)$ | $\chi_{AT/ORT}(SD)/25$ | $\chi_{ORT/AT}(SD)/25$ | $\lvert \chi_{AT/ORT}(SD) - \chi_{ORT/AT}(SD) \rvert /25$ |
|---|---|---|---|---|---|
| I | 14 | 11 | 56% | 44% | 12% |
| II | 14 | 11 | 56% | 44% | 12% |
| III | 16 | 9 | 64% | 36% | 28% |
| IV | 16 | 9 | 64% | 36% | 28% |

Note: $\chi_{AT/ORT}(SD)$ = number of testing scenarios for which the testing strategy AT outperformed the testing strategy ORT in terms of SD;
$\chi_{ORT/AT}(SD)$ = number of testing scenarios for which the testing strategy ORT outperformed the testing strategy AT in terms of SD.

and logical tools to obtain derived properties → Validating the derived properties with experimentation → Formulating theories. The scientific research methodologies are seldom adapted for software researches.

2) Question for Kepler: can fundamental laws be extracted or mined from massive software data? Kepler discovered his three fundamental laws of planetary motions by analyzing the massive data observed by Tycho, and was the first scientist who showed that the multitude of imperfect data could be coped with to forge a theory of surpassing accuracy[58]. With massive software data generated from software experimentation, an open problem is how to discover the underlying laws that govern the behavior of software processes and systems.

3) Question for Kolmogorov: is software behavior random or stochastic? Algorithmic complexity theory allows as rigorous definition of randomness as possible. All non-computable strings are algorithmically random[59]. It is not clear how to calculate the Kolmogorov complexity for software behavior.

4) Question for Wiener: is software behavior controllable? The systematic theories and practice of control begun with Wiener's cybernetics[60] and controllability is a fundamental notion of dynamic systems. In order to effectively control software reliability behavior, it is essential to explore the controllability problem for software and this poses a challenge to software researches.

### 6.2 Theory Discovering

As pointed out in Section 2, existing software researches are mainly concerned with theory inventing, theory testing, and theory customerizing. Theory discovering is largely neglected. Furthermore, software systems are less tackled in comparison with software processes. Experimental researches in the future should pay more attention to theory discovering, devote more efforts to software systems, and serve as a bridge linking between theoretical researches and engineering researches ③.

### 6.3 Learning-Investigation-Assessment Paradigm

Large amount of software research was devoted to software reliability assessment and various approaches were proposed, including black-box approaches[61], white-box approaches[52], evidence-based approaches[62], and least variance approaches[56]. However none of them has been shown to be capable of validating highly dependable software. A fundamental drawback of these approaches is that they only take account of partial information concerning software reliability behavior. The underlying models for software reliability behavior are over-simplifying. A comprehensive paradigm that can combine various sources of reliability-related information, including model checking results, software architecture, software testing, operational profile, legacy data, and human experience, in an appropriate manner, seems indispensable. Note that a large variety of learning approaches is available in the field of machine learning[63–65]. In the Angluim's approach of querying, two types of queries, i.e., equivalence queries and membership queries, are involved. The equivalence querying checks whether a given model is equivalent to the model under learning. A counterexample is generated if the answer is negative. This is similar to the results of model checking[24]. The membership querying checks whether a particular domain element belongs to the unknown model or not. This is similar to software testing. In this way a learning-investigation-assessment (LIA) paradigm may be proposed for software reliability assessment. In

③We may coin a new term, Experimental Softwarics, to refer to experimental researches for software processes and systems. Experimental softwarics is supposed to be the software counterpart to experimental physics. It is different from empirical software engineering which does not explore the nature of software from the physics perspective.

the learning stage, machine learning approaches are applied to identifying the required model for software. For example, an interesting problem is how to identify an extended finite state machine model for software systems by using the Angluim's approach. In the investigation stage, the software model identified in the learning stage is checked and analyzed to derive useful mathematical properties. In the assessment stage, the quantitative reliability indexes are eventually obtained.

### 6.4 Software Cybernetics

As an emerging area that includes software reliability control as one of its topics, software cybernetics explores the interplay between software/software behavior, and control[66,67]. The fundamental question of interest is: when can, and quantitatively speaking, how can software behavior, software processes, or software systems, be adapted or evolved to meet old and new objectives in the presence of a changing environment, e.g., disturbances, faults, or expanded requirements? This emerging and inter-disciplinary area addresses issues and questions related to 1) formalization and quantification of feedback and self-adaptive control mechanisms in software; 2) adaptation of the principles of control theory to software processes and systems; 3) application of the principles of software engineering and theories to control systems; and 4) integration of the theories of software engineering and control engineering. Software cybernetics is an area that is largely un-explored.

## 7 Concluding Remarks

Software reliability experimentation and control is a field that is largely neglected in previous software researches. However the importance of this field is growing. The contribution of this paper can be summarized as follows.

1) The state-of-the-art of empirical studies for software engineering in general, and for software reliability improvement, assurance and control in particular, is reviewed.

2) A new software experimentation methodology is proposed, which is largely theory discovering oriented.

3) A few unexpected or positive results generated from several experimental studies, are presented.

4) Several example topics that deserve future investigation are identified.

**Acknowledgment** The experimental results presented in this paper were obtained with the help of many students of the author, including Bo Gu, Hai Hu, Chang-Hai Jiang, Xiao-Feng Lei, Yan Shi, and Bei-Bei Yin.

## References

[1] Sipser M. Introduction to the Theory of Computation. PWS Publishing Company, 1997.
[2] M.H. Halstead. Elements of Software Science. North-Holland. 1977.
[3] Hudson G R. Program errors as a birth and death process. System Development Corporation. Report SP-3011, Santa Monica, CA, 1967.
[4] Basili V, Shull F, Lanubile F. Using experiments to build a body of knowledge. In *Proc. Ershov Memorial Conference*, 1999, pp.265–282.
[5] Jelinski Z, Moranda P B. Software Reliability Research. Statistical Computer Performance Evaluation, Greiberger W (ed.), Academic Press, 1972, pp.465–484.
[6] Musa J D. A theory of software reliability and its application. *IEEE Trans. Software Engineering*, 1975, SE-1(3): 312–327.
[7] Littlewood B, Verrall J. A Bayesian reliability growth model for computer software. *Applied Statistics*, 1973, 22(3): 332–346.
[8] Goel A L, Okumoto K. A time dependent error detection rate model for a large scale software system. In *Proc. the Third USA-Japan Computer Conference*, San Francisco, USA, 1978, pp.35–40.
[9] Nelson E C. Estimating software reliability from test data. *Microelectronics and Reliability*, 1978, 17(1): 67–74.
[10] Mills H D. On the statistical validation of computer program. FCS-72-6015, IBM Federal System Division, 1972.
[11] Avizienis A. Fault tolerant systems. *IEEE Trans. Computer*, 1976, C-25: 1304–1312.
[12] Randell B. System structure for software fault tolerance. IEEE *Trans. Software Engineering*, 1975, SE-1(2): 220–232.
[13] Cai K Y, Wen C Y, Zhang M.L. A critical review on software reliability modeling. *Reliability Engineering and System Safety*, 1991, 32: 357–371.
[14] Anderson T, Barret P A, Halliwell D N, Moulding M R. Software fault tolerance: An evaluation. *IEEE Trans. Software Engineering*, 1985, SE-11(12): 1502–1510.
[15] Knight J C, Leveson N G. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Software Engineering*, 1986, SE-12(1): 96–109.
[16] Butler R W, Finelli G B. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Software Engineering*, 1993, 19(1): 3–12.
[17] Littlewood B, Strigini L, Validation of ultra-high dependability for software-based systems. *Communications of the ACM*, 1993, 36(11): 69–80.
[18] Reussner R H, Schmidt H W, Poernomo I H. Reliability prediction for component-based architectures. *Journal of Systems and Software*, 2003, 66: 241–252.
[19] Huynh T, Miller J. Further investigations into evaluating website reliability. In *Proc. Int. Symp. Empirical Software Engineering*, Noosa Heads, Australia, 2005, pp.162–171.
[20] Xie M. Software Reliability Modeling. World Scientific, 1991.
[21] Cai K Y. Software Defect and Operational Profile Modeling. Academic Publishers, 1998.
[22] Cai K Y, Wang X Y. Towards a control-theoretical approach to software fault-tolerance. In *Proc. the 4th Int. Conference on Quality Software*, IEEE Computer Society Press, Braunschweig, Germany, 2004, pp.198–205.
[23] Pretschner A *et al.* One evaluation of model-based testing and its automation. In *Proc. Int. Conf. Software Engineering*, St. Louis, MO, USA, 2005, pp.302–401.
[24] E M Clarke Jr, O Grumberg, D A Peled. Model Checking. The MIT Press, 1999.
[25] Biffl B, Gutjahr W J. Using a reliability growth model to control software inspection. *Empirical Software Engineering*, 2002, 7: 257–284.
[26] Stringfellow C, Andrews A A. An empirical method for selecting software reliability growth models. *Empirical Software Engineering*, 2002, 7: 319–343.
[27] Henningsson K, Wohlin C. Assuring fault classification agreement—An empirical evaluation. In *Proc. Int. Symp. Empirical Software Engineering*, 2004, pp.95–104.
[28] Curtis B. Measurement and experimentation in software engineering. *Proc. the IEEE*, 1980, 68(9): 1144–1157.
[29] Tichy W F, Lukowicz P, Prechelt L *et al.* Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 1995, 28: 9–18.

[30] Tichy W F. Should computer scientists experiment more? *Computer,* May 1998, 31(5): 32–40.

[31] Perry D E, Porter A A, Votta L G. Empirical studies of software engineering: A roadmap. In *Proc. ICSE Workshop on Future of Software Engineering,* Edinburgh, UK, 2000, pp.347–355.

[32] Harrison R, Badoo N, Barry E *et al.* Directions and methodologies for empirical software engineering research. *Empirical Software Engineering,* 1999, 4(4): 405–410.

[33] Sjoberg D I K, Hannay J E, Hansen O *et al.* A survey of controlled experiments in software engineering. *IEEE Trans. Software Engineering,* 2005, 31(9): 733–753.

[34] Fenton N, Pfleeger S L, Glass R L. Science and substance: A challenge to software engineering. *IEEE Software,* July 1994, pp.88–95.

[35] Zelkowitz M V, Wallance D R. Experimental models for validating technology. *IEEE Software,* May 1998, pp.23–31.

[36] Rothermel G, Elbaum S, Malishevsky A G *et al.* On test suite composition and cost-effective regression testing. *ACM Trans. Software Engineering and Methodology,* 2004, 13(3): 277–331.

[37] Juristo N, Moreno A M, Vegas S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering,* 2004, 9: 7–44.

[38] Zendler A. A preliminary software engineering theory as investigated by published experiments. *Empirical Software Engineering,* 2001, 6: 161–180.

[39] IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, 1990.

[40] Kitchenham B A, Pfleeger S L, Pickard L M *et al.* Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Engineering,* 2002, 28(8): 721–734.

[41] Cater-Steel A, Toleman M, Rout T. Addressing the challenges of surveys in software engineering. In *Proc. 2005 Int. Symp. Empirical Software Engineering,* 2005, pp.204–213.

[42] Jedlitschka A, Pfahl D. Reporting guidelines for controlled experiments in software engineering. In *Proc. 2005 Int. Symp. Empirical Software Engineering,* Noosa Heads, Australia, 2005, pp.95–104.

[43] Do H, Elbaum S, Rothermel G. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. 2004 Int. Symp. Empirical Software Engineering,* Redondo Beach, CA, USA, 2004, pp.60–70.

[44] Boehm B *et al.* Using empirical testbeds to accelerate technology maturity and transition: The SCRover experience. In *Proc. 2004 Int. Symp. Empirical Software Engineering,* Redondo Beach, CA, USA, 2004, pp.117–126.

[45] Cai K Y, Chen L. Analyzing software science data with partial repeatability. *Journal of Systems and Software,* 2002, 63: 173–186.

[46] Xie M. Software Reliability Modeling. World Scientific, 1991.

[47] Cai K Y. Software Defect and Operational Profile Modeling. Kluwer Academic Publishers, 1998.

[48] Pham H. Software reliability and cost models: Perspectives, comparison, and practice. *European Journal of Operational Research,* 2003, 149: 475–489.

[49] Vokolos F I, Frankl R H. Empirical evaluation of the textual differencing regression testing technique. In *Proc. the Int. Conf. Software Maintenance,* Bethesda, Maryland, USA, November 1998, pp.44–53.

[50] Rothermel G, Untch R H, Chu C *et al.* Prioritizing test cases for regression testing. *IEEE Trans. Software Engineering,* 2001, 27(10): 929–948.

[51] Grassi V, Mirandola R. Devivation of Markov models for effectiveness analysis of adaptable software architectures for mobile computing. *IEEE Trans. Mobile Computing,* 2003, 4(2): 114–131.

[52] Goseva-Popstojanova K, Trivedi K. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation,* 2001, 45: 179–204.

[53] Cai K Y, Lei X F, Shi Y *et al.* Partial-repeatability as a new form of uncertainty. Fuzzy Logic, Software Computing and Computational Intelligence, Yingming Liu, Guoqing Chen, Mingsheng Ying (eds.), *Proc. 11th Int. Fuzzy Systems Association World Congress,* Tsinghua University Press/Springer, Beijing, 2005, pp.157–166.

[54] Whittaker J A. What is software testing? And why is it so hard? *IEEE Software,* January/February, 2000, 17(1): 70–79.

[55] Cai K Y. Optimal software testing and adaptive software testing in the context of software cybernetics. *Information and Software Technology,* 2002, 44: 841–855.

[56] Cai K Y, Li Y C, Liu K. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology,* 2004, 46: 989–1000.

[57] Guo Y L, Shen F J. The History of Physics. Tsinghua University Press, 1993. (in Chinese)

[58] Johannes Kepler. His life, his laws and times. http://kepler.nasa.gov/johannes/.

[59] Li M, Vitanyi P M B. An Introduction to Kolmogorov Complexity and Its Applications, Second Edition. Springer-Verlag, 1997.

[60] Wiener N. Cybernetics: Or Control and Communication in the Animal and the Machine. John Wiley & Sons, 1948.

[61] Lyu M R (ed.). Handbook of Software Reliability Engineering. McGraw-Hill, 1996.

[62] Fenton N, Littlewood B, Neil M. Applying Bayesian belief network in systems dependability assessment. In *Proc. Safety Critical Systems Symposium,* Springer-Verlag, Leeds, UK, 1996, pp.71–94.

[63] Valiant L G. A theory of the learnable. *Communications of the ACM,* 1984, 27(11): 1134–1142.

[64] Angluim D. Queries revisited. *Theoretical Computer Science,* 2004, 313: 175–194.

[65] C de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition,* 2005, 38: 1332–1348.

[66] Cai K Y, Cangussu J W, DeCarlo R A *et al.* An overview of software cybernetics. In *Proc. the 11th Int. Workshop on Software Technology and Engineering Practice,* IEEE Computer Society Press, Amsterdam, Holland, 2003, pp.77–86.

[67] Belli F, Cai K Y, DeCarlo R A *et al.* Introduction to the special section on software cybernetics. *Journal of Systems and Software,* article in press, available online, 2006.

**Kai-Yuan Cai** is a Cheung Kong Scholar (Chair Professor), jointly appointed by the Ministry of Education of China and the Li Ka Shing Foundation of Hong Kong in 1999. He has been a full professor at Beihang University (Beijing University of Aeronautics and Astronautics) since 1995. He was born in April 1965 and entered Beihang University as an undergraduate student in 1980. He received his B.S. degree in 1984, M.S. degree in 1987, and Ph.D. degree in 1991, all from Beihang University. He was a research fellow at the Centre for Software Reliability, City University, London, and a visiting scholar at City University of Hong Kong, Swinburge University of Technology (Australia), University of Technology, Sydney (Australia), and Purdue University (USA). Dr. Cai has published many research papers and is the author of three books: *Software Defect and Operational Profile Modeling* (Kluwer, Boston, 1998); *Introduction to Fuzzy Reliability* (Kluwer, Boston, 1996); *Elements of Software Reliability Engineering* (Tshinghua University Press, Beijing, 1995, in Chinese). His main research interests include software reliability and testing, and intelligent systems and control.