2015 13th International Conference on Frontiers of Information Technology

On Effectiveness of Fault-Seeding using Interaction Patterns

Tamim Ahmed Khan
Department of Software Engineering
Bahria University, Islamabad
Email: tamim@bui.edu.pk

Muhammad Muzammal
Department of Computer Science
Bahria University, Islamabad
Email: muzammal@bui.edu.pk

Anas Ijaz
Department of Software Engineering
Bahria University, Islamabad
Email: anas.ijaz@yahoo.com

***Abstract*—Fault Seeding is a testing technique where faults are artificially injected into an application to assess the effectiveness, i.e. if a given test suite is capable of uncovering the injected faults, of a test suite. This is helpful in establishing confidence in the test suite and is an alternative to structural testing methods. One of the issues with fault seeding is the identification of potential areas in the application, where the faults are to be seeded. We argue that if the intended usage of the application under test could be inferred from the potential users' interactions with the application, such information could be incorporated into the fault-seeding process. This could lead to more effective fault-seeding in a test application. In this work, we study fault seeding mechanisms based on user interactions with the application; and thus give a guided fault seeding mechanism for the purpose. We show the usefulness of the guided fault seeding with the help of a case study using the blackboard application.**

***Keywords*—fault-seeding, interaction patterns, sequential patterns.**

I. Introduction

Fault seeding is a testing technique to measure the effectiveness of a test suite. It is a white-box testing technique where we induce faults artificially in the system under test (SUT) and see if the test suite is capable of uncovering seeded faults. The testers compare detected seeded faults and undetected seeded faults to calculate a confidence measure [1] of the test suite; and then additional test cases are added to the test suite, if required [2].

One of the issues with the fault seeding approach is the identification of potential areas in the application, where the faults are to be seeded. Ideally, we should be able to identify areas in the application, which are not being executed by any test case in the test suite. It is rather clear that a relatively ineffective fault seeding mechanism would be an overhead than being of any advantage. Thus, 'random' fault seeding may not help as a fault may or may not be seeded in parts of the code which are not being executed by any test case in the test suite.

We argue, that if the intended usage of the application under test could be inferred a-priori, it could help in more effective fault seeding and thus, improving the effectiveness of the test suite and ultimately the testing process. It is rather obvious that a test case execution follows some path in the application and the idea is to detect whether that particular path has a bug or otherwise. The test suite gives a complete list of paths which are being followed during the execution of a test suite. We note that (a) although, testing is a sophisticated process and the test suite is carefully designed to ensure that no areas in the application remain untested or under-tested; it is usually the case that some of the application areas are better tested than the others; and thus whilst parts of the application are very well covered in the test suite, some may require additional testing effort, and (b) the testers' perspective of the application usage could be very different from the intended application users' application usage. Thus, there should be a systematic way to incorporate these two factors in the fault seeding process.

In this paper, we propose a 'guided' fault seeding mechanisms based on the *frequent* (or rather infrequent) paths followed (a) by the test cases in the test suite, and (b) by the application users' interaction with the application. This can give an idea of which parts of the application are of more interest to (a) the tester and (b) the user; consequently, identifying application areas that require further attention. To the best of our knowledge, there is no such study which suggests that whether 'random' fault seeding could be more effective than the proposed 'guided' fault seeding approach, and this is the objective of this study.

We study the effectiveness of a 'guided' fault seeding mechanism in contrast with the 'random' fault seeding approach. We consider two parameters for the purpose, (a) test case execution paths, and (b) user-interactions with the application. We thus record test case execution paths and also the user interactions with the application, and then apply a sequential pattern mining algorithm (see Section II-B for a discussion) to report the frequent (or infrequent) paths both from test cases and user interactions, separately. This information is later used to identify the application areas where the faults are to be seeded. In the empirical evaluation, we study the usefulness of the proposed approach in effective fault seeding.

For the purpose of evaluation, we develop a web-based application which is a Learning Management System (LMS) designed primarily for instructors. The instructors can add assignments, quizzes and lecture slides as well as assessments of students registered for a course. LMS also allows attendance management and result management. In addition to provision of these services, LMS records user interactions with the application as well. This feature is required to extract frequent paths to identify potential application areas for fault-seeding. Each control (form, button, etc.) in the application has a specific ID so that any execution path is uniquely identifiable. During the test case execution or the user interaction with the application, the control IDs are recorded in order by LMS in a database for each user, which are later extracted in a time-order manner to serve as input to the sequential pattern mining algorithm. The sequential pattern mining algorithm reports frequent execution paths which are later used in the 'guided' fault seeding process.

The paper is organized as follows. The background of this work is discussed in Section II, and our proposed approach in presented in Section III. The evaluation of the proposed approach is given in Section IV, whereas the related work is reviewed in Section V. Finally, we present conclusions and outlook in Section VI.

II. Background

We present background of our work in two subsection where the former present background to fault seeding and the later presents data/pattern mining.

A. Fault Seeding

Fault seeding as a technique introduced by Mills (1972) and explained in [1] in which the probability that our test suite is capable of finding faults is established as a ratio between the total seeded and the detected seeded faults during testing. This helps in calculating confidence $C = 1$ if $n > N$ and $S/(S - N + 1)$ if $n \leq N$ where S is the number of seeded faults, N is the total number of non-seeded (indigenous) faults which can be found by $N = Sn/s$ where n is actual number of non-seeded faults and s is the number of seeded faults detected during testing. Fault seeding provides an alternative to structural testing techniques providing us with a measure of sufficiency of testing or effectiveness of our test suite.

While doing fault seeding, we need to know what types of faults can be seeded. We have used the error classification given in [3] in which errors are classified as $domain faults$ and $computation faults$. A domain fault results from control flow errors e.g. missing path (absence of missing conditional statement or clause) or predicate/assignment fault caused by incorrect decision at a predicate whereas computational faults occur when computation is wrong yet the path traversed is correct.

B. Interaction Patterns

Algorithm 1 Sequential pattern mining algorithm

```
1: Input: Interaction sequence database $D$ and support threshold $\theta$.
2: Output: All frequent interaction sequences $s$ with support at least $\theta$.
3: $i \leftarrow 1$
4: $L_1 \leftarrow$ All Frequent sequences of length 1 in D
5: while $L_i \neq \emptyset$ do
6:     $C_{i+1} \leftarrow$ Join $L_i$ with itself
7:     for all $s \in C_{i+1}$ do
8:         Compute Support of $s$ in $D$
9:     end for
10:    $L_{i+1} \leftarrow$ all frequent sequences $s \in C_{i+1}$
11:    $i \leftarrow i + 1$
12: end while
13: Stop and output $L_1 \cup \ldots \cup L_i$
```

We use Sequential Pattern Mining (SPM) to extract 'useful' interaction patterns. Sequential patterns are an important data mining technique [4], [5] implied in a variety of applications primarily to understand users' behaviour. We briefly review the sequential pattern mining problem in this section. A discussion on useful applications of SPM is in Section V-B.

The idea of sequential patterns was first proposed by [4] and the objective was to predict customers' next purchase in a retail environment. Agrawal and Srikant [4] proposed the apriori algorithm to mine sequential patterns, on which they improved later on by introducing a GSP algorithm [6]. In this work, we use a variant of the GSP algorithm which is customized to work with LMS to mine frequent sequences of paths taken by (a) the test cases in the test suite and (2) the users. We first introduce a few notations and then give an overview of the GSP algorithm.

A "click" in LMS is called an *item*. A *sequence* is an ordered list of items. A user sequence U is an ordered list of items (clicks) by the user. A database D is a collection of user sequences. A sequence s *contains* a sequence t if all the items in t appear in s in the same order as in t, although gaps are allowed. For example, a sequence $\langle a, b, c, d, e\rangle$ supports a sequence $\langle b, c, e\rangle$ whereas it does not support a sequence $\langle b, a, d\rangle$. The *support* of a sequence s is the number of user sequences U which contain s. A sequence is frequent if it is supported by at least $0 \leq \theta \leq |U|$ sequences. Input to the problem is a database D of user sequences and a support threshold $0 \leq \theta \leq |U|$. C_i is the set of candidate sequences of length i that needs to be tested for being frequent. L_i is the set of candidate sequences which have support at least θ and thus, are frequent. An overview of the GSP algorithm is in Algorithm 1. We now briefly discuss the working of the GSP algorithm.

In the first step, user interactions are loaded in the form of user interaction sequences. A support threshold θ is also provided by the user. Initially, all the frequent sequences of length 1, L_1, are discovered (Line 4). Then, the set L_1 is used to generate candidate sequences of length 2, C_2, and after support counting L_2 is obtained. For example, two frequent sequences of length 1, $\langle a\rangle$ and $\langle b\rangle$, generate four candidate sequences $\langle aa\rangle$, $\langle ab\rangle$, $\langle ba\rangle$, $\langle bb\rangle$, which are then tested for being frequent. After L_2 is obtained, candidate sequences of length 3 onwards, are generated as follows. For any two sequences s and s', if removing the first item in s and the last item in s' yield the same remaining sequences, resulting candidate sequence t is the sequence s appended with the last item in s' (Line 6). For example, a sequence $\langle ab\rangle$ is joined with $\langle bc\rangle$ to yield a candidate sequence $\langle abc\rangle$, whereas $\langle ab\rangle$ and $\langle ab\rangle$ can not be joined as removing the first item in $\langle ab\rangle$ and the last item in $\langle ab\rangle$ yield $\langle b\rangle$ and $\langle a\rangle$, respectively, which are not the same. Thus, frequent 2 sequences onwards, C_i, $i \geq 3$, are generated by joining $L_i - 1$ with itself, and are then tested for being frequent (Line 8). The set of frequent i sequences L_i (Line 10) is then used to generate C_{i+1} (Line 6) and so on. The mining algorithm continues until no more frequent sequences could be found. The output is the list of all frequent sequences (Line 13).

III. Our Approach

We require user interaction history, as a first step, to do weighted fault seeding. We, therefore, store information about each event resulting from some input or user click. Overall schematic is shown in Fig 1

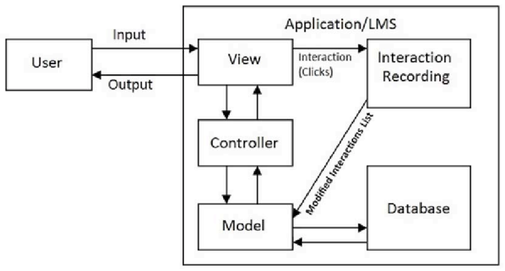

Fig. 1: interaction pattern graph resulting from interaction data collection

In order to record which interactions have taken place and in order to construct a graph of events, we assign every button in the application a specific ID. We consider each form separately as input and produce paths of each component in that form as output. In our case, we are using JSF form components, therefore we convert this algorithm according to JSF provided methods to extract the graph. Initially, the algorithm picks up single web page 'P' as input and extracts all Children components. Then this algorithm performs a recursive loop until it gets sub Children of each child 'c' from Children of P. During this recursion process it adds complete path of each component into a list L. JSF provides a special method to extract the complete path of the component. At the end, the algorithm returns L containing graph of all components of the given form.

When user clicks on a button within the application, interaction ID is recoded and at the end of session when user logs out from their account, the complete interaction by the user is stored in a time-ordered manner. Every signed-in user has his own interaction recording list managed by the application such that IDs stored in database represent different paths followed by a single user, also called interaction patterns. This is shown in Listing 1.

Listing 1: Hash Values for Attributes Lists

```
    ...
    ArrayList<GraphObject> l =
    new ArrayList<GraphObject>(m.getCodomainObjects());
    for (int i=0; i<l.size(); i++)
    {
        if (l.get(i).isNode() == true)
        {
            List<String> result = new ArrayList<String>();
            int hCode = l.get(i).hashCode();
            result.add(Integer.toString(hCode));
            Node n = (Node) l.get(i);
            for (int j=0; j<n.getNumberOfAttributes(); j++)
                result.add(Integer.toString
                (n.getAttribute().hashCode()))
   ...
```

Once the interaction related information is acquired, we represent this information as a graph $G = (V, E)$ where set of vertices V is the set of controls such as buttons, lists, forms, etc. and set of edges E denotes the control flow information. This provides us with a graph like structure as shown in Figure 2.

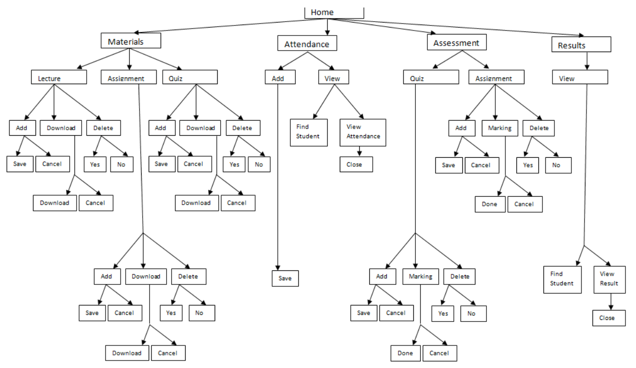

Fig. 2: Interaction pattern graph resulting from interaction data collection

In order to find frequency of interactions, we record how many times a particular control sequence was adopted during a user interaction. This provides us with frequent paths for the application considering a free choice given to the users. However, in order to provide them a first level of assistance, user scenarios are developed as part of testing done for the SUT. We present our interaction pattern extraction mechanism in Algorithm 2.

Algorithm 2 Regression Analysis

```
Require: URL of the web service to be tested, test input data
  and oracle from previous run
  for (i=0; i<num(operations); i++) do
    if test passed, result saved in the database then
      if actual output from current run = expected output read
      from previous run then
        test passed, no regression
      else
        test failed
      end if
    end if
  end for
```

Recall that whilst doing fault seeding, we need to know the types of faults to be seeded. As already discussed, we have used error classification given in [3] in which faults are classified as $domain faults$ and $computation faults$. We have seeded faults according to the error distribution given in [7] and is shown in Table I.

TABLE I: Fault Distribution

Class of Errors	%age
Computational	13%
Initialization	13%
Logic/Control	32%
Interface	18%
Data	24%

We use $IEEE$ fault severity definitions as given in [7] shown in Table II and, in order to avoid system crashes, we only seed faults of types 3 to 5. Faults could be introduced in the SUT in a number of ways. Radome fault seeding [8], through mutation operators in isolated manner [9], through expert human seeder [2] are examples of fault seeding methods. We use random fault seeding for the purpose of evaluation of the usefulness of guided fault seeding.

TABLE II: Fault Severity

Severity	Guidelines
1	Catastrophic–*Causes system failure or crash*
2	Major–*Makes the product not useable*
3	Moderate–*Product usable but bug is customer affecting*
4	Minor–*Product usable and bug non-affecting*
5	Nuisance–*Nature of the fault is such that it can be repaired anytime*

Finally, we apply the fault seeding approach considering the interaction frequencies such that the fault seeding process follows the same percentage of overall faults as the interactions frequencies. We present evaluation of guided fault seeding, the results we obtain and threats to validity of the results in the forthcoming section.

IV. Evaluation

We develop a web application, as discussed earlier, for the purpose of evaluation. Since we intend to evaluate if the guided fault seeding introduced in this paper, is more effective than the random fault seeding, we take the following steps:

1) improve test suite until confidence approaches 100%
2) collect frequency of paths in the usage data
3) apply fault seeding considering frequent paths data
4) run test suite and calculate confidence

The overall process followed for evaluation is shown in Figure 3.

In the first step, faults are seeded randomly to the application and test cases are added into the test suite to achieve a confidence approaching 100%. We process the usage frequency of different paths in the application for the purpose. We conduct experiments with university students where they are requested to use LMS. We introduce LMS to the students by a brief overview about the application. The users (students in this study) are familiar with software testing as they have done modules like software quality engineering, software testing and software quality assurance. The users are encouraged to report any errors during the application usage and the reported errors are recorded. The idea is to obtain the information about the frequency of different paths in the application.

We solicit this information as follows. For the first group of users, we let users to use the application in an exploratory manner. Although we need test suite to evaluate random fault seeding but we need this experiment for the extraction of frequent path as well as to get users' interest in using application. We informed them about presence of faults before testing to engage their interest to catch faults as it was given as a challenge to them. Another advantage of exploratory application usage is in better understanding of the application by the user.

For the second set of users, we ask user to use test cases for the purpose of testing. However, a user could choose the

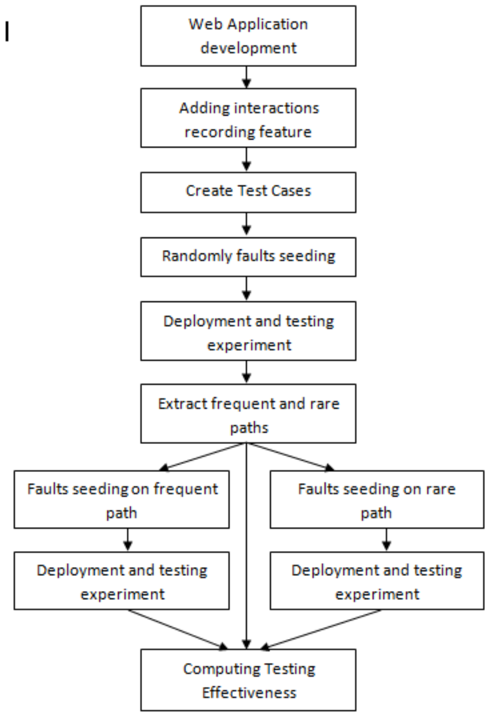
Web Application development
Adding interactions recording feature
Create Test Cases
Randomly faults seeding
Deployment and testing experiment
Extract frequent and rare paths
Faults seeding on frequent path
Faults seeding on rare path
Deployment and testing experiment
Deployment and testing experiment
Computing Testing Effectiveness

Fig. 3: process diagram of our evaluation process

module that they want to test. We divide the test suite into different subsets according to LMS modules, i.e. a subset of test cases group for attendance system, a subset of test cases for lecture management, etc. We repeat the set of experiments with different student groups in order to obtain a reasonable usage dataset.

Finally, we request student advisers and support staff in departments within faculty of engineering sciences[1] and we update our frequency of usage data through our second set of data. We use both sets of data for the evaluation. We also keep an eye on the outcome of application usage to adjust our test cases as well as to seed more faults on the application paths where we are not seeding seeded faults. The results are presented in Table III

TABLE III: Random Seeding Data

item description	value in #
Total test cases	61
Total seeded faults	52
Seeded detected faults	40
Unseeded detected faults	28
Seeded undetected faults	12
New test cases Added for undetected seeded faults	13
Total Test Cases in Updated Suit	74

Now we are ready to perform our experiment which is to test if guided fault seeding performs better than random fault seeding. We are guaranteed an outcome since we have test suite with a confidence 96% with random fault seeding and if we seed more faults around paths with more (or less) usage

[1]This research was conducted in Bahria University Islamabad Campus, Pakistan

frequency, the confidence should not drop and should remain same. This would mean that the test cases should be able to find all freshly seeded faults. In case, the confidence drops down, we find out those areas that have not been executed by test cases previously and we have eliminated possibilities that a path is completely free of seeded faults in our discussion above. All randomly seeded faults are first order faults [10] whereas faults on frequent and infrequent paths are seeded in second order because of limited area of code. We present statistics of fault seeding along frequent paths step in Table IV.

TABLE IV: Guided (frequent) Seeding Data

item description	value in #
Total test cases	74
Total seeded faults	52
Seeded detected faults	44
Unseeded detected faults	29
Seeded undetected faults	8

The confidence for this experiment, where we consider frequent paths, drops down to 59.8%. Next, we consider weighted faults seeding approach discussed above and the faults seeded following infrequent paths. We apply the same test suite and report number of faults escaped our test suite. We report change in the confidence which was previously at 96% as shown in Table V.

TABLE V: Infrequent testing step data

item description	value in #
Total test cases	74
Total seeded faults	52
Seeded detected faults	48
Unseeded detected faults	28
Seeded undetected faults	4

The confidence for infrequent path fault seeding experiment drops down to 59.8%. The drop in the confidence measure due to more seeded faults escaping from the test suite shows limitations of random fault seeding approach. Considering table IV and table V, it is evident that there were more seeded undetected faults which escaped in the former case. The number of test cases in both the cases are 74 which is due to the fact that we do not intend to add or delete test cases but we want to see if presence of additional faults would dent confidence in any way. We have added twelve faults along (in)frequent paths as shown in table IV and table V. Finally, We present limitations of our approach. The algorithm presented in preceding section processes graph resulting from user interactions. This graph is directly dependent upon the number of controls and forms present in an application and hence the application size would dictate the processing time of the algorithm bearing a quadratic complexity. The experiment itself needs more interaction data which would do further conditioning of the results. The results are not stunning in the sense random fault seeding is already known for compromises in its own entirety. We, however, propose a novel approach to fault seeding.

V. Related Work

We first review relevant studies for fault seeding and we subsequently discuss applications of sequential pattern mining.

A. Fault seeding

Fault seeding is an effective technique to access testing effectiveness by measuring test coverage [11] where faults are artificially injected into the application and programmer keeps track of seeded faults during testing. A difficulty in this technique is that injected faults must be representative of the unseeded non-discovered faults [12]. Random fault seeding is an approach proposed and used in a number of research e.g., in [11],[13] where faults are seeded randomly on random location of application. Later these faults are detected by software testers and developer using test suite. On the basis of undetected seeded faults, which are not identified by test suite, we evaluate unseeded remaining faults in application.

Fault seeding through mutation operators in isolated manner is proposed in [14] where a faulty/mutated program is produced by changing original program by introducing faults using mutation operators in program. Mutant force the program to generate different output then original program. The tester, then kill the mutant using test suite, only if successful. Fault seeding through expert human seeder is proposed in [15] the faults are seeded based upon knowledge of programming language and system nature by the expert. Fault seeding is also used in [16] where fault seeding is used as a means to evaluate quality of test suite and as a means to evaluate proposed methodology. A comparison and an evaluation of fault seeding techniques is conducted in several research articles e.g., in [17] where the authors have conducted research to see effectiveness of various fault seeding methods.

B. SPM applications

Sequential patterns have been used in a variety of application domains. For example, in retail environments, sequential patterns have been used to predict future customer purchases [4], for recommending products [18], for shelf management, etc. Similarly, in stock markets, for predicting future trends; for fault analysis, in detecting the events which lead to a failure; in education for examining JAVA code dependencies [19]; in information retrieval for document categorization [20] and so on. One interesting class of applications of sequential patterns is in bio-informatics,e.g. in protein fold recognition [21], [22], protein function prediction [22], analysing gene expression data [23], etc. See [24] for a detailed discussion on applications of SPM.

We now briefly review a couple of useful applications of sequential pattern mining which are more relevant to this work in methodology.

In bio-informatics, one such work is on protein function prediction using sequential patterns [22]. The proposed approach works in two phases. First, frequent sequences are mined from a known protein dataset. Next a classifier is used to predict the function of the sequence. The effectiveness of the proposed approach is demonstrated with the help of experiments.

Sequential patterns have been used to design recommendation systems for web users by mining users' web usage data [18]. The idea is to find frequent navigation paths and then suggest the next page that (a) would be accessed by the user or (b) would be of interest to the user; in order to perform tasks like web page pre-fetching ((a) above) to improve users' navigation experience, or recommending products of interest to the user ((b) above) to improve users browsing experience and ultimately the sales.

VI. Conclusion

Fault seeding is a confidence based technique, as opposed to structural testing techniques, at white-box level in which the probability that our test suite is capable of finding faults is established. This is done by running the application with seeded faults and finding a ratio between total seeded and detected seeded faults during testing.

We propose a technique where we gather usage data from an application to find frequent (or infrequent) paths in the application. We then compare guided fault seeding with random fault seeding technique to see the effectiveness of guided fault seeding. Guided fault seeding is original in the sense that we not only propose a new approach to fault seeding but we also evaluate effectiveness of our approach.

Guided fault seeding clearly shows that an analysis of usage pattern would be more effective to uncover faults but with prior knowledge of frequency of usage through the application. As an outlook, we plan to run experiments with more test data and with large scale applications to see scalability of our approach.

References

[1] S. L. Pfleeger and J. M. Atlee, *Software engineering - theory and practice (4. ed.).* Pearson Education, 2009.

[2] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[3] W. Howden, "Reliability of the path analysis testing strategy," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 3, pp. 208 – 215, sept. 1976.

[4] R. Agrawal and R. Srikant, "Mining sequential patterns," in *ICDE*, P. S. Yu and A. L. P. Chen, Eds. IEEE Computer Society, 1995, pp. 3–14.

[5] M. Muzammal and R. Raman, "On probabilistic models for uncertain sequential pattern mining," in *ADMA (1)*, ser. LNCS, L. Cao, Y. Feng, and J. Zhong, Eds., vol. 6440. Springer, 2010, pp. 60–72.

[6] P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, Eds., *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, ser. LNCS, vol. 1057. Springer, 1996.

[7] F. Grigorjev, N. Lascano, and J. L. Staude, "A fault seeding experience. argentina: *Motorola Global Software Group*."

[8] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *ISSTA*, 1996, pp. 195–200.

[9] K. S. H. T. Wah, "Fault coupling in finite bijective functions," *Softw. Test., Verif. Reliab.*, vol. 5, no. 1, pp. 3–47, 1995.

[10] Y. Singh, *Software testing*. New Delhi: Cambridge university, c2012.

[11] N. Lascano and J. L. S. F. Grigorjev, "A fault seeding experience," in *ASSE-2003*, 2003.

[12] B. Boehm and D. Port, "Defect and fault seeding in dependability benchmarking," in *In Proc. of the DSN Workshop on Dependability Benchmarking*, 2002, pp. Washington, D.C.

[13] J. H. H. A. J. Offutt, "A semantic model of program faults," in *in ACM SIGSOFT, 1996*, 195-200.

[14] K. S. H. T. Wah, "Fault coupling in finite bijective functions," in *Science of Computer Programming, vol. 5, no. 1, 1995*, 3-47.

[15] T. K. Tsai and R. K. I. M.-C. Hsueh, "Fault injection techniques and tools," in *Computer, vol. 30, no. 4, April 1997*, 75-82.

[16] T. A. Khan and R. Heckel, "On model-based regression testing of web-services using dependency analysis of visual contracts," in *FASE*, 2011, pp. 341–355.

[17] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[18] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa, "Discovery and evaluation of aggregate usage profiles for web personalization," *Data Mining and Knowledge Discovery*, vol. 6, no. 1, pp. 61–82, 2002.

[19] T. Ishio, H. Date, T. Miyake, and K. Inoue, "Mining coding patterns to detect crosscutting concerns in Java programs," in *WCRE*, A. E. Hassan, A. Zaidman, and M. D. Penta, Eds. IEEE, 2008, pp. 123–132.

[20] S. Jaillet, A. Laurent, and M. Teisseire, "Sequential patterns for text categorization," *Intelligent Data Analysis*, vol. 10, no. 3, pp. 199–214, 2006.

[21] K. Wang, Y. Xu, and J. X. Yu, "Scalable sequential pattern mining for biological sequences," in *CIKM*, D. A. Grossman, L. Gravano, C. Zhai, O. Herzog, and D. A. Evans, Eds. ACM, 2004, pp. 178–187.

[22] M. Wang, X. Shang, and Z. Li, "Sequential pattern mining for protein function prediction," in *ADMA*, ser. LNCS, C. Tang, C. X. Ling, X. Zhou, N. Cercone, and X. Li, Eds., vol. 5139. Springer, 2008, pp. 652–658.

[23] S. F. Hussain, "Bi-clustering gene expression data using co-similarity," in *ADMA (1)*, ser. Lecture Notes in Computer Science, J. Tang, I. King, L. Chen, and J. Wang, Eds., vol. 7120. Springer, 2011, pp. 190–200.

[24] M. Gupta and J. Han, "Applications of pattern discovery using sequential data mining," in *Pattern Discovery Using Sequence Data Mining: Applications and Studies*, P. Kumar, P. R. Krishna, and S. B. Raju, Eds. IGI Global, 2012, ch. 1, pp. 1–23.