Randomization for Testing Systems of Systems

Qianhui Liang¹ and Stuart H. Rubin²

¹*School of Information Systems
Singapore Management University
Singapore 178902*

²*SPAWAR Systems Center (SSC)
Code 56340, 53560 Hull Street, San Diego
CA 92152-5001, USA
althealiang@smu.edu.sg, stuart.rubin@navy.mil*

Abstract

This paper applies randomization theory to the problem of selecting software test cases for software systems and applications in order to overcome the hurdle of high cost in testing componentized Systems of Systems (SoS). We have used a corner point semantics, which can approximate a proof of correctness – termed a pseudo-proof of correctness. Test cases for each component are designed to be mutually orthogonal, or randomized. Integration testing is performed through a composition of the test cases for components with some value-added test cases to cover integration aspects of the system. Integration testing is also designed in such a way that the testing algorithm is written in randomized form. The advantages offered by such randomization are ever present in the algorithm, programming language, integration, and workflow design.

Keywords: Randomization, software testing, systems of systems, test cases, component testing, integration testing

1. Introduction

In general, software testing can be expensive. An informal survey of software manufacturers in industries including telecommunications, computers, automotive and financial indicated that testing consumes 30 to 70 percent of the resources in the software development life cycle. One reason for the costly testing is the combinatorially explosive number of execution paths of a piece of software. For more complex software systems such as Systems of Systems (SoS), their behavior is even more complicated. Although testing can never be 100 percent exhaustive, software testing is meant to have as high coverage of execution paths as possible. Thus, it is imperative to lower the cost of software development, to minimize the product introduction delays, and to allow fewer faults to remain in the product. In particular, the cost of software testing has become one of the major

hurdles to reducing the time and monetary expenditure in the software development life cycle.

Nowadays, more of the software systems and applications are designed with the criteria of well-componentization in mind for the purpose of easy and effective reuse in the future by other software. Complex Systems of Systems (SoSs) are examples of such software systems. This design trend has been further supported by the emerging integration paradigms like Service Oriented Architecture (SOA). Componentization in SoSs can be used to help bring down the cost of software testing, as some reusable components or sub-systems should have already gone through testing before they have been used in previous applications. Furthermore, due to componentization, test cases may be designed to be orthogonal such that more execution paths can be covered. Moreover, integration testing at multiple levels creates more opportunity for improving the effectiveness of software tests by randomizing the testing to reduce its complexity, which also cuts down on the cost of testing.

In this paper, we apply the randomization theory to the problem of designing software test cases for complex Systems of Systems (SoSs) in order to overcome the hurdle of high cost in testing componentized software. We have used a corner point semantics, which can approximate a proof of correctness - termed a *pseudo-proof of correctness*. Test cases for each component are designed to be mutually orthogonal, or randomized. Integration testing is a composition of the test cases of components with some value-added testing cases to cover integration-oriented aspects of the software. Integration testing is also designed in such a way that the testing algorithm is written in randomized form. The advantages afforded by such randomization are ever present in the algorithm, programming language, integration, and workflow design.

The remainder of this paper is organized as follows. In section 2, we review randomization and how it can be applied to testing componentized software. In section 3, we describe the framework of randomized software

testing. In section 4, we walk through the process of designing test cases for a software system under our development by using randomization. We conclude the paper in section 5.

2. Randomizing software testing

Randomization theory was originally developed by Chaitin and Kolmogorov [1] and may be defined by any string whose size in bits is roughly the same as any computer program capable of outputting it. The Unsolvability of the Randomization Problem [10] is defined as follows: the complexity of a particular string cannot be proven in a formal system that is less complex than the program producing it. In other words, given a formal system, a formal system of greater complexity is needed to prove it random. Heuristics are search randomizations in space-time and such randomizations are proven to reduce the evolutionary time for heuristics in the large [2]. Domain-specific knowledge or local inductive randomization is necessary in learning and conducting a proof when global or domain-general proofs are impossible [2][3][4]. Randomization has also been proved to provide savings in composing Web-service-oriented software systems [5] and in theory-based verification of workflows [7].

We can use a corner point semantics, which can approximate a proof of correctness – again, termed a pseudo-proof of correctness. That is, the proof of algorithm/program correctness will stem from the compositional testing of its System of Systems (SoS); where, each system (and SoSs) will be tested using functional random-basis testing. Here, test cases are designed to be mutually orthogonal (i.e., to the extent practical) and are thus said to be randomized. Such randomization allows for the coverage of a maximal number of execution paths. The percentage of paths so covered bears proportion to the degree of randomization. As a consequence, integration testing will capture many more such paths, but not all. Even more paths are tested if the algorithm/program is written in randomized form, but again not provably all of them, in the general case. For example, a more exhaustive test will result from the use of parametrized subroutine calls in lieu of multiple expansions of in-line code because there will be a proportionately greater frequency of execution of proportionately less code here.

Pseudo-proofs of correctness for non-trivial codes are necessarily recursively enumerable - not recursive. They are eminently practical because the constraints on optimality are relaxed. Finally, randomized algorithms/code can suggest novel algorithmic/coding methodologies by making latent symmetries far more evident and even formalizable. Truly, the advantages afforded by randomization are ever-present - in algorithm,

programming language, and program design; in testing, integration, workflow design, and so on.

3. Framework

In this section, we describe in detail the framework for randomized software testing for componentized software systems or applications.

Figure 1. Framework for Randomized Software Testing

Figure 1 shows the general framework for randomized software functional testing. We can see from the figure that such testing consists of two parts: randomized component testing and randomized integration testing. Component testing and integration testing can be studied similarly in a recursive way. We describe two parts in detail in the remainder of this section.

3.1. Randomized component testing

Here, test cases are generated over 1) the *input parameters* of the component or system to be tested, and 2) the possible *value ranges* that each parameter can take. Input parameters may include the arguments of the command line entered from the terminal or read from a configuration file, the state of an interface such as the values of the global variables and constants, input from a connected device, or the initialization of local variables. Different values of input parameters may cause different behaviors of the component or system. Thus, the values are grouped according to their impact on system behavior. These groups do not overlap and each group can be represented by a single value. For example, if the logic of the system does not change for a parameter's value ranging from 100 to 999, the values of 100 to 999 can be grouped together for this parameter and be represented by

a single value (e.g. 100). Table 1 shows an example of a system with four input parameters, each having three distinct value ranges.

Table 1: Input parameters and value ranges

Input parameters	Value ranges1	Value ranges2	Value ranges3
A	A1	A1	A3
B	B1	B2	B3
C	C1	C2	C3
D	D1	D2	D3

To obtain the above table implies the need for certain knowledge of the system specification and of what it is expected to behave like from a user point of view. Generally, new knowledge of the code is required. Given the above table, each functional test case can then be generated by the following procedure:

```

for each input parameter {
    pick a value range;
    take the representative value of the range;
}
form a test case;

```

One way to get the test cases is to exhaustively generate all possible test cases by repeating the above procedure. However, this is not practical in most cases; although, this will have 100% coverage of the code. We now define randomization in selecting test cases. Randomization of test cases is defined based on the input parameters. In particular, randomization is performed for a certain degree of input parameter combinations.

Definition 1: A randomization of component testing is said to have a certain degree if it is applied to input parameter combinations with that number or less of distinctive input parameters.

For example, the least degree possible for randomization is one, meaning that randomization is only applied to individual parameters.

In the example listed in Table 1, randomization of degree one results in a set of cases that test whether each single parameter value has caused error. The next degree of randomization, or 2d degree randomization, in addition to generating test cases for one single parameter value, also generates test cases corresponding to the errors caused by the concurrence of two specific values for two input parameters, respectively. Similarly, a 3d degree of randomization can also be applied. In the example we use, 3d degree of randomization is the highest degree of randomization.

First degree randomization and second degree randomization of component testing are listed in table 2. Rows 2 through 10 show the results of first degree randomization and rows 11 through 19 show the results of 2d degree randomization. We have followed the Orthogonal Array (OA) approach (OA L9) proposed by

Taguchi [8] and suggested to be applied for software testing by [9] for the detailed selection method.

Table 2: Randomization of component testing

Input parameters	1 st (2d) degree randomization for component testing
A	A1, B1, C1, D1
	A2, B1, C1, D1
	A3, B1, C1, D1
B	A1, B2, C1, D1
	A1, B3, C1, D1
C	A1, B1, C2, D1
	A1, B1, C3, D1
D	A1, B1, C1, D2
	A1, B1, C1, D3
AB, AC, AD, BC, BD, CD	A1, B1, C1, D1
	A1, B2, C2, D2
	A1, B3, C3, D3
	A2, B1, C2, D3
	A2, B2, C3, D1
	A2, B3, C1, D2
	A3, B1, C3, D2
	A3, B2, C1, D3
	A3, B3, C2, D1

As the randomization degree of testing increases, the possible scenarios where the system may be broken are more exhaustive. In other words, in the case of software testing, a higher degree of randomization represents a higher coverage of execution paths in the system, and thus will allow the detection of a greater number of faults. This is consistent with the fact that lower randomization contains a relative higher portion of repetitive information, which results in a coarser grain of testing. In contrast, further randomization removes the repetitive information and thus results in test cases that approach the true behavior of the system. This results in better testing results.

3.2. Randomized integration testing

In this subsection, we describe randomized integration testing based on randomized component testing.

Test cases in integration testing must include the randomized test cases for all its components as a form of regression testing. This is to make sure that any old bugs embedded in the components do not reappear in the integrated system and no new bugs are introduced into the components by integration.

The presence of multiple components in a system to be tested using system integration implies that randomization can be applied to generate more test cases in addition to the test cases for each individual component. The idea is that we can identify all the components of the system, identify all the test cases for each component, and perform randomization just as we

do with the table of input parameters and value ranges in the case of component testing. Table 3 shows an example of a system with three components, where each component has four input parameters, and three value ranges and has undergone 2d degree randomization.

Table 3: Components and randomized test cases

Components	Component C(1)	Component C(2)	Component C(3)
Component test cases 1	A1,B1, C1, D1	A'1,B'1, C'1, D'1	A''1,B''1, C''1, D''1
Component test cases 2	A1, B2, C2, D2	A'1, B'2, C'2, D'2	A''1, B''2, C''2, D''2
Component test cases 3	A1, B3, C3, D3	A'1, B'3, C'3, D'3	A''1, B''3, C''3, D''3
Component test cases 4	A2, B1, C2, D3	A'2, B'1, C'2, D'3	A''2, B''1, C''2, D''3
Component test cases 5	A2, B2, C3, D1	A'2, B'2, C'3, D'1	A''2, B''2, C''3, D''1
Component test cases 6	A2, B3, C1, D2	A'2, B'3, C'1, D'2	A''2, B''3, C''1, D''2
Component test cases 7	A3, B1, C3, D2	A'3, B'1, C'3, D'2	A''3, B''1, C''3, D''2
Component test cases 8	A3, B2, C1, D3	A'3, B'2, C'1, D'3	A''3, B''2, C''1, D''3
Component test cases 9	A3, B3, C2, D1	A'3, B'3, C'2, D'1	A''3, B''3, C''2, D''1

We can define the degree of randomization in integration testing in a similar way to that in randomizing component testing.

Definition 2: A randomization of integration testing is said to have a certain degree if it is applied to combinations of component test cases with that number or less of distinctive components.

The first 12 test cases generated by the second degree randomization of integration testing are listed in table 4.

4. Case study

Our case study detailed component testing of a PC-based automated negotiation engine called ServiceNegotiator for (Web) service-based SoS. At first, we had prepared 800 test cases, which may take a full-time research engineer more than one week to finish. In view of the change of plan, the submission date was moved forward. We were not sure, within the limited time, how many test cases from the original design we were able to finish and which test cases we should choose in order to detect as many faults as possible. We prepared a new test plan using randomization of component testing and the integration testing described above to ameliorate the testing stress.

Table 4. Randomization of integration testing

Input parameters	2d degree randomization of integration testing
C(1)C(2), C(1)C(3), C(1)C(4), C(2)C(3), C(2)C(4), C(3)C(4)	A1,B1, C1, D1, A'1,B'1, C'1, D'1, A''1,B''1, C''1, D''1
	A1,B1, C1, D1, A'1, B'2, C'2, D'2, A''1, B''2, C''2, D''2
	A1,B1, C1, D1, A'1, B'3, C'3, D'3, A''1, B''3, C''3, D''3
	A1,B1, C1, D1, A'2, B'1, C'2, D'3, A''2, B''1, C''2, D''3
	A1,B1, C1, D1, A'2, B'2, C'3, D'1, A''2, B''2, C''3, D''1
	A1,B1, C1, D1, A'2, B'3, C'1, D'2, A''2, B''3, C''1, D''2
	A1,B1, C1, D1, A'3, B'1, C'3, D'2, A''3, B''1, C''3, D''2
	A1,B1, C1, D1, A'3, B'2, C'1, D'3, A''3, B''2, C''1, D''3
	A1,B1, C1, D1, A'3, B'3, C'2, D'1, A''3, B''3, C''2, D''1
	A1, B2, C2, D2, A'3, B'3, C'2, D'1, A''1, B''2, C''2, D''2
	A1, B2, C2, D2, A'1, B'2, C'2, D'2, A''1, B''3, C''3, D''3
	A1, B2, C2, D2, A'1, B'3, C'3, D'3, A''2, B''1, C''2, D''3
	...

We first divide the functions of the system into several categories. Each category is considered as a component. Input parameters and value ranges were found for each component. Randomization of component testing is used to find test cases for component testing. For example, listed in table 5 are the input parameters and value ranges for a key component (findUtility), which returns the best weighted utility when the available resource is increased or decreased.

Table 5: Input parameters and value ranges for the component of findUtility

Input parameters	Value ranges1	Value ranges2	Value ranges3
Current resource	Lowest	In the middle	Highest
Current utility	Lowest	In the middle	Highest
Current weight	Lowest	In the middle	Highest
Input source	From file	Entered from terminal	From caller
Function	Decrease	Increase	
Utility type	Int	Float	Double
Jdk version	1.5	1.6	1.7
Scale of systems	Small	Medium	Large
Service choices	Limited	Medium	Rich

In this case, we have applied 2d degree randomization for component testing. In total, we have generated 400 test cases to cover the code of the component. We have identified 89 faults using these test cases during our testing.

We have saved on testing 60 percent of the test cases by using the method for randomization of component testing when compared with the original test plan.

5. Conclusion

In this paper, we showed how randomization can be applied to testing componentized SoSs by identifying the orthogonal aspects of the systems so as to heuristically generate test cases that can effectively cover as many execution paths of the code as possible. Two forms of such randomization are discussed, i.e. randomization of component testing and randomization of integration testing. We also showed the effectiveness of such randomization in reducing testing cost by comparing the number of test cases originally generated and generated using randomization.

The advantages afforded by such randomization are ever present in the algorithm, programming language, integration, and workflow design. As part of the future work, we are in the process of proving the advantages of randomization in the above-listed domains.

Acknowledgments

The authors gratefully acknowledge the support of SSC-PAC for providing funding for this research under DTRA grant BA08MSB008.

References

- [1] G.J. Chaitin, "Randomness and Mathematical Proof," *Sci. Amer.*, 232 (5), pp. 47-52, 1975.
- [2] S.H. Rubin, Shu-Ching Chen, J.B. Law, and G.K. Lee, "On the inherent necessity of heuristic proofs," *2005 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4, pp. 3890-3896, 2005.
- [3] Q. Liang and S.H. Rubin, "Randomized Local Extrema for Heuristic Selection in TSP," *IEEE International Conference on Information Reuse and Integration*, pp. 336 - 340, 2006.
- [4] S.H. Rubin, "On randomization and discovery," *Information Sciences*, 177(1) pp. 170-191, 2007.
- [5] Q. Liang and S.H. Rubin "Randomization in Searching for Composites of Software Components," *IEEE International Conference on Information Reuse and Integration*, pp. 42 - 48, 2007.
- [6] W. Sadiq and M.E. Orlowska, "Analyzing process models using graph reduction techniques," *Information Systems*, 25(2), pp.117-134, 2000.
- [7] Q. Liang and S.H. Rubin "Rule Randomization for Propositional Logic-based Workflow Verification," *IEEE International Conference on Information Reuse and Integration*, pp. 374 - 379, 2008.
- [8] Taguchi, Genichi, "System of Experimental Design," Don Clausing, ed., UNIPUB/Kraus International Publications, New York, Vols.1, 2, 1987.
- [9] Tatsumi, Keizo, "Test Case Design Support System," *Proceedings of ICQC*, Tokyo, 1987.
- [10] S.H. Rubin, "Computing with words," *IEEE Transactions on Systems, Man, and Cybernetics, Part B* 29(4), page 518-524, 1999.