Applying Random Testing to Constrained Interaction Testing

Yasuhiro Hirasaki
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
Email: y-hirask@ist.osaka-u.ac.jp

Hideharu Kojima
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
Email: hkojima@ist.osaka-u.ac.jp

Tatsuhiro Tsuchiya
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
Email: t-tutiya@osaka-u.ac.jp

Abstract—This paper discusses interaction testing using random testing. Random testing can generate test cases very fast; but directly applying this method to interaction testing may result in insufficient interaction coverage if constraints exist over parameter values. We propose a random testing approach that is tailored to constrained interaction testing to solve this problem. In this approach, if a test case that was randomly generated violates the constraints, then new test cases are systematically generated to compensate the loss in interaction coverage that would be caused by simply discarding that constraint-violating test case. The technical challenge here is how to reduce the number of those newly generated test cases. We propose a novel algorithm for this purpose and two methods that can be incorporated in the algorithm. Experimental results show that the proposed approach can generate test suites very fast and that the proposed two methods work very effectively in reducing the number of additional test cases.

Keywords—Random testing, combinatorial interaction testing, constraints

I. INTRODUCTION

Recently, software systems have increasingly become highly configurable, making testing all combinations of parameters unrealistic. Combinatorial Interaction Testing (CIT) is known as an effective testing technique that can be used to tackle this combination explosion problem [1], [2], [3], [4], [5]. A common CIT approach is $t$-way testing which requires covering all interactions involving $t$ parameters, where $t$ is an integer chosen by the tester, typically two or three. The effectiveness of this technique is backed by the belief and fact that many faults can be triggered by interactions involving a small number of parameters [6], [7]. On the other hand, CIT has a drawback of large execution time required to create a test suite. Especially when the system under test (SUT) has many parameters, current test case generation algorithms may take long time to produce a test suite.

Random testing has started gaining attention as an alternative to combinatorial interaction testing, as it can overcome this CIT’s drawback. Because of its simplicity, random testing can generate test cases very fast. Recently, Arcuri and Briand showed that when using random testing, any $t$-way interaction is tested with at least probability 63% even if the test suite size is the same as the size of the smallest test suite for $t$-way testing [8].

However, when there are constraints over the input space of the SUT, naively using random testing can result in insufficient interaction coverage. This occurs because constraints may make some particular interactions hard to kill and simply discarding constraint-violating test cases may leave these die-hard interactions untested. An illustrative example will be shown in the next section.

Our work addresses this problem, aiming at making use of the efficiency in test case generation in the presence of constraints while maintaining the probabilistic guarantee of interaction coverage. In our approach, if a randomly generated test case violates the constraints, then the test case is replaced with a collection of test cases that are systematically generated to compensate the loss in interaction coverage that would be caused by simply discarding that constraint-violating test case. The challenge here is how to reduce the number of those newly generated test cases. We devise a novel algorithm for this purpose as well as two methods that can be incorporated with the proposed algorithm. We conduct an experiment to evaluate the performance of the proposed approach.

The remainder of this paper is organized as follows. Section II describes interaction testing, $t$-way testing, a common strategy of CIT, and random testing. Section III describes the overview of the proposed approach. Section IV describes the algorithm that generates a collection of test cases from a constraint-violating test case. Two methods are also presented that can be incorporated into the algorithm. Section V shows the results of an experiment. Section VI describes the threats to validity of the results. Section VII concludes this paper.

II. BACKGROUND

A. Model of SUT

Here we describe the model of an SUT. This model is commonly used in the literature on CIT. The SUT consists of a finite set of parameters and a finite set of constraints. We let $k$ denote the number of parameters. The $i$th parameter has a finite domain $M_i$ of possible values. We assume that parameters are arranged in descending order of the size of the domains. A test case is a value assignment to the parameters. In other words, a test case is an element of $M_1 \times M_2 \times \ldots \times M_k$; that is, a vector $(v_1, v_2, \ldots, v_k)$ such that $v_i \in M_i$. A test suite is a collection of test cases.
We often need to discuss incomplete test cases where values are not specified for some parameters. Following the terminology of [4], we call such an incomplete test case a value schema. We use symbol ‘*’ to represent that a value is not specified for a parameter and call this symbol a wildcard. Hence a value schema is formally defined as an element of \( M_1 \cup \{*\} \times \ldots \times M_k \cup \{*\} \).

### TABLE I. MODEL OF AN SUT

<table>
<thead>
<tr>
<th>Parameters and domains</th>
<th>OS</th>
<th>Browser</th>
<th>Protocol</th>
<th>DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 7</td>
<td>IE</td>
<td>IPv4</td>
<td>MySQL</td>
<td></td>
</tr>
<tr>
<td>Windows 8</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Sybase</td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>If OS is Linux, then Browser is not IE.</td>
</tr>
</tbody>
</table>

For example, consider the SUT described in Table I. This example is adopted from [9]. Here there are four parameters, i.e., OS, Browser, Protocol and DBMS. As shown in this table, these parameters take either three or two values. The vector of these values, each for one parameter, is a test case. For example, (Windows 7, Firefox, IPv4, Oracle) is a test case. A value schema can have a wild card. For example, (Windows 7, *, IPv4, *) is a value schema. Note that any test case is also a value schema.

Each test case must satisfy all given constraints to be executed. A test case is said to be valid if it satisfies all the constraints; it is invalid, otherwise. Formally a set of constraints is represented as a mapping \( c : M_1 \times M_2 \times \ldots \times M_k \rightarrow \{true, false\} \) such that \( c(v) = true \) if and only if test case \( v \) is valid. This notion can be naturally generalized to value schemas. A value schema is valid if and only if it can be extended to a valid test case by setting specific values to parameters that have a wildcard.

The example shown in Table I has a simple constraint:

If OS is Linux, then Browser is not IE.

Hence (Windows 7, IE, IPv4, Oracle) is a valid test case, whereas test case (Linux, IE, IPv4, Oracle) is invalid. On the other hand, (Linux, *, IPv4, *) is a valid value schema because it can be extended to, for example, (Linux, Firefox, IPv4, Oracle) which is a valid test case.

An interaction is a combination of values, each for a different parameter. An interaction is said to be a \( t \)-way interaction if it involves exactly \( t \) parameters. A \( t \)-way interaction is identical to a value schema that has exactly \( t \) parameters on which a value is specified. Hence we say an interaction is valid if its corresponding value schema is valid. An interaction is said to be covered by a test case or by a test suite if it occurs in the test case or at least one of the test cases in the test suite.

### B. Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) is a testing technique that requires covering all interactions of interest. A common strategy is \( t \)-way testing, which requires covering all interactions involving \( t \) parameters. In this paper we focus on the problem of covering \( t \)-way interactions.

### C. Applying Random Testing to Interaction Testing

A drawback with CIT is that time taken for the test case generation becomes long when the size of the SUT increases. Applying random testing to covering interactions is one possible way to overcome this drawback. Although there are many variants of random testing, we limit our attention to the simplest one where a test case is generated by sampling an element from the input space \( M_1 \times \ldots \times M_k \) uniformly randomly. This can be performed by simply picking up one value from \( M_i \) for each parameter \( p_i \); thus test cases can be generated very fast.

In [8], Arcuri and Briand provided some fundamental results which formally show the fault detection capability of random testing in interaction testing.

**Theorem 1**: Suppose that there are no constraints. Any \( t \)-way interaction is covered with at least probability 63% by executing a test suite if random testing is used to construct the test suite and the size of the test suite is at least \( |M_1| \times |M_2| \times \ldots \times |M_k| \).

Proof. Directly follows from Theorems 1 and 2 in [8].

However this probabilistic guarantee does not hold if there are constraints over the input space. The authors of [8] presented an extreme example as follows. Suppose that the SUT consists of \( k \) binary parameters \( \{p_1, p_2, \ldots, p_k\} \) with \( M_i = \{0, 1\} \) and the following constraint:

\[
p_1 = 1 \land p_2 = 1 \Rightarrow p_3 = 1 \land \ldots \land p_k = 1
\]

In this case, the 2-way interaction \((1, 1, *, *, *, \ldots, *)\) occurs only in one valid test case, i.e., \((1, 1, \ldots, 1)\). Therefore the probability that this interaction occurs in a randomly generated test case is only \(1/2^k\). This probability decreases by half as \( k \) increases by one. Hence when \( k \) is large, there is almost no chance to test this particular interaction in random testing.

### III. OVERVIEW OF THE PROPOSED APPROACH

This section describes our proposed test case generation approach. The idea of the approach is as follows: Each test case is randomly created. If the test case is valid, then it is added to the pool of test cases. Otherwise, a set of valid test cases are created from that invalid test case and added to the test case pool. The algorithm that creates the set of valid test cases is presented in the next section.

### Table II. Test Suite that Covers All 2-Way Interactions

<table>
<thead>
<tr>
<th>Test Case</th>
<th>OS</th>
<th>Browser</th>
<th>Protocol</th>
<th>DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows 7</td>
<td>IE</td>
<td>IPv4</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>Windows 8</td>
<td>Firefox</td>
<td>IPv6</td>
<td>MySQL</td>
</tr>
<tr>
<td>3</td>
<td>Linux</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Sybase</td>
</tr>
<tr>
<td>4</td>
<td>Windows 8</td>
<td>IE</td>
<td>IPv6</td>
<td>Sybase</td>
</tr>
<tr>
<td>5</td>
<td>Windows 7</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Oracle</td>
</tr>
<tr>
<td>6</td>
<td>Windows 8</td>
<td>IE</td>
<td>IPv4</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>Linux</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Oracle</td>
</tr>
<tr>
<td>8</td>
<td>Windows 7</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>Linux</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

Table II shows a test suite that meets the requirement of 2-way testing for the above example. As shown in this table, only nine test cases suffice to cover all 2-way interactions, whereas there are a total of \( 3 \times 2 \times 2 \times 3 = 36 \) test cases.
IV. ALGORITHM FOR GENERATING NEW TEST CASES

This section describes the design of \textit{createTestcases}(v, t), the function that generates a collection of valid test cases that cover all \textit{t}-way interactions that occur in \textit{v}. A naive algorithm could for example work as follows. For every interaction that occurs in \textit{v}, check whether it is valid or not. If it is valid, then a valid test case is generated by extending the interaction. This algorithm would generate \textit{k} \times (\textit{k} - 1) \times \ldots \times (\textit{k} - \textit{t})! \textit{t}! test cases from a single invalid test case in the worst case. On the other hand, our proposed algorithm outputs only \textit{t} + 1 test case in the best case.

A. Basic Algorithm

The basic algorithm is described as follows. We assume that \textit{t} < \textit{k}, since the case \textit{t} = \textit{k} is not interesting.

Input: invalid test case \textit{v}

Output: collection of test cases \textit{C}

Step 0 Set \textit{v} to value schema \textit{v}'. Initialize \textit{C} to empty.

Step 1 Let \textit{S} be the set of parameters that have values in \textit{v}'. Split \textit{S} into \textit{A}_1, \textit{A}_2, \ldots, \textit{A}_{\textit{t}+1} such that \textit{A}_1 \cap \textit{A}_j = \emptyset (i \neq j), \textit{A}_1 \cup \textit{A}_2 \cup \ldots \cup \textit{A}_{\textit{t}+1} = \textit{S}, and \( |\textit{A}_i| > 0 \).

Step 2 Create a value schema from \textit{v}' by replacing the values on the parameters in \textit{A}_i with a wildcard. A total of \textit{t} + 1 value schemas are generated.

Step 3 For each of the \textit{t} + 1 value schemas, determine whether it is valid or not and perform the following process.

1) If it is valid, then extend it to a valid test case by replacing wildcards in it with specific values and add the test case to \textit{C}.

2) If it is invalid and has exactly \textit{t} values, then simply terminate this step.

3) If it is invalid and has more than \textit{t} values, recursively perform the process from Step 1 with \textit{v}' set to this value schema.

Once all steps have terminated, the resulting \textit{C} becomes the output.

Here we explain this algorithm using a concrete example. Suppose that the SUT has six binary parameters \textit{p}_1, \ldots, \textit{p}_6 which take either 0 or 1 and a constraint such that

\[ p_1 = 1 \Rightarrow p_2 = \ldots = p_6 = 1 \]

Now suppose that test case \textit{v} = (1, 1, 1, 0, 0, 0) has been randomly generated and that \textit{createTestcases}(v, 2) is executed since this test case is invalid. Figure 1 schematically describes how the execution of \textit{createTestcases}(v, 2) progresses.

In Step 0, \textit{v}' is set to (1, 1, 1, 0, 0, 0). In Step 1 in the first iteration of the algorithm, \textit{S} is set to \{\textit{p}_1, \ldots, \textit{p}_6\}, since all the parameters have an assigned value in \textit{v}'. \textit{S} is partitioned into three mutually disjoint sets: \textit{A}_1, \textit{A}_2, \textit{A}_3. Here suppose that \textit{A}_1 = \{\textit{p}_1\}, \textit{A}_2 = \{\textit{p}_2, \textit{p}_3\}, \textit{A}_3 = \{\textit{p}_4, \textit{p}_5, \textit{p}_6\}.

In Step 2, the three value schemas shown below are derived from \textit{A}_1, \textit{A}_2, \textit{A}_3:

\((*, 1, 1, 0, 0, 0), (1, *, *, 0, 0, 0), \text{and} (1, 1, 1, *, *, *)\). Step 3 is executed for each of the three value schemas. Value schema \((*, 1, 1, 0, 0, 0)\) is valid, that is, it can be extended to a valid test case. The only valid test case obtained by extending this value schema is (0, 1, 1, 0, 0, 0); thus this test case is generated and added to collection \textit{C} (Step 3-1). Value schema \((1, 1, 1, *, *, *)\) is also valid; thus a test case obtained by extending it is generated and added to \textit{C}. Test case \((1, 1, 1, 1, 1, 1)\) is the only valid test case that can be obtained from the value schema.

In contrast, value schema \((1, *, *, 0, 0, 0)\) is invalid, and thus the algorithm is repeated from Step 1 with \textit{v}' set to \((1, *, *, 0, 0, 0)\) (Step 3-3). In the next iteration, \textit{S} becomes \{\textit{p}_1, \textit{p}_4, \textit{p}_5, \textit{p}_6\}. Suppose that \textit{S} were partitioned into \textit{A}_1 = \{\textit{p}_1\}, \textit{A}_2 = \{\textit{p}_4, \textit{p}_5\} and \textit{A}_3 = \{\textit{p}_6\} (Step 1), resulting in value schemas \((*, *, *, 0, 0, 0), (1, *, *, *, *, 0)\) and \((1, *, *, 0, 0, *, *)\) (Step 2). Step 3 is executed for each of the
value schemas. Value schema \((*,*,0,0,0)\) is valid; thus it is extended to a valid test case and added to \(C\). There are a few possible test cases that can be extended from this value schema (e.g., \((0,0,0,0,0)\)). In such a case, one of them is randomly selected. Value schema \((1,*,*,*,0)\) is invalid and it has only two parameters that have an assigned value; thus this value schema is simply discarded (Step 3-2). Note that this value schema can be considered as a two-way interaction and this interaction can never be covered by any test case. Value schema \((1,*,*,0,0)\) is invalid and thus the algorithm is repeated from Step 1 with \(v' = (1,*,0,0,*).\)

One can show that output \(C\) satisfies Properties 1 and 2 as follows. Property 1 trivially holds because any test case in \(C\) is added to \(C\) in Step 3-1 and the test case is always a valid test case that is generated by extending a valid value schema.

The reason why Property 2 holds is as follows. If a value schema \(v'\) is invalid and has more than \(t\) values (Steps 1 and 3-3), then every valid \(t\)-way interaction occurs in at least one of the \(t+1\) value schemas generated from \(v'\) in Steps 1 and 2. If a value schema is valid (Step 3-1), then it can be extended to a valid test case and the test case contains all \(t\)-way interactions that occur in the value schema. If a value schema is invalid and has exactly \(t\) values (Step 3-2), then the only \(t\)-way interaction that occurs in the value schema is trivially invalid.

This algorithm leaves some room for detailed design. In the rest of this section, we show two methods that can be incorporated into the basic algorithm.

B. Avoiding Generating Redundant Test Cases

This method is to use an additional termination condition for recursion of the algorithm of computeTestCases\((v,t)\). This condition is that in Step 3 of the algorithm the value schema processed in the current step is identical to or “part of” a value schema that has already occurred in the current execution of the algorithm. We say that a value schema \(v_1\) is part of another value schema \(v_2\) if every parameter that is assigned a value in \(v_1\) has the same value in \(v_2\).

For example, in our running example illustrated in Figure 1, value schema \((*,*,0,0,0)\) is part of \((*,1,1,0,0,0)\) which has already occurred. Thus \((*,*,0,0,0,0)\) is not processed and Step 3 handling \((*,*,0,0,0,0)\) simply terminates. The correctness of the algorithm is maintained because all valid interactions in \(v_1\) are covered by the test cases derived from \(v_2\).

C. Strategic Parameter Set Splitting

In the basic algorithm of createTestCases\((v,t)\), Step 1 divides \(S\) into \(t+1\) mutually disjoint sets, where \(S\) is the set of parameters that have already been assigned a specific value in value schema \(v\). However, Step 1 of the basic algorithm does not specify how to partition the parameter set \(S\) into \(t+1\) disjoint sets. A simple design choice is to uniformly split the set into \(t+1\) sets of almost the same size.

The heuristic proposed here splits \(S\) in a different manner, aiming to reduce the number of test cases generated. The heuristic is based on two ideas. The first idea is that if many of the \(t+1\) value schemas derived from the invalid test case are valid, then the number of newly generated test cases becomes small, because the number of recursions becomes small. The second idea is that when a value schema is valid, if it has many parameters that have an assigned value, then the resulting test cases become few in number, because many interactions are covered by the single valid test case derived by extending the value schema.

Based on the ideas, we devise a greedy-type heuristic for parameter set splitting. Here we describe the heuristic using the running example. First, the first group of parameters, \(A_1\) is determined. Staring from \((*,*,*,*,*)\), parameters that have a value are gradually increased, while keeping the value schema valid as follows.

\[
\begin{align*}
(*,*,*,*,0) & \rightarrow \text{valid} \\
(*,*,*,0,0) & \rightarrow \text{valid} \\
(*,*,0,0,0) & \rightarrow \text{valid} \\
(*,1,0,0,0) & \rightarrow \text{valid} \\
(1,1,0,0,0) & \rightarrow \text{valid} \\
(1,1,1,0,0,0) & \rightarrow \text{invalid}
\end{align*}
\]

As a result, \((*,1,1,0,0,0)\) is derived and \(A_1\) is set to \(\{p_1\}\). The second parameter group \(A_2\) is determined the same way, starting from \((1,*,*,*,*,*)\). However, this time this is not possible, because:

\[
(1,*,*,*,*,0) \rightarrow \text{invalid}
\]

In such a case, the remaining parameter set \(\{p_2,p_3,p_4,p_5,p_6\}\) is divided into subsets of the same size. Since \(A_2\) and \(A_3\) are yet to be determined, the parameter set is split into two halves \(\{p_2,p_3\}\) and \(\{p_4,p_5,p_6\}\) and \(A_2\) is set to \(\{p_2,p_3\}\). Finally \(A_3\) is determined as \(\{p_4,p_5,p_6\}\), as \(A_3\) must be \(S\backslash(A_1 \cup A_2)\).
V. EXPERIMENT

We performed an experiment in order to evaluate the performance of the approach and to clarify the effects of the proposed methods. In this experiment, we consider the problem of testing two-way interactions ($t = 2$). The experiment was performed on a PC with Windows 8 Professional 64 bits OS, Intel Core i7-3537U 2GHz CPU and 8Gbyte memory. The algorithm was implemented using the Java language. Our implementations use binary decision diagrams (BDDs) [10] to perform tests whether a value schema is valid or not. This could be done by using other constraint solvers. For BDD manipulations, we use JBDD libraries.\(^1\)

We applied the proposed approach to five SUT models which represent well-known real-life applications. These applications and models are explained in [11] in details. The parameters of these models are explained in the Parameters column of Table III in the form $l_1^{m_1}l_2^{m_2} \ldots$ which means that the model has $m_i$ parameters that have a domain of size $l_i$. The Constraints column shows the number of constraints and the number of parameters involved in each of the constraints. If the SUT model has $z$ constraints that involves $w$ parameters, then the fact is represented as $w^z$. For example, the SPIN-S model consists of 18 parameters, including 13 binary parameters and five parameters whose domain size is four, and 13 constraints that involve two parameters.

For each SUT model, we set $n$, the number of iterations of the main algorithm, to $|M_1| \times |M_2|$ where $M_1$ and $M_2$ are the two largest parameter domains (see Section II-A). By Theorem 2, this number guarantees that the test suite generated by the algorithm can cover any two-way interaction with at least probability $63\%$.

The previous section proposed an additional termination condition for recursion and described two different design options for splitting parameter set $S$. As a result, we have four different settings in using the proposed approach.

- H1: Uniformly random parameter set splitting.
- H2: Uniformly random parameter set splitting and the additional termination condition for recursion.
- H3: Strategic parameter set splitting.
- H4: Strategic parameter set splitting and the additional termination condition for recursion.

Our interest is how many test cases are added to provide the probabilistic guarantee. We measure the size of the resulting test suite to evaluate the algorithm with the four different settings. We also measured the execution time of the algorithm and the ratio of the covered interactions to all valid interactions. We executed 100 runs for each given SUT model. The minimum, the maximum and average of test suite size over the 100 runs are shown in Table III.

The theoretical result (Theorem 2) guarantees that every interaction can be covered with $63\%$; but the experimental results show that the coverage actually achieved was much higher than this lower bound and reached nearly $100\%$. This can be accounted for the fact that the size of the resulting test suite constructed by the algorithm was larger than $n$.

One can see from Table III that significant reduction in test suite size was achieved with strategic parameter set splitting (the cases of H3 and H4). The reduction achieved by using the additional recursion termination condition was not as drastic as in the case of strategic parameter set splitting but still substantial (the cases of H2 and H4).

The effects of these methods vary for different problems. The problems of Apache, GCC, and Bugzilla have less severe constraints than the other problems and exhibited greater reduction in test suite size. Roughly speaking, using both methods (the case of H4) test suite size was reduced by $90\%$ or more compared to the case of H1. The two methods also achieved significant reduction in test suite size for the remaining two problems, SPIN-S and SPIN-V; but the degree of reduction was less than the other problems. This result is counter-intuitive because severer constraints require adding more test cases and thus the two methods should have more tangible effects, though this was not the case in the experiment. We plan to further investigate how constraints affect the performance of the proposed approach in future.

\(^1\)http://javaddlib.sourceforge.net/jdd/

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parameters</th>
<th>Constraints</th>
<th>$n$</th>
<th>Setting</th>
<th>Coverage (%)</th>
<th>Test suite size</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>$2^{12} \times 3^{10} \times 4^{5} \times 6^{1}$</td>
<td>$2^{12} \times 3^{10} \times 4^{5}$</td>
<td>30</td>
<td>H1</td>
<td>$100.0$</td>
<td>$1.532$</td>
<td>$2.323$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCC</td>
<td>$2^{12} \times 3^{10}$</td>
<td>$2^{12} \times 3^{10}$</td>
<td>9</td>
<td>H1</td>
<td>$100.0$</td>
<td>$5.665$</td>
<td>$7.317$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bugzilla</td>
<td>$2^{12} \times 3^{10}$</td>
<td>$2^{12} \times 3^{10}$</td>
<td>16</td>
<td>H1</td>
<td>$100.0$</td>
<td>$368$</td>
<td>$742$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPIN-S</td>
<td>$2^{12} \times 3^{10}$</td>
<td>$2^{12} \times 3^{10}$</td>
<td>16</td>
<td>H1</td>
<td>$100.0$</td>
<td>$296$</td>
<td>$461$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPIN-V</td>
<td>$2^{12} \times 3^{10}$</td>
<td>$2^{12} \times 3^{10}$</td>
<td>16</td>
<td>H1</td>
<td>$100.0$</td>
<td>$5.785$</td>
<td>$6.415$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parameters</th>
<th>Constraints</th>
<th>$n$</th>
<th>Setting</th>
<th>Coverage (%)</th>
<th>Test suite size</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As the reduction in test suite size directly led to the reduction of execution time, execution time was also significantly decreased using strategic parameter set splitting, although this method required an increased execution time per test case.

For comparison purposes, we measured the execution time of a CIT test suite generation tool. We chose CASA [12]. CASA uses a simulated annealing-based algorithm and has proven to be very effective in minimizing CIT test suites [12]. Note that CIT tools guarantee to cover every valid interaction. We executed 10 runs for each given SUT model and measured the size of the resulting test suites and the execution time. Table IV shows the average size of test suites and the average execution time over the 10 runs.

Comparing Tables III and IV, one can see that the execution time of the proposed random testing algorithm is much smaller by some order of magnitude than that of CASA. Table IV also shows the size of the test suites obtained by CASA. The size is much smaller than the proposed random testing approach; but the ratio between the two approaches significantly varies for different problems.

These results seem to show that there is no clear winner. The proposed random testing approach is superior in execution time for test case generation and can achieve high interaction coverage. CIT tools guarantee perfect interaction coverage and is very effective in minimizing test suite size. Hence the best approach depends on testing scenarios. A thorough discussion on this topic can be found in [8].

VI. Threats to Validity

In the experiment we examined only five models of SUTs. Although these models were derived from real-world applications, it is unclear whether or not they can be regarded as representatives of typical real-world software systems. Hence the observations made from the experimental results may not necessarily hold in general.

Because of the nature of random testing, we executed 100 runs of the test suite generation process for each different setting in the experiment. This number of runs may not be sufficient to draw solid statistical conclusions. We plan to apply some statistical test on the results obtained.

VII. Conclusion

This paper addressed the problem of applying random testing to the constrained interaction testing problem. The proposed approach compensates the loss in interaction coverage by generating new test cases if a randomly generated test case violates the given constraints. We devised an algorithm for generating these test cases with two methods that can be incorporated into the algorithm. The experimental results showed that using both methods can significantly reduce the number of test cases that are added. Possible directions of future work include conducting more comprehensive experimental studies and improving the proposed algorithms.

ACKNOWLEDGMENTS

We thank Mr. Shingo Tanak for his preliminary work. Some ideas in this paper are due to him. We also thank Mr. Daiki Shigeoka for providing the performance results of CASA.

REFERENCES