Towards an Automated Unified Framework to Run Applications for Combinatorial Interaction Testing

Bestoun S. Ahmed*
Dept of Mathematics and Computer Science, Karlstad Univ, Sweden and Dept of Computer Science, Czech Technical Univ in Prague, Czech Republic
bestoun@kau.se

Amador Pahim
Red Hat Czech s.r.o., Brno, Czech Republic
apahim@redhat.com

Cleber R. Rosa Junior
Red Hat, Inc., Westford, USA
crosa@redhat.com

D. Richard Kuhn
Natl Inst of Standards and Technology, Gaithersburg, MD, USA
kuhn@nist.gov

Miroslav Bures
Dept of Computer Science, Faculty of Electrical Eng, Czech Technical Univ, Prague, Czech Republic
buresm3@fel.cvut.cz

ABSTRACT

Combinatorial interaction testing (CIT) is a well-known technique, but industrial experience is needed to determine its effectiveness in different application domains. We present a case study introducing a unified framework for generating, executing and verifying CIT test suites, based on the open-source Avocado test framework. In addition, we present a new industrial case study to demonstrate the effectiveness of the framework. This evaluation showed that the new framework can generate, execute, and verify effective combinatorial interaction test suites for detecting configuration failures (invalid configurations) in a virtualization system.

KEYWORDS

Automated testing framework, Software testing, Combinatorial testing applications, Software quality assurance, Test automation

ACM Reference Format:
Bestoun S. Ahmed*, Amador Pahim, Cleber R. Rosa Junior, D. Richard Kuhn, and Miroslav Bures. 2019. Towards an Automated Unified Framework to Run Applications for Combinatorial Interaction Testing. In *Proceedings of ACM conference (EASE2019).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Combinatorial interaction testing (CIT) (sometimes called t-way testing) is based on the *covering array* (CA) [11], a matrix that includes all t-way combinations of input parameter values, for a specified level of t (usually $t \leq 6$ for software testing) of the system-under-test (SUT). Research activities have focused mainly in two directions (1) generating combinatorial interaction test suites that provide t-way coverage, and (2) applying a combinatorial interaction approach to test industrial systems.

CIT has shown impressive results in many testing studies and large-scale industrial projects [6, 17]. The usefulness of CIT could be for example a systematic reduction of a test suite or detecting new faults in a SUT due to interactions of input parameters, or identifying invalid configurations of the SUT. In fact, finding new applications for CIT is an active research area.

To apply CIT on any SUT, the first step is to model the input parameters or configurations of the system. Typically, this process involves identifying the input parameter and configuration values that are needed in testing. For continuous-valued parameters, equivalence class partitioning will generally be needed to reduce the domain size to a tractable level. The CA generation tool uses the input model to produce test suites that cover all t-way combinations of values of the CA variables. A limited form of this method is "pairwise" or "all-pairs" testing, which includes all 2-way combinations of parameter values. Stronger forms of combinatorial testing use 3-way, 4-way, or higher strength CAs to detect complex faults that depend on multiple factors interacting.

Once the test suites are generated, they must be executed and their output verified, steps which can be combined and automated. For example, test generation and execution can be combined using a scripting language (e.g.,[22]). However, applying these steps may vary from one application to another, so implementing CIT in practice is usually application-specific.

In this paper, we introduce a generic unified framework approach to apply CIT in practice. This framework is an output of a successful industry-academia-government collaboration effort. We have integrated the CIT capabilities into the Avocado[1] framework. Avocado is an open source testing framework maintained by Red Hat Inc. and Avocado Community Contributors. The framework consists of a combination of tools and libraries to ease automated testing by providing a set of programs for test execution and different APIs for writing test cases. The test can be written in a user's programming language or using a Python API. The new plugin will add

EASE2019, 2019, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://avocado-framework.github.io/

the capability of CIT to the framework. Originally, Avocado was designed as a flexible framework that can be used to run any set of test cases for an application as far as the plugin of that application is available. For example, the user can use it for unit testing, virtualization testing, security testing, or mobile application testing. Almost everything is a plugin in Avocado and the development of a new plugin is made straightforward to extend the functionality of the runner for testing any new application.

Using these capabilities of Avocado, we have designed and implemented a new and unique plugin to extend the Avocado framework by interacting with the other plugins, which allows generating much more efficient and effective test data. In doing so, the user will need only to follow the input modeling style of Avocado to enter the values of the SUT and then let Avocado generate, run, and verify the test cases.

2 BACKGROUND AND LITERATURE

CIT relies on a covering array (CA), to derive the combinatorial interaction test suites. A CA is based on t-way coverage criteria (where t represents the desired interaction strength, which is the number of factors interacting). $CA(N; t, k, v)$, also expressed as $CA(N; t, v^k)$, is a combinatorial structure constructed as an array of N rows and k columns on v values such that every $N \times t$ sub-array contains all ordered subsets from the v values of size t at least once. A mixed-level covering array $(MCA)(N; t, k, (v_1, v_2, \dots\ v_k))$ or $MCA(N; t, k, v^k)$ may be adopted when the number of component values varies [19], while the Constrained Covering Array (CCA) may be adopted when there are constraints among the values of input parameters [3].

Research activities in CIT include (1) the generation of combinatorial interaction test suites, and (2) the application of CIT [3]. In fact, the problem of test suite generation has received more attention from the research community. Generation algorithms vary from random generation [12] to mathematical constructions for limited small size and interaction strength such as generation from Orthogonal Arrays (OA) [11], to the deterministic generation methods like In parameter Order (IPO) algorithm [16] and its variants IPOG, IPOG-D [15], and IPOG-F [9]. Metaheuristic algorithms have also been used widely in the last decade to optimize the generation process. Here, many algorithms are used, such as Genetic Algorithms (GA) [5], Ant Colony Algorithms (ACA) [23], Particle Swarm Optimization (PSO) [2], and Tabu Search [18]. A comprehensive survey of these generation algorithms and many others can be found in [3]. In fact, with the availability of all these algorithms and tools, algorithm research for CIT can be said to have reached a mature state, making CIT practical for real-world applications.

Many studies have investigated different applications of combinatorial methods to software testing and program verification. Many applications emerged in this direction, including investigation of the relationship between code coverage and t-way coverage [13], fault detection and characterization [25], graphical user interface testing (GUI) [26], and model-based testing and mutation testing [8]. In fact, there are many applications of CIT in the software testing discipline. Combinatorial interaction testing also finds its way to other fields rather than software testing. For example, it has been used in satellite communication testing, hardware testing [7], advanced material testing [20], dynamic voltage scaling (DVS) optimization [24], and gene expression regulation [21]. Many other application domains can be found in the literature, and researchers are actively discovering new uses for CIT.

As mentioned previously, to apply CIT in practice, there is a need to organize variables and values of the SUT into some interpretable input model for input to the CA test generation tool. After generating the test cases, tests should be executed on the SUT, and then the test output should be verified for the pass and fail criteria. These steps are applicable for almost all applications. However, details of each step may vary from one application to another. For example, the input model, and the execution of the test cases may vary depending on the SUT. There are always some manual activities in this process. In the literature, there are some efforts [12] to develop adaptive solutions to generate and execute the test cases, but still, they are application-specific solutions. A more practical solution is to develop a flexible automated framework that can generate, run, and evaluate combinatorial test suites on any SUT. One approach to integrating these steps is CITLab [10], designed to improve the interoperability among combinatorial testing tools, by providing a framework for defining domain-specific languages.

In this paper, we introduce an enhancement of the Avocado testing framework to include CIT capability and apply this new capability to automate testing of configuration specifications for the open source hypervisor Qemu. Avocado provides a set of tools and supporting libraries for test automation on the Linux platform. The framework can take the input of the SUT as a model and generate test cases according to it, then run and verify the test cases, as described in the following sections.

3 THE AVOCADO FRAMEWORK

Avocado is an open source testing framework maintained by Red Hat Inc. and Avocado Community Contributors, that is designed to give common ground to both quality assurance (QA) teams and Developers. The framework is a set of tools and libraries to help with automated testing. Here, the native tests are written in Python; however, any executable can serve as a test. Avocado consists of three main components, the test runner, libraries, and plugins. The test runner enables the user to run the test cases. Avocado provides the flexibility of writing test cases in Python or any other programming language. In both cases, there are facilities to record the activities during the test, such as information collection of the system and automatic logging. Libraries are used to create and write test cases in an expressive way. The plugins are extensions to Avocado for adding more functionality and features to the framework. The ability to add more plugins to the framework easily assures maximum flexibility for future developers. Figure 1 shows the basic building blocks of the framework from the user perspective. The framework supported by the GDB[2] front-end for the user interface. The test runner relies on plugins and many of them can be used during test execution. The output of the testing process can be saved and used in JSON, Xunit, HTML, or TAP formats. Hence, the output file can be used in different ways by the developers or testers. For example, it can be integrated with Jenkins[3] to trigger the testing process.

[2]https://sourceware.org/gdb/
[3]https://jenkins.io/

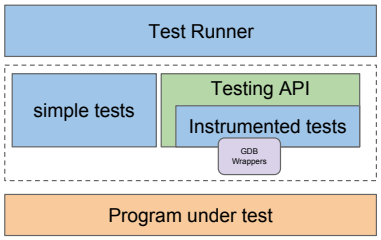


Figure 1: Avocado high level test writer view

To automate the testing process of any SUT, the tester needs to create test cases and then run them using the plugins. The plugin for that SUT must be used to automate the process. The test runner will automatically run the SUT with the necessary environment (i.e., necessary resources and programs) to execute the test cases. For example, suppose we want to test a mobile application using some set of test cases and record the output in any of the used formats. Here, the test runner will run the test cases using a plugin to start the mobile emulator. When the test runner initiates the emulator, the test cases will be executed automatically by the Avocado, and the output will be saved directly to the user output format.

If the user of Avocado wants to test an application and the plugin is not available, then the user first needs to create a plugin to start the environment and setting up all the relevant services, stubs, and programs to start the application. In fact, the creation of a new plugin for this purpose is simplified by our development team. The user needs to follow a few simple steps to create this plugin. These steps can be found in the Avocado documentation[4].

These flexibilities of the Avocado framework make it attractive to implement an automated unified framework for combinatorial interaction testing applications. Here, we extend Avocado to handle the combinatorial interaction testing automatically to perform the model creation, test case generation, test case execution, and evaluation of the test oracle. By adding this feature, we can get the benefits of CIT to reduce and detect faults in the SUT, with the benefits of Avocado to automate, run and verify the testing process. In the next section, we illustrate this CIT extension to Avocado.

4 THE CIT EXTENSION TO AVOCADO

The CIT extension to Avocado adds significant capabilities to the framework and makes it possible to apply CIT in a fully automated testing process (see Figure 2). Here, the user first needs to model the SUT input parameters to the CIT file format, i.e., determine the parameters to be included in tests, and values for each parameter. The output of CIT test generation will be a set of tests that covers all t-way combinations of the parameter values, for a selected level of t. Thus, if t=2, the tests will instantiate all sets of variables taken two at a time with all pairs of possible values for these variables. For example, in Figure 2, there are four input parameters, each with a different number of values. The input parameters and values are also represented in Avocado in a tree representation model. The test resolver will resolve this input model to an interpreted format understandable by the CIT plugin. The CIT plugin will generate

[4]http://avocado-framework.readthedocs.io/en/59.0/

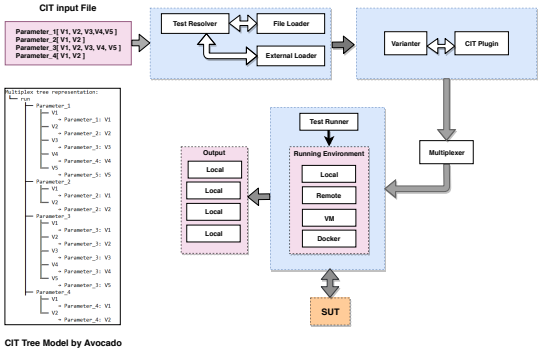


Figure 2: Basic structure of Avocado with CIT extension

the test cases and send them to the multiplexer that creates the scripts to be run on the SUT. In some applications, the test suite is a configuration setting of the SUT and there are still some input files or codes that must be input with the configuration. We call this type of input a *variant*. Hence, if there is a variant in the SUT, Avocado will consider it with the test suite through the multiplexer. The multiplexer will then send the test cases to the test runner. The test runner executes the test cases with respect to the SUT. For each test case, the Avocado will record the output and show a verification message on the screen.

The test runner can run the test cases in different ways. As shown in Figure 2, it can run the test cases on the conventional local machine, remote machine, virtual machine, or even on a Docker container. After running the test cases, the runner saves the recorded results of the output in a customizable format. The output format can be in XML, JSON, Tap, or HTML format, as selected by the user. The final runner output and test time execution are also shown on On the output screen in Pass and Fail forms. Here, the PASS and FAIL depend on the output of the test case. For a simple test, it is PASS when the exit code is "0". For an instrumented test, it is considered PASS for example when there is no exception in setUp(), testMethod(), or tearDown(). A test is also considered PASS when the runner gets the final test status. This PASS and FAIL is also customizable, and the user can state the PASS and FAIL based on the output summary of the test.

Combinatorial interaction test suites are generated with the previously developed algorithm PSTG [1, 4]. PSTG generates combinatorial interaction test suites using Particle Swarm Optimization (PSO). The algorithm has been assessed extensively and proved its efficiency, with different benchmarks and experiments in the literature. The user also can contribute to the test generation algorithm as it is an open source framework.

5 AN INDUSTRIAL CASE STUDY

To show the effectiveness of the automated framework, it is applied in an industrial case study of validating configurations in a virtualization environment. In addition to the evaluation aim of this case study, it also demonstrates a somewhat different application of CIT, in which CAs are used in configuration checking as compared with conventional code testing.

5.1 Object of the Study

The object of the study is the Qemu virtualization Project[5], specifically, the Qemu Block Layer configuration checking tests. Qemu is a machine emulator and virtualizer. An essential component of this virtualizer is its Block Layer. Every emulated or virtualized machine will need at least one virtual disk to fulfill its purpose. Qemu supports a variety of disk image formats, and they can be created in many different ways. To ensure that a configuration can function correctly on the virtualizer, a set of shell scripts and commands are run on Qemu. In this case study, there are 192 such validation scripts. A configuration is determined to be *valid* if all 192 validation scripts pass, otherwise the configuration is *invalid*.

We present a real case study from the industry. In this work, CIT tests are not used to detect coding errors, but to identify invalid configuration settings. The case study is a sample of bigger projects that were inspired and adopted by the Red Hat Quality Assurance (QA) team. We used the Avocado framework as a test bed.

To cover with the required testing of those many formats and options for creating Qemu disk images, the Qemu Project holds a 'qemu-iotests' directory, with the validation test cases[6]. The tests consist of BASH scripts that will be executed by the Avocado Test Runner. Avocado will receive the combinations from the CIT Plugin, parse it into a Tree object and iterate that Tree object to create the Test Suite, an object containing each variation of tests per parameters combination. Test scripts will then be executed with a given 'Variant'(i.e., a combination of parameters and variants) in place, to be consumed in the form of environment variables. The test script will create the virtual disk image following those options and then manipulate it to test if the generated image file complies with the requirements, as specified within the test script. Based on the test assessment of the created image manipulation, the test script will return the corresponding exit code to Avocado. Using that exit code, Avocado will mark the final test status as PASS (exit code 0) or FAIL (exit code non-0).

5.2 The System Under Test

The virtualized system under test used for this experiment consists of five input parameters. Each parameter represents a specific type of image that can be used as an option in the system configuration that has to be run as a Qemu project. The Avocado tree representation[7] of the input model can be generated by Avocado.

As can be seen in the tree representation model, the system has five inputs, each of them having different configuration values. For example, the image format could be one of five values, i.e., raw, qcow, qcow2, luks, and vmdk. The full meaning of each parameter and its corresponding values is shown in Table 1. A possible system configuration is a combination of these variables and settings, so the input model structure for combinatorial test generation would be designated $5^2 2^3$, i.e., two variables with five values and three with two values. Note that this expression also gives the number of possible combinations, in this case 200. Combining these variables together will form a configuration; however, this does not mean that the configuration is a valid configuration, thus the need for validation scripts as explained above.

5.3 Evaluation and Analysis

To evaluate the effectiveness of Avocado for CIT, we present here the results of our case study. As previously mentioned, we are not evaluating the test generation algorithm efficiency of the Avocado framework, as it has been reviewed extensively in the literature. We used the PSTG algorithm to generate the test cases. More evaluation results and comparison with other algorithms and tools can be found in the literature (e.g., [1, 4]). Note also that the Avocado framework is composed of many plugins and tools, and evaluating them is beyond the scope of this paper.

Here, we aim to validate the effectiveness and performance of the Avocado CIT extension regarding four critical issues:

(1) The coverage validity of the generated test cases
(2) The performance of the testing process
(3) The efficiency of the multiplexer integration with test generation when generating different t-way test suites.
(4) Level of fault detection for different t-way test suites.

As the generation algorithm uses PSO concepts, it generates nondeterministic results due to the random initialization of the algorithm. To this end, we ran each test several times and addressed the results to assure a fair statistical experiment. For coverage validity, we checked it in each run; however, due to limited space in the paper, we only present one graph for each t-way test suite. We ran the test cases for the performance and effectiveness 30 times and then produced a box-plot for them. All the experiments were performed on a Linux Fedora personal computer with 2.9 GHz Intel Core i5 CPU and 8GB 2133 MHz of memory.

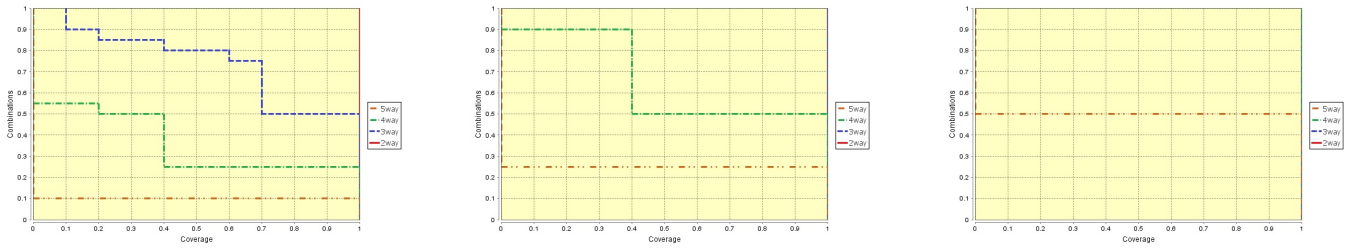
The basic concept of the CIT is that all the combination tuples must be covered by the generated test suite at least once. To assure that our Avocado framework covers all these tuples, we used the Combinatorial Coverage Measurement Command Line Tool (CCMCL)[8] for evaluation. The tool was developed by the National Institute of Standards and Technology (NIST) to measure and validate the coverage of a t-way test suites. Figure 3 shows the coverage measure for 2-way, 3-way, and 4-way test suites.

The CCMCL tool is a coverage strength meter to determine the combinatorial coverage of any test suite and also identify any missing combinatorial tuples. Note that combinatorial coverage as evaluated by CCMCL is different from conventional structural coverage measures such as statement or branch coverage. Combinatorial coverage is a measurement of the proportion of t-way combinations included in a test suite (a static measure), rather than a dynamic execution-related code measure such as statement coverage. It is clear from Figure 3a that the generated 2-way test suite achieves the 100% coverage of the tuples. Here, the red indicator line is on the right side of the graph which indicates the 100% achieved coverage level. Figure 3a also shows the 3-way, 4-way, and 5-way (i.e., the exhaustive test suite with full strength) for the same 2-way test suite for comparison. Here, we can see that the 2-way test suite can assure the full coverage of 2-way combinatorial tuples but it only covers 50% of 3-way combinatorial tuples (blue line), 25% of

[5] https://www.qemu.org/
[6] The used scripts can be found in https://github.com/qemu/qemu/tree/master/tests/qemu-iotests
[7] Example of the tree model can be found here https://bit.ly/2UmWrEW
[8] https://github.com/usnistgov/combinatorial-testing-tools

Table 1: Parameter and corresponding value meanings of the SUT

Parameter/Value	Meaning
img_format	The format in which the Qemu image will be created.
raw	no specific format, raw data.
qcow	The versatile Qemu Copy On Write format, first version.
qcow2	The versatile Qemu Copy On Write format, second version.
luks	The Linux Unified Key Setup encrypted image format.
vmdk	VMWare image format.
img_protocol	The protocol used to access the image.
file	Direct access through the filesystem.
nbd	Network Block Device protocol, enabling a remote server to be used as block device.
cache_mode	The method used to cache data with the image file.
none	The host page cache is skipped and writes happen directly from the userspace buffers and the image.
writeback	The default option, where direct cache and no-flush are off. The host page cache is used and writes are reported to the guest as complete when they are committed to the page cache.
writethrough	Doesn't batch metadata updates. Writes are reported as complete after data is committed to the image.
directsync	Writes reported as complete when the data is committed to the image, skipping the host page cache.
unsafe	Same as "writeback" with additional ignore for the flush commands (no-flush enabled).
misalign	Misalign allocations for direct writes.
true	Enabled
false	Disabled
qemu_img	The qemu-img binary to create images with.
/usr/bin/qemu-img	The Fedora 27 version (2.10).
/git/qemu/qemu-img	The latest upstream version (master).



(a) 2-way full coverage **(b) 3-way full coverage** **(c) 4-way full coverage**

Figure 3: The coverage measure for 2-way, 3-way, and 4-way test suites

4-way combinatorial tuples (green line), and 10% of 5-way combinatorial tuples (brown line). The full coverage of 3-way and 4-way combinatorial tuples can be achieved with 3-way and 4-way test suites respectively in Figures 3b and 3c. Figure 3c also illustrates a basic property of CAs. A CA for t-way combinations will also provide 100% coverage of s-way combinations, for any $s < t$, i.e., the designated strength and all lower strength tuples. For example, in Figure 3c the 4-way test suite covers the 4-way, 3-way and 2-way combinations.

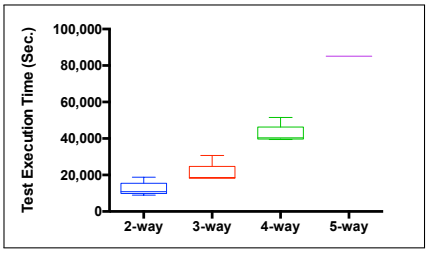
To evaluate the performance of the testing process, we compared the execution time of each t-way test suite. This time represents the total time the Avocado framework takes to generate, execute, and validate the test cases. Figure 4a shows the box plot analysis to predict the performance and compare the execution time.

The box plot in Figure 4a reveals a number of salient characteristics of the Avocado and also CIT in general. We can see that the execution time increases with the interaction strength as the size of the test suite is increasing towards the exhaustive (5-way) test suite. Even though the lower quartile of the 3-way test suite is near to the upper quartile of the 2-way test suite, the time of execution has a notable difference among the test suites. Also, we can see that the deviation and differences among individual test execution time within the same t-way test suite are not more significant as the top-to-bottom whisker width is small.

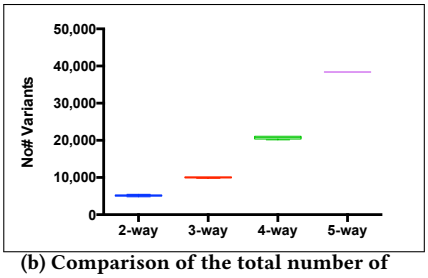
To test the efficiency of the multiplexer integration with test generation, we have compared the number of variants generated for the SUT. Here, the number of the variant is computed by the number of total variants multiplied by the number of test cases generated for a specific t-way value. For this case study, we have 192 local variants that each test case must run to validate the configurations. For example, there are 200 test cases for the 5-way test suite. Hence, the total number of variants generated is $200 \times 192 = 38400$ variants to be run for the 5-way testing. Figure 4b shows the box plot analysis for the total number of variants generated for each test suite.

From Figure 4b, we can see a clear difference among the generated variants. There is not even a matching between the lower and upper quartiles of two different variant sets, which in turn shows the test generator and also the multiplexer efficiency in generating variants.

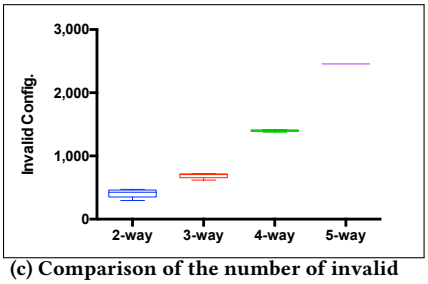
Finally, to evaluate the effectiveness of finding invalid configurations in the case study, we compared the output of Avocado for different test suites. Figure 4c shows a box plot analysis graph to compare different t-way test suites. To give a better understanding of the number and proportion of these values, Table 2 shows median values of the variants' number, invalid configurations out of these variants, and the ratio between them. Note that, as mentioned in Section 5.1, the counts and ratios in Table 2 do not refer to errors in the code, but to configurations that are not valid in the virtualization environment.



(a) Comparison of the test execution time by each t-way test suite



(b) Comparison of the total number of variants generated by avocado during the testing with respect to a t-way test suite



(c) Comparison of the number of invalid configurations detected by Avocado with respect to a t-way test suite

Figure 4: The coverage measure for 2-way, 3-way, and 4-way test suites

Table 2: Median values of the generated variants and invalid configurations found with the ration between them

Test Suite	#Variants	#Invalid Cofig.	Ratio %
2-way	5184	425	8.19
3-way	9984	705	7.06
4-way	20736	1405	6.77
5-way	38400	2455	6.39

Table 3: Invalid Configuration Identification Test Case Results

	2-way	3-way	4-way	5-way
Test cases:	470	722	1415	2455
removing (t-1)-way	-	376	960	2113
removing (t-2)-way	-	-	572	768
removing (t-3)-way	-	-	-	382

An important observation can be found in Figure 4c and also Table 2 . Here, we note that the number of detected invalid configurations increases with the growth of the combinatorial interaction strength. We found that the total number of invalid configurations in the SUT is 2455 when the full strength (i.e., 5-way) is considered. As can be predicted in the upper quartile, in the best case, the 2-way test suite can detect 470 invalid configurations, which is almost 19% of the total faults. In the same way, the 3-way test suite can detect nearly 29% of total invalid configurations in the best case, while 4-way can detect almost 58% of total invalid configurations in the best case. We can observe that for this application, the t-way test suite can reduce the size and the execution time of the test suites dramatically. However, the full strength combinatorial test suite is necessary to detect all the invalid configurations. Another important observation is that using combinations of input configuration setting values is an effective way to identify invalid configurations. Here, Avocado presents an excellent choice for automating configuration validation and testing.

We can see from the results that more invalid configurations are found as the test size and combination strength increase. However, it is not immediately clear from these results how many of those configurations detected at 5-way are the result of containing a particular 2-way, 3-way, or 4-way combination that detects an invalid configuration. As it is clear from Figure 3, a configuration with some particular combination strength will also exist in other higher strength combinations of configurations. For example, if we have five binary variables, A, B, C, D, E, and a 2-way combination A=0, B=1 and that results in an invalid configuration, then any 3-way or higher strength combination that includes these values for A and B will also be detected as an invalid configuration. In particular, the 3-way combinations ABC = 010, ABC = 011, ABD = 010, ABD = 011, ABE = 010, and ABE = 011 would all include the invalid combination. Therefore a test such as ABCDE = 01011 would include three 3-way combinations, three 4-way combinations, and one 5-way combination that detect an invalid configuration, but these counts are redundant because only the 2-way combination AB=01 is needed for detection. It is easy to see how these multiple counting situations would increase with a large number of variables. To address this situation and provide a deeper analysis of the test cases, we have reviewed those invalid or failed configurations to determine if this situation is occurring. We have developed a simple open source tool[9] that counts the occurrence of the combinations in any t-way test suite based on the strength. Table 3 shows the result of this analysis.

Table 3 shows the result of analyzing the test suites. We run the test suites with Avocado and monitor the output of each test case. As mentioned previously, these test suites are used with the multiplexer to produce the variants' set. We identified the invalid configurations in the variant set; then we analyzed each configuration case for covered combinations. As shown in the table, we found and isolated the number of invalid variants' combinations in the configurations. Hence, the numbers used in Table 3 are the variants' combinations after multiplication by the configuration test suite.

We have addressed the size of the invalid configurations for each t-way test suite. In addition, we have addressed this size of an invalid configuration after removing the repeated lower t-way combinations. Here, we used $t-1$, 2, and 3 test suites. For example, for the 5-way test suite, we also generate the 4-way, 3-way, and 2-way combinations and compared them with the used equivalent test suites to identify the repeated test cases based on the tuples.

As we note from the results in Table 3, some configurations are failed (determined to be invalid) in the lower strength of combinations, and they are also repeated in the higher strength test

[9]https://github.com/bestoun/CombinatorialCounter

suite. For example, using the analysis tool on the invalid 5-way test cases, we found that 342 repeated 4-way invalid configurations out of those 2455, which results in 2113 5-way configurations after removing them. Also, there are 1345 repeated 3-way invalid configurations out of those 2113 remaining configurations, which results in 768 5-way configurations after removing them. In the same way, there are 368 repeated 2-way invalid configurations out of those 768 remaining configurations, which results in 382 5-way configurations after removing them.

We can conclude from Table 3 that for this application, lower strength combinations are responsible for part of those invalid configurations; however, higher interaction strength (greater value of t) combinations are needed to detect all the possible configuration failures. In fact, this shows that unlike studies of fault detection in the literature, e.g. [14], running only pairwise (2-way) test cases is not enough for this application to trigger most of the failures. The reason for this difference is that other applications of combinatorial testing have generally been for detecting errors in code. In this case, however, testing addressed detection of configurations that could not be supported in the virtual machine environment, rather than detecting code flaws. This is a different use of combinatorial methods, but t-way testing was shown to be highly effective. Like any other configurable system, for larger configurations of this application in the industry, running full exhaustive testing in most cases is impossible. Hence, running lower combination strength test suites with Avocado is an option to assure quality and avoid triggering configuration failure.

6 CONCLUDING REMARKS

We have demonstrated a method of automating the combinatorial interaction testing process, using the open source Avocado testing framework with CIT capabilities implemented in a plugin. Within Avocado CIT, the tester needs only to establish the environment of the application to be tested. The Avocado framework was used for validating virtual machine configurations for Qemu, demonstrating that Avocado can be a cost-effective tool for automating this essential step in the virtualizer setup.

Avocado is a flexible and customizable framework in which other capabilities, features, algorithms, and tools can be added easily through a plugin. We plan to add constraint handling capabilities to the framework through a constraint solver. Avocado is a freely available open source project freely available[10].

ACKNOWLEDGEMENTS

This research is funded by Red Hat Czech s.r.o. as a collaboration project with Software Testing Intelligent Lab (STILL) in CVUT and part of Avocado testing framework project. Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.

REFERENCES

[1] Bestoun S. Ahmed and Kamal Z. Zamli. 2010. PSTG: A T-Way Strategy Adopting Particle Swarm Optimization. In *Proceedings of the 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation (AMS '10)*. IEEE Computer Society, Washington, DC, USA, 1–5. https://doi.org/10.1109/AMS.2010.14

[2] Bestoun S. Ahmed and Kamal Z. Zamli. 2011. A Variable Strength Interaction Test Suites Generation Strategy Using Particle Swarm Optimization. *J. Syst. Softw.* 84, 12 (Dec. 2011), 2171–2185. https://doi.org/10.1016/j.jss.2011.06.004

[3] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730. https://doi.org/10.1109/ACCESS.2017.2771562

[4] Bestoun S. Ahmed, Kamal Z. Zamli, and Chee Peng Lim. 2012. Application of Particle Swarm Optimization to Uniform and Variable Strength Covering Array Construction. *Applied Soft Computing* 12, 4 (April 2012), 1330–1347. https://doi.org/10.1016/j.asoc.2011.11.029

[5] P. Bansal, N. Mittal, A. Sabharwal, and S. Koul. 2014. Integrating greedy based approach with genetic algorithm to generate mixed covering arrays for pair-wise testing. In *2014 Seventh International Conference on Contemporary Computing (IC3)*. 629–634. https://doi.org/10.1109/IC3.2014.6897246

[6] Redge Bartholomew. 2013. An industry proof-of-concept demonstration of automated combinatorial test. In *Proceedings of the 8th International Workshop on Automation of Software Test*. IEEE Press, 118–124.

[7] S. Yu. Borodai and I. S. Grunskii. 1992. Recursive generation of locally complete tests. *Cybernetics and Systems Analysis* 28, 4 (01 Jul 1992), 504–508. https://doi.org/10.1007/BF01124983

[8] Miroslav Bures and Bestoun S. Ahmed. 2017. On the Effectiveness of Combinatorial Interaction Testing: A Case Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 69–76. https://doi.org/10.1109/QRS-C.2017.20

[9] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, and D.R. Kuhn. 2008. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology* 113, 5 (2008), 287–297. cited By 91.

[10] Angelo Gargantini and Paolo Vavassori. 2012. CitLab: a laboratory for combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 559–568.

[11] Alan Hartman and Leonid Raskin. 2004. Problems and algorithms for covering arrays. *Discrete Mathematics* 284, 1 (2004), 149 – 156. https://doi.org/10.1016/j.disc.2003.11.029 Special Issue in Honour of Curt Lindner on His 65th Birthday.

[12] R. Huang, X. Xie, T. Y. Chen, and Y. Lu. 2012. Adaptive Random Test Case Generation for Combinatorial Testing. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. 52–61. https://doi.org/10.1109/COMPSAC.2012.15

[13] Jerry Huller. 2000. Reducing time to market with combinatorial design method testing. In *In Proceedings of the 2000 International Council on Systems Engineering (INCOSE) Conference*. 16–20.

[14] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. 2004. Software fault interactions and implications for software testing. *IEEE transactions on software engineering* 30, 6 (2004), 418–421.

[15] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG-IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing. *Softw. Test. Verif. Reliab.* 18, 3 (Sept. 2008), 125–148. https://doi.org/10.1002/stvr.v18:3

[16] Yu Lei and Kuo-Chung Tai. 1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98)*. IEEE Computer Society, Washington, DC, USA, 254–261.

[17] Xuelin Li, Ruizhi Gao, W Eric Wong, Chunhui Yang, and Dong Li. 2016. Applying combinatorial testing in industrial settings. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, 53–60.

[18] Kari J. Nurmela. 2004. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics* 138, 1 (2004), 143 – 152. https://doi.org/10.1016/S0166-218X(03)00291-9 Optimal Discrete Structures and Algorithms.

[19] Arturo Rodriguez-Cristerna, Jose Torres-Jimenez, W. GÃşmez, and W.C.A. Pereira. 2015. Construction of Mixed Covering Arrays Using a Combination of Simulated Annealing and Variable Neighborhood Search. *Electronic Notes in Discrete Mathematics* 47 (2015), 109 – 116. https://doi.org/10.1016/j.endm.2014.11.015 The 3rd International Conference on Variable Neighborhood Search (VNS'14).

[20] Ulrich S. Schubert. 2004. Experimental Design for Combinatorial and High Throughput Materials Development. Edited by James N. Cawse. *Angewandte Chemie International Edition* 43, 32 (2004), 4123–4123. https://doi.org/10.1002/anie.200385086

[21] Dennis E. Shasha, Andrei Y. Kouranov, Laurence V. Lejay, Michael F. Chou, and Gloria M. Coruzzi. 2001. Using Combinatorial Design to Study Regulation by Multiple Input Signals. A Tool for Parsimony in the Post-Genomics Era. *Plant Physiology* 127, 4 (2001), 1590–1594. https://doi.org/10.1104/pp.010683

[22] Anwar Sherif. 2016. Combinatorial Testing: Implementations in Solutions Testing. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 59–64. https://doi.org/10.1109/ICSTW.2016.39

[23] Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. 2004. Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing. In *Proceedings*

[10] https://github.com/avocado-framework/avocado

of the 28th Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '04). IEEE Computer Society, Washington, DC, USA, 72–77.

[24] Diary R. Sulaiman and Bestoun S. Ahmed. 2013. Using the combinatorial optimization approach for DVS in high performance processors. In *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*. 105–109. https://doi.org/10.1109/TAEECE.2013.6557204

[25] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. 2004. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *SIGSOFT Softw. Eng. Notes* 29, 4 (July 2004), 45–54. https://doi.org/10.1145/1013886.1007519

[26] Xun Yuan, Myra B. Cohen, and Atif M. Memon. 2011. GUI Interaction Testing: Incorporating Event Context. *IEEE Transactions on Software Engineering* 37, 4 (July 2011), 559–574. https://doi.org/10.1109/TSE.2010.50