Testing Django Configurations Using Combinatorial Interaction Testing

Justyna Petke

CREST Centre, University College London, UK
j.petke@ucl.ac.uk

Abstract. Combinatorial Interaction Testing (CIT) is important because it tests the interactions between the many parameters that make up the configuration space of software systems. We apply this testing paradigm to a Python-based framework for rapid development of web-based applications called Django. In particular, we automatically create a CIT model for Django website configurations and run a state-of-the-art tool for CIT test suite generation to obtain sets of test configurations. Our automatic CIT-based approach is able to efficiently detect invalid configurations.

1 Introduction

Software testing is a challenging and highly important task. It is widely believed that half the costs of software projects are spent on testing. Search-based software engineering (SBSE) techniques have also been successfully applied to the problem [6]. Moreover, over half of the literature on the whole field of SBSE is concerned with testing [7]. In order to apply an automated test method to the problem at hand, one might require information about the inner workings of the system to be tested. Such white-box testing techniques [3, 8, 11, 16, 17] generate test suites that systematically exercise the program, for instance, to cover all program branches. These, however, require knowledge of inner workings of the system under test, setup could take significant amount of time, while resultant test suite might not be better than the one generated manually [4].

The vast majority of software systems can be configured by setting some top-level parameters. For checking system behaviour under the different settings, knowledge of the inner workings of such systems is not necessarily required. This is a situation where black-box testing methods, such as Combinatorial Interaction Testing (CIT), come in handy. The aim of an automated technique for software configurations is to generate a test suite that exercises various system settings. Testing all possible combinations of parameters is infeasible in practice. There exist, for instance, a model for the Linux kernel that contains over 6000 features that can be set[1]. Even if all these took Boolean values, 2^{6000} configurations would have had to be generated in order to test them all. In order to

[1] Linux kernel feature model is available at: `https://code.google.com/p/linux-variability-analysis-tools/source/browse/2.6.28.6-icse11.dimacs?repo=formulas`

avoid this combinatorial explosion problem, techniques such as CIT have been introduced.

Combinatorial Interaction Testing (CIT) aims to test a subset of configurations, yet preserve high fault detection rate when compared with a set of all possible parameter combinations. It is a light-weight black-box testing technique that allows for efficient and effective automated test configuration generation [12]. Several studies have shown that CIT test suites are able to discover all the known interaction faults of the system under test [1, 18, 10, 15]. Hence, we have chosen this method to test Django, a very popular Python-based framework for rapid development of web-based applications.

Django[2] was designed to help developers create database-driven websites as quickly as possible. Among popular sites using it are: Instagram, Mozilla and The Washington Times. Django is written in Python and comes with its own set of unit tests. A global settings file is also provided and contains parameters that can be configured in any Django-based web application. The set of values for some of the Django settings is potentially infinite, since they admit strings. We have thus concentrated on Boolean parameters only. We used CIT to test the various combinations of Django's Boolean settings available and discovered several invalid configurations.

2 Background

Combinatorial interaction testing (CIT) has been used successfully as a system level test method [1, 2, 10, 9, 14, 15, 18]. CIT combines all t-combinations of parameter inputs or configuration options in a systematic way so that we know we have tested a measured subset of the input or configuration space. A CIT test suite is usually represented as a covering array (CA): $CA(N; t, v_1^{k_1} v_2^{k_2} ... v_m^{k_m})$, where N is the size of the array, t is its strength, sum of $k_1, ..., k_m$ is the number of parameters and each v_i stands for the number of values for each of the k_i parameters.

Suppose we want to generate a pairwise interaction test suite for an instance with 3 parameters, where the first parameter can take 4 values, the second one can only take 3 values and the third parameter can take 5 values. Then the problem can be formulated as: $CA(N; 2, 4^1 3^1 5^1)$. Furthermore, in order to test all combinations one would need $4 * 3 * 5 = 60$ test cases. If, however, we cover all interactions between any two parameters, then we only need 20 test cases. Such a test suite is called a 2-way or pairwise test suite.

There are several approaches for covering array generation. The two most popular ones use either simulated-annealing (SA) or a greedy algorithm. The SA-based approach is believed to produce smaller test suites, while the greedy one is regarded to be faster [5]. A state-of-the-art CIT tool that implements an SA-based algorithm is Covering Arrays by Simulated Annealing (CASA)[3]. It is relatively mature in the CIT area, so we chose it for our experiments.

[2] Django is available at: `https://www.djangoproject.com/`

[3] CASA is available at: `http://cse.unl.edu/~citportal/`

3 Setup

In order to generate a CIT test suite for Django, we need to first find the parameters that we can configure. The source distribution of Django comes with a top-level settings file called GLOBAL_SETTINGS.PY, an extract of which is shown in Figure 1. There are 137 parameters defined. In order to ease tester effort we only consider Boolean ones. Otherwise, we would have to inspect each non-Boolean parameter to identify which values are allowed (theoretically each parameter of type *string* can take infinitely many values). This can be done as a future step.

```
# Whether a user's session cookie expires when the Web browser is closed.
SESSION_EXPIRE_AT_BROWSER_CLOSE = False
# The module to store session data
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
```

Fig. 1. An extract from Django's GLOBAL_SETTINGS.PY file.

We construct a CIT model for GLOBAL_SETTINGS.PY automatically by counting the number of Boolean parameters. There are 23 such parameters. Hence the CIT model is: $CA(t; 2^{23})$. We produce test configurations for $t = 2$ and $t = 3$ using the CASA tool, that is, we produce a pairwise test suites and a test suite that covers all value combinations between any three parameters. Higher-strength CIT is feasible [13], however, pairwise testing is the most popular both in the literature as well as in the industry. 3-way testing is not as frequently used and there are only few studies focusing on higher-strength CIT[4].

Next, we automatically construct multiple copies of the SETTING.PY file based on the CIT test configurations that will be substituted in turn with the default settings file that is created whenever a Django project is started. We first ran the unit test suite to check if all tests pass. We do not know whether these exercise all possible Django configurations. Next, we evaluate each CIT test case by first running the Django development server. Afterwards we re-run the tests and invoked two websites: 'Welcome to Django' page and the 'polls' website, which is the default website used in Django tutorials[5]. We chose these two webpages since they are the most basic ones and hence any fault-triggering test case for these will likely produce a fault for more complex webpages. We use MacBook Air with 1.7 GHz Intel Core i7 processor and 8GB of RAM.

We emphasise that each step is automated (except for voting which could be automated as well): from parameter extraction through model generation and test case generation to actual testing of the system. Therefore, the whole process could potentially be applied to any other configurable software system, without knowledge of the inner workings of such a system.

[4] This is partially due to the fact that the higher interaction strength t is required, the larger the number of test cases that need to be generated. Pairwise testing is believed to be good enough, especially since several empirical studies have shown that 6-way testing can discover all the known faults [10].

[5] Django tutorials are available at: `https://www.djangoproject.com/`

4 Results

The source distribution of Django comes with its own test suite. It is composed of unit tests that perform 9242 checks on the MacBook Air laptop used. Thus, we first ran the existing test suite to check if all of them pass. It is possible that some of these test various Django configurations, but we have not investigated this. The original test suite did not reveal any faults.

The CASA tool, which uses simulated-annealing, produced 8 test configurations that cover all pairwise interactions between the 23 parameters extracted from GLOBAL_SETTINGS.PY settings file in less than a second. 2^{23} tests would have been needed to test all possible combinations of parameter settings. 3-way test suite was created within 42 seconds. The CIT test suites generated are the smallest possible for the chosen criteria[6]. We have created the default project, as presented in part 1 of Django's 'Writing your first Django app' tutorial, and substituted the SETTINGS.PY file with the automatically generated variants in turn. In 4 out of 8 configurations, from the pairwise test suite, an error occurred. An explanation, however, was provided by Django as shown in Figure 2.

```
Django version 1.9.dev20150502163522, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

Fig. 2. An error caught by Django.

Before moving forward, we have added a constraint to the CIT model that the parameter 'ALLOWED_HOSTS' must be set to '['127.0.0.1', 'localhost']' if the 'DEBUG' parameter is False. We re-generated and re-run the tests and they all passed. Next, we invoked the 'Welcome Page' after starting the server for each of the different test setting configurations generated using CASA. We repeated the experiment with the 'polls' website described in the Django tutorial. If the voting page opened properly, we also posted a vote. Since our results were similar for both 'Welcome Page' and 'polls' websites, we report only those for the 'polls' website. We mention differences where applicable.

Our experiments revealed that most of the combinations of configuration values were invalid. For the pairwise test suite 6 out of the 8 test configurations did not allow for the 'polls' webpage to be invoked. In particular, in 4 cases HTTP error 301 was thrown and the website did not load within pre-specified amount of time (10 seconds). In 2 cases security HTTP error 400 was thrown with the following message: 'You're accessing the development server over HTTPS, but it only supports HTTP.'

The 3-way test suite for the 'polls' website consisted of 23 test cases. 4 runs produced correct results; 11 produced HTTP error 301 and timed-out; and security HTTP error 400 was observed 6 times. Additionally, two configurations triggered an error that only occurred when invoking the 'polls' webpage, but not the 'Welcome to Django' one. It was triggered by pressing the voting button, causing redirection to 'Forbidden page' and throwing HTTP error 403.

[6] The smallest known test suite sizes for various CIT models are available at: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

We tried to find the minimal configurations causing each type of error. We first extracted configurations that were set to the same values in all the settings files causing the same type of error. We then compared these against the default settings to find changes. In the case of the HTTP 301 error, there was only one line that all the relevant settings files had in common, namely 'PREPEND_WWW = True', which was set to 'False' in the default settings file. We checked that indeed it was the cause of the HTTP 301 error. Moreover, we found a post online from a user of Django, who had encountered the same problem. It took him 5 days to figure out the root cause of this error manually. Using the same approach, we found that the security HTTP error 400 was caused by 'SECURE_SSL_REDIRECT = True' and 'Forbidden page' HTTP 403 errors were caused by 'CSRF_COOKIE_SECURE = True'. This is not to say that the three configurations are always invalid. In Django documentation for 'PREPEND_WWW' parameter, for instance, it states 'This is only used if CommonMiddleware is installed'.

This small experiment has shown that even though Django server seems to be well-tested, one needs to be careful when modifying the default settings. CIT can provide a quick way of finding invalid configurations for a particular Django project.

5 Conclusions

Many real-world software systems are highly configurable. Combinatorial interaction testing (CIT) techniques have been developed specifically for such systems. CIT test suites cover all interactions between any set of t parameters. Different parameter values can be set, for instance, via modifying a top-level settings file. An example of such a system is a very popular framework for rapid web development called Django. By applying CIT techniques to test basic websites written in Django automatically we discovered that under many test configurations invoking a basic website produces errors. Moreover, each step of our approach is (or could be) automated and does not involve any knowledge of the inner workings of the Django system. Therefore, it can be applied to any other configurable software system.

References

1. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
2. Cohen, M.B., Colbourn, C.J., Gibbons, P.B., Mugridge, W.B.: Constructing test suites for interaction testing. In: Proceedings of the International Conference on Software Engineering. pp. 38–48 (May 2003)
3. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Software Eng. 39(2), 276–291 (2013), http://doi.ieeecomputersociety.org/10.1109/TSE.2012.14

4. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013. pp. 291–301 (2013), http://doi.acm.org/10.1145/2483760.2483774
5. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering 16(1), 61–102 (2011)
6. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing (keynote). In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation (2015)
7. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45(1), 11 (2012), http://doi.acm.org/10.1145/2379776.2379787
8. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Software Eng. 36(2), 226–247 (2010), http://doi.ieeecomputersociety.org/10.1109/TSE.2009.71
9. Kuhn, D.R., Okun, V.: Pseudo-exhaustive testing for software. In: SEW. pp. 153–158. IEEE Computer Society (2006)
10. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Software Eng. 30(6), 418–421 (2004)
11. McMinn, P.: Search-based software test data generation: a survey. Softw. Test., Verif. Reliab. 14(2), 105–156 (2004), http://dx.doi.org/10.1002/stvr.294
12. Petke, J., Cohen, M., Harman, M., Yoo, S.: Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. Software Engineering, IEEE Transactions on PP(99), 1–1 (2015)
13. Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13. pp. 26–36. ACM, Saint Petersburg, Russian Federation (August 2013)
14. Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the International Symposium On Software Testing and Analysis. pp. 75–86 (2008)
15. Qu, X., Cohen, M.B., Woolf, K.M.: Combinatorial interaction regression testing: A study of test case generation and prioritization. In: ICSM. pp. 255–264. IEEE (2007)
16. Tillmann, N., de Halleux, J.: White-box testing of behavioral web service contracts with Pex. In: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), TAV-WEB 2008, Seattle, Washington, USA, July 21, 2008. pp. 47–48 (2008), http://doi.acm.org/10.1145/1390832.1390840
17. Tonella, P.: Evolutionary testing of classes. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. pp. 119–128 (2004), http://doi.acm.org/10.1145/1007512.1007528
18. Yilmaz, C., Cohen, M.B., Porter, A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering 31(1), 20–34 (Jan 2006)