Configuration Aware Prioritization Techniques in Regression Testing

Xiao Qu
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska 68588-0115
xqu@cse.unl.edu

Abstract

Configurable software lets users customize applications in many ways, and is becoming increasingly prevalent. Regression testing is an important but expensive way to build confidence that software changes introduce no new faults as software evolves, resulting in many attempts to improve its performance given limited resources. Whereas problems such as test selection and prioritization at the test case level have been extensively researched in the regression testing literature, they have rarely been considered for configurations, though there is evidence that we should not ignore the effects of configurations on regression testing. This research intends to provide a framework for configuration aware prioritization techniques, evaluated through empirical studies.

1. Introduction

User configurable software — software that can be customized through a set of options by the user — is becoming increasingly prevalent. Internet Explorer (IE), for example, has many options that can be easily controlled by the user through the GUI, such as the "pop up blocker" — the user can decide if pop ups are allowed when he or she browses web pages, and the web pages will show in different formats according to different choices. From a testing perspective, even if each configuration appears largely similar, the underlying execution of code for the same set of test cases may differ widely across configurations [5, 11]. This implies that we need to not only consider test cases but also configurations for testing configurable systems.

Regression testing, with the goal of detecting newly introduced faults in modified software, is an expensive part of the software maintenance process and is often resource limited [8, 9]. Configurability will further increase the cost of regression testing, magnifying it by the number of configurations.

Due to the large configuration space in most systems, it is infeasible to test all possible configurations so sampling of the configurations should occur instead. Even if sampling techniques are applied, it is still possible to run out of time, therefore more important configurations, which have higher probabilities of detecting more faults, should be tested earlier, i.e. they should have higher priorities.

Efforts have been made to address similar problems at the test case level and a primary focus has been on techniques for reducing test suite size (selection) (e.g., [3, 10, 14]) or for ordering test cases (prioritization) (e.g., [6, 15]). None of this research, however, has explicitly considered issues involving configurations. Testing techniques considering configurations have been created [13, 17], but to date, little work has addressed the problems of regression testing configurable systems as they evolve.

Giving the foregoing discussion, our research has an overall goal of providing a framework for configuration aware prioritization techniques. This framework aims to address the prioritization problem in regression testing configurable systems — it will provide both a set of sampling techniques for configurations, and a set of prioritization techniques for both test cases and configurations.

All of the above will be evaluated through empirical studies, on several software systems, with respect to appropriate metrics. Because the related metrics [6] do not necessarily fit in our environment, we will develop new ones. Moreover, existing studies have shown that configurations have an impact on testing [5, 11]; we intend to understand the types of faults and test cases that are sensitive to configurations.

2. Background and Related Work

This section provides background on sampling and regression testing that will be utilized in our work.

2.1. Combinatorial Interaction Testing

Sampled configurations for testing are either manually developed or generated by different techniques. Among the sampling techniques is a systematic method called Combinatorial Interaction Testing (CIT) [4], which was originally applied to sample programs' input space.

Recent work also suggests that CIT may provide an effective way to sample configurations for testing [7, 17]. A CIT entity samples the full configuration space so that it includes all t-way combinations of choices between options, where t is called the strength of testing.

2.2. Regression Testing Techniques

There has been a large body of work on traditional regression testing for both test case selection (e.g., [3, 10, 14]) and test case prioritization (e.g., [6, 15]). But most of this work has focused on the test case as the object, rather than the configuration, and none of it has been applied to a CIT environment. In [1], Bryce and Colbourn present an algorithm to prioritize CIT test suites. They do not, however, experiment on real software subjects, nor do they address the key element of weighting the various elements that drive prioritization.

In [2] the authors prioritize existing test suites for GUI-based programs by t-way interaction coverage, but the test suites themselves are not generated by CIT techniques. There will be little benefit if this prioritization heuristic is applied to CIT samples as in our environment.

In [17] the authors sample the configurations with CIT techniques, but they test only the selected configurations, and conduct fault characterization on the results. This work considers only a single version of a system and mainly addresses the fault localization problem.

In [13] the authors propose a technique to detect latent faults based on configuration changes. Configuration changes are mapped to the underlying code and tests are then selected or created that cover the impacted areas. Though this technique is deployed in an incremental way, it is not traditional regression testing since it only addresses configuration relevant faults in a single version of a system.

In [6] a metric that is commonly used for prioritization is the *Average Percentage of Faults Detected* or APFD. One drawback of this metric is that it assumes that the number of test cases and detected faults do not change. In our environment, it is possible that we do not find all the faults and we do not run the same number of tests / configurations. In [16] Walcott et al. address this problem by assigning a penalty to the missed faults which allows their APFD to become negative, but we prefer a method which just calculates the area covered by detected faults and complete test cases under the resource constraint, whose value can never be negative.

3. Goals and Approaches

This section describes the goal of this research and specifies several activities we will perform to achieve this.

3.1. Overall Goal

The overall goal of this research is to provide a framework for configuration aware prioritization techniques. This goal is visualized in Figure 1 and described in Section 3.2.

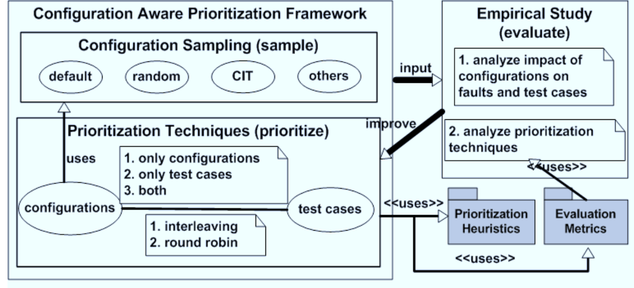

Figure 1. Overview of research

3.2. Activities

To achieve this goal, we are performing the following activities; each is mapped to a box in Figure 1 labeled with the same name:

1. sample full configuration space (*sample*);
2. develop prioritization techniques and combine them at different levels (*prioritize*);
3. evaluate the framework and analyze the impacts of configurations (*evaluate*).

Many methods can be used to *sample* configurations: use a single default configuration, randomly select configurations, generate configurations by CIT or other techniques. The method of sampling should neither be too arbitrary nor too random. We will review the existing literature on sampling techniques, and choose appropriate ones for study.

To accomplish the *prioritize* activity, we will develop different techniques to combine prioritization at the configuration and the test case level. The combination will be considered in two dimensions: the method of combination ("interleaving" or "round robin") and the degree of prioritization (to prioritize configurations only, to prioritize test cases only or to prioritize both).

One supporting task is to develop heuristics (left shaded box in Figure 1), including the information that drives prioritization and the ways we use it. Many existing prioritization techniques are based on executions of previous versions [6, 15]. These often rely on code coverage information. We will investigate information other than this, which does not require previous version executions. As for the usage of selected information, besides traditional techniques that are applicable to configurations / test cases regardless of how they are generated, we will also consider techniques that are specifically designed for certain sampling methods. In this work, we will address both traditional and sampling based techniques.

The first two activities will be *evaluated* in an empirical approach, shown as the "**input**" relation in Figure 1. This activity aims at analyzing the impact of configurations on faults and test cases, and analyzing prioritization techniques. In order to evaluate the effectiveness of the prioritization techniques, we will examine existing metrics or de-

velop new ones if necessary (right shaded box in Figure 1). The results of these analyses may be helpful in providing feedback to "**improve**"(Figure 1) prioritization techniques.

3.3. Scope and Research Plan

The research just defined will be scoped in a number of ways to make its completion feasible in a reasonable amount of time.

We will limit the number of techniques that will be investigated in the activities proposed above. Adjustments to these activities may be made as we acquire a better understanding of adapting, creating, and empirically evaluating techniques. For the *sample* activity, we will focus on the CIT technique, and for sampling based prioritization techniques, we will mainly work on CIT based prioritization.

Though our primary work is to create a general framework for prioritizing configurable systems, specific domains may have specific properties. User configurable systems, component based systems, and software product lines all belong to configurable environments but in different domains; due to this we can model them in different ways and prioritize with different techniques. In our current research, we will limit the application to user configurable systems.

4. Preliminary Work

At this point, we have investigated configuration sampling techniques and developed some prioritization techniques. In the next sections, we discuss our efforts for each activity, as well as ongoing work if applicable.

4.1. Sampling Configurations

In order to find effective sampling techniques that will work for prioritizing configurations, we first studied different sampling techniques at the test case level [12]: random selection and CIT [4]. Aside from a comparison between these two sampling methods, in order to further examine the effectiveness of CIT in sampling, we also generated all possible program inputs, and regarded the fault detection of the full test suite as a benchmark. In this study, we conducted experiments on two software subjects, **flex** and **make**, each contains several consecutive versions with seeded faults. Our results showed that on average, the CIT test suite can detect a comparable number of faults as the full test suite, and detect more faults than the same size random test suite. Another observation is that the higher the testing strength, the more faults are detected. In most versions, 2-way (pairwise) CIT test suites can detect as many faults as the full test suite, while 2-way random test suites miss more faults more regularly. Currently, we are working on an extended version of this work, in which we generalize CIT based prioritization beyond pairwise and experiment on an additional software subject.

We then extended the above work to a configurable software system, **vim** [11]: we developed a specification for the configurations, and sampled configurations by pairwise CIT. We have also randomly generated the same number of configurations as the pairwise CIT. Our results showed that both of these samples can detect all seeded faults when all configurations are tested cumulatively, but the CIT sample is distributed more towards higher fault finding, and this advantage is verified to be significant by a Wilcoxon two sample test. The CIT sample was also compared to the default configuration provided with the software, and we found that half of the CIT configurations detect more faults.

4.2. Prioritization Heuristics

In our initial study, we first applied a *traditional* block coverage based prioritization technique [6], and then tried techniques that leverage CIT properties — we extended Bryce and Colbourn's *interaction benefit* based prioritization technique [1] by developing different weighting schemes and methods of extracting information from either block coverage or the specification. The specification based prioritization does not require an execution of the previous version. Moreover, we applied both a re-generation technique as well as a reordering of an existing sample. In our study, all prioritized test suites exhibit early fault detection, compared to unordered test suites.

In our second study, we applied *interaction benefit* based prioritization at the configuration level, driven by block coverage, fault detection and the specification. We concluded that prioritized configurations can exhibit earlier fault detection compared to unordered configurations, given the same set of test cases.

In summary, at this point, we have developed 8 prioritization heuristics, which can be applied either at the test case or the configuration level: *traditional* block coverage, *traditional* fault detection, *interaction benefit* block coverage, *interaction benefit* fault detection, *interaction benefit* specification, re-generating *interaction benefit* block coverage, re-generating *interaction benefit* fault detection, and re-generating *interaction benefit* specification.

Currently, we are working on combining prioritization at the configuration and the test case level for **vim**.

4.3. Prioritization Metrics

In our first study, existing metrics did not fit into our environment, which includes different numbers of detected faults and different sizes of test suites, so we have re-derived the APFD formula [6] as Normalized APFD or NAPFD. The better the performance of the prioritization (more faults are detected earlier), the higher the value of NAPFD. The same metric is used in our second study.

4.4. Configuration Impact Evaluation

Besides evaluating the sampling and prioritization techniques, we want to quantify and understand in more depth

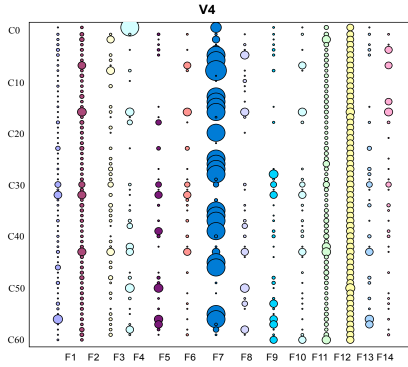

Figure 2. Fault Density

the impact of configurations. In our second study, we verified the impact of configurations on testing that was seen in [5], but in multiple versions of a system. We also quantified the impact with respect to block coverage, fault detection, and more finely, the number of test cases that can detect certain faults (Fault Density), under each configuration. Figure 2 shows the results of the Fault Density for one version of **vim**, where the x and y axis corresponds to faults (F_i) and configurations (C_j), respectively, and the size of each bubble stands for the number of test cases that can detect a particular fault under a certain configuration: faults are dependent or independent of configurations, at either the test case or the test suite level.

Moreover, we examined some specific faults to understand what makes them independent or dependent of configurations, in other words, the impact of configurations on faults and test cases.

Currently, we are setting up and running the same experiments on another user configurable system, **bash**, and we are trying to investigate more about the interactions between configurations, test cases and faults.

5. Expected Contributions and Future Work

Through the activities described in Section 3.2, this research is expected to make the following contributions:

1. A framework for configuration aware prioritization, including sampling and prioritization techniques.
2. A set of types of information to drive prioritization and the methods to implement this.
3. Newly developed metrics that quantify the effectiveness of the prioritization techniques.
4. A better understanding of the types of faults and test cases that are affected by changing configurations.

In the future, we are going to experiment on additional user configurable systems. To improve the prioritization, we are going to extend the heuristics that leverage CIT properties, and investigate profiling data or other information that can drive prioritization. Since the prioritization will be conducted at both the test case and the configuration level, our developed metric may not necessarily fit, so we intend to develop other metrics if needed.

6. Acknowledgments

This work was supported in part by the National Science Foundation through CAREER award CCF-0747009.

References

[1] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Jnl. Info. Softw. Tech.*, 48(10):960–970, 2006.

[2] R. C. Bryce and A. M. Memon. Test suite prioritization by interaction coverage. In *Wkshop. Dom. Spec. Appr. Softw. Test. Auto.*, pages 1–7, 2007.

[3] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, 1997.

[5] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. In *Int'l. Wkshop. Adv. Modl. Test.*, pages 1–9, Nov. 2006.

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.

[7] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, 2004.

[8] H. Leung and L. White. Insights into regression testing. In *Int'l. Conf. Softw. Maint.*, pages 60–69, Oct. 1989.

[9] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.

[10] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Symp. Found. Softw. Eng.*, pages 241–251, Nov. 2004.

[11] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Int'l. Symp. Softw. Test. Ana.*, pages 75–86, 2008.

[12] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Int'l. Conf. Softw. Maint.*, pages 255–264, 2007.

[13] B. Robinson and L. White. Testing of user-configurable software systems using firewalls. In *Int'l. Symp. Softw. Reli. Eng.*, pages 177 – 186, 2008.

[14] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methd.*, 6(2):173–210, Apr. 1997.

[15] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Int'l. Symp. Softw. Test. Ana.*, pages 97–106, July 2002.

[16] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Int'l. Symp. Softw. Test. Ana.*, pages 1–11, July 2006.

[17] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Softw. Eng.*, 31(1):20–34, Jan. 2006.