Feature-Interaction Aware Configuration Prioritization

Son Nguyen
The University of Texas at Dallas
Richardson, Texas, USA
sonnguyen@utdallas.edu

ABSTRACT
Unexpected interactions among features induce most bugs in a configurable software system. Exhaustively analyzing all exponential number of possible configurations is prohibitively costly. Thus, various sampling methods have been proposed to systematically narrow down the exponential number of configurations to be tested. Since testing all selected configurations can require a huge amount of effort, fault-based configuration prioritization, that helps detect bugs earlier, can yield practical benefits in quality assurance. In this paper, we propose CoPo, a novel formulation of feature-interaction bugs via common program entities enabled/disabled by the features. Leveraging from that, we develop an efficient feature-interaction-aware configuration prioritization technique for configurable systems by ranking configurations according to their total number of potential bugs. We conducted several experiments to evaluate CoPo on a public benchmark. We found that CoPo outperforms the state-of-the-art configuration prioritization methods. Interestingly, it is able to detect 17 not-yet-discovered feature-interaction bugs.

CCS CONCEPTS
• Software and its engineering → Software defect analysis;

KEYWORDS
configurable code, feature interaction, configuration prioritization

ACM Reference Format:

1 PROBLEM STATEMENT & BACKGROUND
A configurable system can provide a very large number of configuration options that are used to control different features to tailor systems’ properties to the needs of users and the requirements of customers. Features can interact with others in some non-trivial manners to modify or influence the functionality of one another [18]. Unexpected interactions might induce bugs. In facts, most configuration-related faults are caused by feature interactions [2, 6, 12]. Bugs in a particular variant can be detected by traditional methods. Unfortunately, such tools cannot be directly applied to configurable systems. Furthermore, exhaustively analyzing a system is infeasible due to the exponential number of configurations.

Researchers have proposed several techniques to narrow the configuration space by eliminating invalid configurations that violate the feature model of the system, which defines the feasible configurations via the constraints among the features [4, 5, 7, 9, 10, 15]. However, the number of configurations is still exponential. To address this problem, researchers introduce various configuration selection strategies. The popular strategies include the sampling algorithms [1, 8, 11–14, 17] such as combinatorial interaction testing [8, 11, 13, 14], one-enabled [12], one-disabled [1], to reduce the number of variants to be tested. Still, those algorithms assume the chances of detecting interaction bugs are the same for all those configurations. Thus, interaction faults might be discovered only after the last variants in such samples is tested. Finally, to detect defects in the system in an effective manner, after configuration selection, the selected set of configurations need to be prioritized for testing [3].

2 MOTIVATION AND OBSERVATION
The state-of-the-art method on configuration prioritization, similarity-based prioritization (SP) [3] is still limited in detecting feature interaction bugs. This approach aims to cover as many different features, but does not examine the nature of interaction between them, which is the key aspect causing interaction bugs in a variant.

To illustrate the problem, let us consider a simplified version of two feature-interaction bugs in the Linux kernel in Figure 1:

- A compile-time error occurs (using an undeclared function on line 24) in the configurations in which CONFIG_TWL4030_CORE, CONFIG_OF_DEVICE, and CONFIG_SPARC are enabled.
- A run-time error occurs (dereferencing a NULL variable, line 12) in the configurations where CONFIG_TWL4030_CORE, CONFIG_OF_DEVICE are enabled, CONFIG_SPARC, CONFIG_OF_IRQ are disabled.

SP is based on the idea that dissimilar test sets are likely to detect more defects than similar ones [3]. In Fig. 1, there are 24 legal configurations under test. This set is then narrowed down to 20 by the 4-wise algorithm. By SP, the configuration with the maximum number of features is selected to be the first one under test where TWL4030_CORE, IRQ_DOMAIN, OF_IRQ, and OF_DEVICE are enabled, and SPARC is disabled. The next one is the configuration that has the minimum number of similar features as the previously selected ones, where TWL4030_CORE, IRQ_DOMAIN, OF_IRQ, and OF_DEVICE are disabled, and SPARC is enabled. Although the second variant is most dissimilar compared to the first one, there is no bug revealed by the second. Hence, the resulted schedule is not an efficient testing order because the run-time and compile-time bugs are not detected until the 4th and 6th variants are tested respectively, while the variant with both bugs would only be discovered via the 7th one. Another issue of SP is that the quality of the resulted order is strongly depended on the first selection.
We propose a novel approach for configuration prioritization based on detecting feature-interaction bugs. Unlike the SP which does not examine the nature of potential bug-inducing feature interactions, we analyze the code to detect feature interactions via operations on the program entities that are shared between features. Those operations, when the features are enabled or disabled, potentially create a violation(s) of the program semantics, which helps identify interaction bugs. The suspicious score of a variant is determined via the total number of potential bugs. The variants are ranked according to their suspicious scores.

**Feature.** In CoPo, a feature in configurable code is considered to be implemented via the operations including declare, assign, use, and destruct on a set of program entities including variables and functions. For example, feature L is implemented via operation declare on the function of_platform_populate and variable node.

**Feature Interaction.** Features interact with one another through their operations on the shared program entities depending on the manners that the shared entities operate in the features. In Fig. 1, of_platform_populate is declared in L, and L interacts with Z where the function is used. In a configurable code, enabling or disabling a feature is determined by certain configuration options’ selection. In Fig. 1, L is enabled if CONFIG_SPARC=F, and Z is enabled when CONFIG_TW4830_CORE and CONFIG_OF_DEVICE are true. In fact, an option is used to configure source code of configurable system, such that the option’s value determines the presence/absence of segments of code. Thus, we detect feature interactions via common entities operating in the code segments determined by option selections. For example, because of_platform_populate is respectively declared and used in the code segments determined by CONFIG_SPARC=F and CONFIG_OF_DEVICE=T, there is a potential declare-use interaction among features. In general, with the set of 4 operations, we define 9 kinds of feature interaction such as declare-declare, declare-use and assign-use (7 of total 16 combinations are redundant, e.g., declare-use and use-declare) that are statically detected through program entities shared between features.

**Configuration Prioritization.** To prioritize a set of configurations, our algorithm assigns a suspicious score to all configurations. The score of a configuration is determined via the number of suspicious selections that potentially cause interaction bugs and flaws in different kinds such as *use without assignment* and *memory leak* that the variant might have. In CoPo, suspicious option selections are identified based on the related kind of feature interaction. For example, \{(CONFIG_SPARC=T, CONFIG_OF_DEVICE=T)\} are considered as suspicious because the related *declare-use* interaction, and they potentially cause a *use without declaration* bug.

For example, by running CoPo, there are 4 suspicious selections:

- \{(CONFIG_SPARC=T, CONFIG_OF_DEVICE=T)\}
- \{(CONFIG_OF_IRQ=F, CONFIG_TW4830_CORE=T)\}
- \{(CONFIG_SPARC=T, CONFIG_TW4830_CORE=T)\}
- \{(CONFIG_IRQ_DOMAIN=F, CONFIG_OF_IRQ=F)\}

and a much better test schedule, where CoPo’s top-ranked configuration discovers both above feature interaction bugs.

### 4 EMPIRICAL EVALUATION

We compare CoPo against SP and random prioritization when we run them on the results of 4 advanced sampling methods [12]. We use the Variability Bugs Database [2] with 98 real-world, verified configuration-related bugs in different versions of configurable systems. We select 46 buggy versions that have bugs not related to pointers or external data. We run each sampling method on each version separately to select a subset of valid configurations, which is the input of the prioritization performed by the three methods. The output schedules of the methods are then evaluated via two metrics Average Percentage Faults Detected (APFD) [16] and average rank (AVGR). The higher APFD and the lower AVGR, the better schedule.

<table>
<thead>
<tr>
<th></th>
<th>CoPo</th>
<th>SP</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>APFD</td>
<td>AVGR</td>
<td>APFD</td>
</tr>
<tr>
<td>Pairwise</td>
<td>0.93</td>
<td>1.58</td>
<td>0.75</td>
</tr>
<tr>
<td>Three-wise</td>
<td>0.95</td>
<td>2.45</td>
<td>0.89</td>
</tr>
<tr>
<td>OneEnabled</td>
<td>0.92</td>
<td>10.93</td>
<td>0.70</td>
</tr>
<tr>
<td>OneDisabled</td>
<td>0.86</td>
<td>14.76</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Table 1 shows the average APFD and average rank for the three approaches. As seen, CoPo achieves about 20% higher APFD on average, and is able to rank the buggy configuration in a much higher rank than SP and random approaches.

**Conclusion.** We introduce CoPo, an efficient feature-interaction-aware configuration prioritization technique for configurable systems. The novel idea of CoPo is the code analysis to detect interactions between features and use them to rank configurations. Currently, we formulate feature interaction statically through a complete set of operations on the entities shared between features. More sophisticated interactions relevant to pointers and external data such as files or databases can be detected by using other data structures in the same principle and other analyses.
REFERENCES


