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:
Son Nguyen. 2018. Feature-Interaction Aware Configuration Prioritization. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3236024.3275437

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 combinations*. 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.

```
1  #ifdef CONFIG_TWL4030_CORE
2  #define CONFIG_IRQ_DOMAIN
3  #endif
4  #if !defined(CONFIG_SPARC)
5  int of_platform_populate(int node){
6      return 0;                                   L
7  }
8  #endif
9  #ifdef CONFIG_IRQ_DOMAIN
10 int irq_domain_simple_ops = 1;
11 void irq_domain_add(int *ops){
12     int irq = *ops;
13 }
14 #endif
15 #ifdef CONFIG_TWL4030_CORE
16 int twl_probe(int n){
17     int *ops = NULL, status = n;
18 #ifdef CONFIG_OF_IRQ
19     ops = &irq_domain_simple_ops;
20     status = 0;
21 #endif
22     irq_domain_add(ops);
23 #ifdef CONFIG_OF_DEVICE
24     status = of_platform_populate(n);           Z
25 #endif
26     return status;
27 }
28 #endif
```

Figure 1: A simplified buggy version of Linux kernel

Observation. If we consider the declaration of `of_platform_populate` in `L` (line 5) and its use in `Z` (line 24), the configuration which enables `Z` and disables `L` (`CONFIG_OF_DEVICE=T, CONFIG_SPARC=T`) should be tested earlier to detect the first interaction bug.

3 APPROACH

We propose CoPo, 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_TWL4030_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 thecode 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_TWL4030_CORE=T}`
- `{CONFIG_SPARC=T, CONFIG_TWL4030_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 1: Comparison Results

	CoPo		SP		Random	
	APFD	AVGR	APFD	AVGR	APFD	AVGR
Pairwise	0.93	1.58	0.75	4.03	0.68	5.23
Three-wise	0.95	2.45	0.89	4.74	0.82	8.00
OneEnabled	0.92	10.93	0.70	27.67	0.63	36.37
OneDisabled	0.86	14.76	0.57	38.21	0.60	37.34

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

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[2] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Trans. Softw. Eng. Methodol.*, 26(3):10:1–10:34, January 2018.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 197–206, New York, NY, USA, 2014. ACM.

[4] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM.

[5] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[6] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 113–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 46–55, New York, NY, USA, 2012. ACM.

[9] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[10] Christian Kästner. Virtual separation of concerns: toward preprocessors 2.0. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(1):42–46, 2012.

[11] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 227–235, New York, NY, USA, 2013. ACM.

[12] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA, 2016. ACM.

[13] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 459–468, Washington, DC, USA, 2010. IEEE Computer Society.

[15] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 347–350, Washington, DC, USA, 2008. IEEE Computer Society.

[16] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[17] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. *SIGOPS Oper. Syst. Rev.*, 45(3):10–14, January 2012.

[18] Pamela Zave. Programming methodology. chapter An Experiment in Feature Engineering, pages 353–377. Springer-Verlag New York, Inc., New York, NY, USA, 2003.