Localizing Configurations in Highly-Configurable Systems

Paul Gazzillo
University of Central Florida
paul@pgazz.com

Ugur Koc
University of Maryland, College Park
ukoc@cs.umd.edu

ThanhVu Nguyen
University of Nebraska-Lincoln
tnguyen@cse.unl.edu

Shiyi Wei
University of Texas at Dallas
swei@utdallas.edu

ABSTRACT

The complexity of configurable systems has grown immensely, and it is only getting more complex. Such systems are a challenge for software testing and maintenance, because bugs and other defects can and do appear in any configuration. One common requirement for many development tasks is to identify the configurations that lead to a given defect or some other program behavior. We distill this requirement down to a challenge question: given a program location in a source file, what are valid configurations that include the location? The key obstacle is scalability. When there are thousands of configuration options, enumerating all combinations is exponential and infeasible. We provide a set of target programs of increasing difficulty and variations on the challenge question so that submitters of all experience levels can try out solutions. Our hope is to engage the community and stimulate new and interesting approaches to the problem of analyzing configurations.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software testing and debugging*;

KEYWORDS

Configurations, Variability, Program Analysis, Testing

ACM Reference Format:
Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing Configurations in Highly-Configurable Systems. In *22nd International Systems and Software Product Line Conference - Volume A (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3233027.3236404

1 INTRODUCTION

The complexity of configurable systems has grown immensely, and it is only getting more complex. This complexity creates a great challenge for software testing and maintenance. Critical, widely-used software, such as Linux, BusyBox, Firefox, and Apache, have millions or billions of configurations. While bugs can and do appear in any configuration [1], there are simply too many configurations to test them all separately. With the proliferation of Internet-of-things devices, maintenance and testing highly-configurable systems are even more essential, given the variety of devices using different configurations of the same software.

Many aspects of software maintenance are impeded by configurability, including testing, localizing and repairing bugs, security auditing, and finding code smells and dead code. All must apply to every configuration of the system. One simple distillation of these tasks is to identify interesting configurations: *Given some point of interest in a program, what are the configurations that reach that point of interest?* A point of interest can be a particular line, file, program slice, bug, security violation, or some other subset of program behavior. Ideally, we would like to discover the complete space of configurations that reach the given point.

The ability to answer this question enables a developer to pinpoint relevant configurations when confronted with a defect or when undertaking a maintenance task. For example, bugs in the Linux kernel source code have been introduced inadvertently in one configuration when making repairs to another [1]. Identifying the configurations reaching a bug location will help localize the source of these *variability bugs*.

Localizing configurations remains an open problem for several reasons. The major challenge is *scalability*. For instance, the Linux kernel has over 14,000 configuration options. There are more combinations of these options than the estimated number of atoms in the universe[1] by many orders of magnitude. Further complicating this issue is that not all combinations are valid configurations. For instance, `x86` platform options should be restricted when compiling for an `arm` processor, and reasoning about these constraints is computationally expensive.

Another important difficulty is that many systems often rely on *third-party software* where the source code is unavailable. For Internet-of-things devices, it may not be possible to obtain source code or operating system details to analyze their configurations. Furthermore, in systems with multiple, heterogeneous devices, it is difficult to combine the configurations of each device into a cohesive model.

Even when source code is available, *build systems tend to be ad-hoc*. Each piece of software can implement its own build rules

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3236404

---

[1] https://www.wolframalpha.com/input/?i=estimated+number+of+atoms+in+the+universe

and configuration options may not be explicitly documented. Configurability can be implemented in many ways, sometimes using custom solutions. For C systems, Makefiles and the preprocessor are common options. These tools have difficult semantics, impeding analysis, making a general solution to localizing configurations difficult to realize. For our challenge case, we focus on this build-time configurability.

2 THE CHALLENGE

> *Given a specific program location in the source code, can you apply automatic analysis techniques to find concrete configurations that include the program location in question?*

A program location is a source file name and a line number. A concrete configuration is a valid combination of configuration options that can be used to build the target system. Since many configurations may cover the intended program location, the ideal answer is a compact characterization that captures all configurations covering that location. This characterization may be expressed as a logical formula that we can query to produce valid, concrete configurations reaching the program location.

A naive solution to this challenge is to enumerate all possible combinations of configuration options. Configuring and building the system for each combination enables a search for those that contain the target program location. This approach is not practical, even for small configurable systems, because it is exponential in the number of configuration options. The axTLS web server, for example, contains only 94 configuration options, but the naive approach would require enumerating 2^{94} configurations, an unrealistic proposition.

There are two categories of realistic approaches to solving this challenge. *Static analyses* summarize configuration behavior by analyzing the program source code, including its build system, without executing the program. A purely static approach might not be feasible when the system contains third-party libraries or compiled code. *Dynamic analyses* run the program using a sample of configurations, obtain execution traces, and use the traces to build a model describing the configuration behavior. A purely dynamic approach might not provide a precise model describing the complete configuration space. A solution may need to combine both static and dynamic approaches, particularly for very large systems.

Solution requirements. A valid solution will, for any line number or file name, be able to produce at least one concrete configuration that, when used to configure and build the target system, includes the given program location. The quality of the solution will be rated both on how often it provides a correct answer for given program points as well as how thoroughly the answer covers the space of configurations for the given program point.

To enable a wide audience to tackle the challenge, we provide a range of difficulty levels with target systems of increasing complexity. Submissions may also choose to find configurations that reach a source file only instead of a program location, but are encouraged to work towards the latter. They may also focus on finding a single configuration or a partial space of configurations instead of the complete space of configurations. These are the options for target systems:

Easy We provide a benchmark consisting of C programs collected from the variability bug database [1][2]. The database consists of programs derived from real-world bugs or other defects, but they are very small and depend on a small number of configuration options such that a naive exhaustive enumeration is possible. Our benchmark, given in the accompanying repository, is made more challenging, because correct solutions should use the full set of configuration options given in `option_list.txt`. Thus, submitted solutions should work with this set of options when identifying a configuration, as described in the accompanying repository. This dataset may also be useful as a proof-of-concept for solution approaches other than the naive one.

Medium The axTLS web server[3] is a relatively small configurable system. Even so it has enough configuration options, 94, to make exhaustive search infeasible.

Hard The BusyBox toolkit[4] provides a single executable containing common GNU utilities. It is frequently used in embedded systems such as routers to provide a rich operating system with a small footprint. It has over a thousand configuration options.

Ultimate The Linux kernel source code[5] is arguably the most complex, configurable open-source project; it contains over 14,000 configuration options, a nightmare for scalability.

Ground truth is not provided for the target systems, because finding the ground truth is essentially equivalent to this very challenge. In lieu of ground truth, submitters can validate the correctness of their solution against the provided set of program locations by configuring the target systems and checking for the existence of the given program location. Instructions for obtaining source code, lists of locations for each target system, and a description of the input format are available in the accompanying repository for this challenge:

https://github.com/paulgazz/splc18challengecase

Given the ad-hoc nature of real-world build system implementations, repurposing the solution program for new systems may be tedious. Therefore the solution only needs to work on at least one of the target systems of the submitter's choice. For simplicity, solutions may focus on Boolean configuration options, making system-specific manual settings for non-Boolean options.

In your submission, please provide the following:

1. Which target system(s) the tool supports.
2. Whether your tool produces a single configuration per program location or a space of configurations.
3. Whether your tool can produce configurations for a source file and line number or just a source file.
4. The source code and easy-to-use instructions on building and running the tool. A virtual machine prepared to build and run the tool is a good option.
5. Running time measurements for the tool on the given program locations, including machine specifications.

---

[2] http://vbdb.itu.dk/#search/

[3] http://axtls.sourceforge.net/

[4] https://busybox.net/

[5] https://www.kernel.org/

(6) A script that runs the solution program for the given list(s) of program locations from the repository.

Solution evaluation. We will evaluate solutions by measuring how well the given program finds correct configurations of the program locations for the supported target program(s) as well as running time and resource usage[6]. For each program location, we will query the solution program for a configuration, build the configuration (if the solution program produced one), and check whether the program location is included. While a correct solution should be able to determine when a given program location is unreachable, we will only provide program locations that are reachable by some configuration. Each solution will be measured with the following formulae:

$$\text{Precision} = \frac{N_{\text{correct}}}{N_{\text{correct}} + N_{\text{wrong}}}$$

$$\text{Recall} = \frac{N_{\text{correct}}}{N_{\text{correct}} + N_{\text{missed}}}$$

$$\text{Accuracy} = \frac{N_{\text{correct}}}{N_{\text{all}}}$$

N_{all} is the total number of program locations we provide for a given target system. N_{correct} is the number correctly identified, i.e., the configurations that include the program location. N_{wrong} is the number incorrectly identified, while N_{missed} is the number for which the solution finds no configuration.

This pseudo-code describes how the above values will be computed during evaluation:

```
for each program location do
    ask the solution program to produce a .config;
    if the solution program fails then
        $N_{\text{missed}}$ += 1;
    else
        build the .config;
        if the program location is included then
            $N_{\text{correct}}$ += 1;
        else
            $N_{\text{wrong}}$ += 1;
        end
    end
end
```

For solution programs that produce a space of configurations, rather than just a single configuration, we will test the configuration space for correctness by sampling it and checking that all samples include the program location. All samples must include the program location to count towards N_{correct}, otherwise the answer counts towards N_{wrong}.

3 EXISTING TOOLS

Several existing tools analyze configurations in build systems and source code and may be useful as inspiration or used as components in a proposed solution. We have provided links to these tools in the accompanying repository.

Kmax [15] collects build system configurations for Kbuild-style Makefiles of the kind used in Linux and BusyBox. It uses static analysis to derive a Boolean expression of configuration options. KBuildMiner [4] also produces Boolean expressions for source files built with Kbuild Makefiles. It uses a heuristic parsing technique that trades off precision for speed. Makex takes a similar parsing approach [24]. KconfigReader [17] converts constraints on configuration options described in the Kbuild build system into Boolean expressions. Dietrich et al. describe GOLEM [11], which uses a dynamic approach to build system analysis, trying one or more configuration variables at a time to see which C files are enabled. GOLEM has been compared to other tools including KBuildMiner and Makex to evaluate coverage [12]. All of the above techniques consider only the build system, and do not inspect configurations within source files. Several static approaches based on new parsing algorithms exist to deal with preprocessor configurations within C source files [14, 16, 18].

The iTree tool [32] uses dynamic analysis and machine learning to construct an "interaction tree" and from the tree constructs configurations achieving high coverage. The iGen tool [25] dynamically infers *interactions*–logical formulae describing how configuration settings map to code coverage. The tool employs an iterative algorithm that runs the system, captures coverage data, processes data to infer interactions, and then generates new configurations to further refine interactions in the next iteration. The self-adaptive REFRACT [33] architecture monitors software for bugs and generates configuration guards to avoid the observed bugs. REFRACT works by monitoring a running program (e.g., Firefox), and when the program encounters a bug, REFRACT analyzes configurations encountered during the buggy run to create good configurations that do not exhibit the bug and also constructs guards for the program to avoid similar buggy behaviors.

Yilmaz et al. [37] describes several techniques and tools that use *covering arrays* to generate configurations. The UNL's CIT portal[7] provides a family of tools for generating and checking covering arrays using Simulated Annealing [7]. NIST's *Automated Combinatorial Testing for Software*[8] project provides ACTS [19] tool as well as precomputed covering arrays for standard configuration spaces.

4 RELATED WORK

Interaction discovery. Reisner et al. [29] developed the symbolic executor, Otter, and used it to fully explore the configuration space of a software system and extract interactions in conjunctive form. Symbolic execution, however, has scalability limitations and it is language specific. Consequently, several authors from the same group have started to use dynamic analyses and producing the aforementioned tools iTree and iGen that run much faster than Otter.

Lillack et al. describe a system that uses static analysis to derive a configuration map of source code [20]. Zhang and Ernst describe a tool that combines dynamic and static analysis to help users reconfigure a system undergoing software evolution [40]. Ouellet et al. describe static techniques to localize configurations of source

---

[6] Please see the repository's README.md file for specifications of the evaluation machine and running time expectations.

[7] http://cse.unl.edu/~citportal

[8] https://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software

code in avionics systems [28]. Several sampling approaches have been studied for exploring configuration spaces to find optimal configurations [22, 27, 31].

Feature interactions and presence conditions. Thüm et al [36] classify the feature interactions and presence conditions problems in software product line research. Since then, there have been multiple attempts to address these problems. Apel et al. [3] study the number of feature interactions in a system and their effects, including bug triggering, power consumption, etc. Lillack et al. [21] use (language-specific) taint analysis to find interactions in Android applications. Nadi et al [23] and von Rhein et al. [30] present tools that work with presence conditions that are already provided. Czarnecki and Pietroszek [9] check for well-formedness errors in UML featured-based model templates using an SAT solver.

Combinatorial interaction testing. Many researchers have explored combinatorial interaction testing (CIT) [6, 26, 34, 38], a family of techniques for testing a program under a systematically generated set of configurations. One particularly popular approach is called t-way covering arrays which, given a coverage *strength* t, generates a set of configurations containing all t-way combinations of option settings at least once. Over the last 30 years, many studies have focused on improving the speed, quality and flexibility of covering arrays [5, 8, 10, 13, 39]. Yilmaz et al. [37] used covering arrays to detect and characterize variability bugs in complex configuration spaces.

Build system analysis. As discussed in Section 3, several existing tools deal specifically with analyzing configurations in build systems [4, 15, 17, 24]. In addition, Tamrawi et al. [35] developed SYMake, a symbolic Makefile evaluator. SYMake generates a symbolic dependency graph from Makefiles for use in identifying code smells and in refactoring. Adams et al. [2] describe MAKAO, a visualization tool for Makefile dependencies. It extracts a concrete dependency graph for a single configuration.

ACKNOWLEDGMENTS

We would like to thank Julia Lawall for bringing up this challenge problem and providing concrete instances of it for the Linux source code. We would also like to thank the reviewer as well as the chairs Sarah Nadi and Timo Kehrer for organizing the challenge track.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 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).* ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2642937.2642990

[2] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France.* 114–123. https://doi.org/10.1109/ICSM.2007.4362624

[3] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD '13).* ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2528265.2528267

[4] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-code Mapping in Two Large Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10).* Springer-Verlag, Berlin, Heidelberg, 498–499. http://dl.acm.org/citation.cfm?id=1885639.1885698

[5] Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (Oct. 2006), 960–970. https://doi.org/10.1016/j.infsof.2006.03.004

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. 1996. The combinatorial design approach to automatic test generation. *IEEE Software* 13, 5 (Sept. 1996), 83–88. https://doi.org/10.1109/52.536462

[7] M.B. Cohen, Amanda Swearngin, Brady Garvin, Jacob Swanson, Justyna Petke, Kaylei Burke, Katie Macias, Ronald Decker, Wayne Motycka, and Zhen and Wang. [n. d.]. Combinatorial Interaction Testing Portal. ([n. d.]). http://cse.unl.edu/ cit-portal, Accessed on 2018-01-16.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. 2003. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings.* 38–48. https://doi.org/10.1109/ICSE.2003.1201186

[9] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06).* ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/1173706.1173738

[10] Gulsen Demiroz and Cemal Yilmaz. 2012. Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles.* 9–16.

[11] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12).* ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2362536.2362544

[12] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. 21–30. http://doi.acm.org/10.1145/2362536.2362544

[13] Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. 2011. Feedback Driven Adaptive Combinatorial Testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11).* ACM, New York, NY, USA, 243–253. https://doi.org/10.1145/2001420.2001450

[14] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05).* IEEE Computer Society, Washington, DC, USA, 379–388. https://doi.org/10.1109/ICSM.2005.23

[15] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/3106237.3106283

[16] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12).* ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/2254064.2254103

[17] Christian Kästner. 2016. KconfigReader. https://github.com/ckaestne/kconfigreader. (2016). [Online; accessed 12-Jan-2018].

[18] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11).* ACM, New York, NY, USA, 805–824. https://doi.org/10.1145/2048066.2048128

[19] Author: Richard Kuhn (NIST), Author: Raghu Kacker (NIST), and Author: Yu Lei (UTA). [n. d.]. Advanced Combinatorial Test Methods for System Reliability. ([n. d.]). https://csrc.nist.gov/publications/detail/journal-article/2010/advanced-combinatorial-test-methods-for-system-reliability DOI: https://content.csrc.nist.gov/publications/detail/journal-article/2010/advanced-combinatorial-test-methods-for-system-reliability.

[20] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14).* ACM, New York, NY, USA, 445–456. https://doi.org/10.1145/2642937.2643001

[21] M. Lillack, C. KÃďstner, and E. Bodden. 2017. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2756048

[22] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE).* ACM Press, New York, NY, 643–654. https://doi.org/10.1145/2884781.2884793

[23] S. Nadi, T. Berger, C. KÃďstner, and K. Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (Aug. 2015), 820–841. https://doi.org/10.1109/TSE.2015.2415793

[24] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to Detect Variability Anomalies in Linux. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, Washington, DC, USA, 107–116. https://doi.org/10.1109/CSMR.2012.21

[25] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. 2016. iGen: Dynamic Interaction Inference for Configurable Software. In *Foundations of Software Engineering (FSE)*. ACM, 655–665.

[26] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2 (Feb. 2011), 11:1–11:29. https://doi.org/10.1145/1883612.1883618

[27] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 61–71. https://doi.org/10.1145/3106237.3106273

[28] Maxime Ouellet, Ettore Merlo, Neset Sozen, and Martin Gagnon. 2012. Locating Features in Dynamically Configured Avionics Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1453–1454. http://dl.acm.org/citation.cfm?id=2337223.2337449

[29] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 445–454. https://doi.org/10.1145/1806799.1806864

[30] A. v Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 178–188. https://doi.org/10.1109/ICSE.2015.39

[31] Norbert Siegmund, Stefan Sobernig, and Sven Apel. 2017. Attributed Variability Models: Outside the Comfort Zone. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 268–278. https://doi.org/10.1145/3106237.3106251

[32] Charles Song, Adam Porter, and Jeffrey S. Foster. 2012. iTree: Efficiently Discovering High-coverage Configurations Using Interaction Trees. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 903–913. http://dl.acm.org/citation.cfm?id=2337223.2337329

[33] Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. 2014. Beyond the Rainbow: Self-adaptive Failure Avoidance in Configurable Systems. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/2635868.2635915

[34] Kuo-Chung Tai and Yu Lei. 2002. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 28, 1 (Jan. 2002), 109–111. https://doi.org/10.1109/32.979992

[35] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 650–660. http://dl.acm.org/citation.cfm?id=2337223.2337300

[36] Thomas ThÃijm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. 2004. Analysis strategies for software product lines. (2004).

[37] C. Yilmaz, M. B. Cohen, and A. A. Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering* 32, 1 (Jan. 2006), 20–34. https://doi.org/10.1109/TSE.2006.8

[38] C. Yilmaz, S. FouchÃľ, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc. 2014. Moving Forward with Combinatorial Interaction Testing. *Computer* 47, 2 (Feb. 2014), 37–45. https://doi.org/10.1109/MC.2013.408

[39] X. Yuan, M. B. Cohen, and A. 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

[40] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 152–163. https://doi.org/10.1145/2568225.2568251