Supporting Software Product Line Testing by Optimizing Code Configuration Coverage

László Vidács*, Ferenc Horváth†, József Mihalicza‡, Béla Vancsics† and Árpád Beszédes†
*MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary
Email: lac@inf.u-szeged.hu
†University of Szeged, Szeged, Hungary
Email: {hferenc,vancsics,beszedes}@inf.u-szeged.hu
‡NNG LLC, Budapest, Hungary
Email: jmihalicza@gmail.com

***Abstract*—Software product lines achieve much shorter time to market by system level reuse and code variability. A possible way to achieve this flexibility is to use generic components, including the core system, in different products in alternative configurations. The focus of testing efforts for such complex and highly variable systems often shifts from testing specific products to assessing the overall quality of the core system or potential new configurations. As a complementary approach to feature models and related combinatorial testing methods optimizing for feature coverage, we apply a source code oriented analysis of variability. We present two algorithms that optimize for high coverage of the common code base in terms of C++ preprocessor-based configurations with a limited set of actual configurations selected for testing. The methods have been evaluated on iGO Navigation, a large industrial system with typical configuration support for product lines, hence we believe the approach can be generalized to other systems as well.**

Keywords*—*Variability, Preprocessor, Configurations, Software Product Line, White box testing

I. Introduction

Software product lines are an efficient approach for achieving system level reuse [24]. Instead of separately developing distinct products for similar needs, in software product lines, a common code base contains the implementation of the core competence of the vendor, while actual features offered to customers are served by a customization layer on top of this core system. This setup results in increased overall complexity compared to standalone projects, but allows much shorter time to market by reusing components of previous projects.

In the C++ domain, projects may differ in their target platform, for example, where platform can mean both compilation environment or target operating system. Supporting various compilers on various platforms, both 32 and 64 bit and similar variations, using basically the same source code, this variability can be implemented by branching the code with preprocessor directives where needed. Besides specifying the target environment, functional differences are often configurable with preprocessor parameters as well: enabling or disabling (sub)modules, changing fundamental types (e.g. character type of strings or precision of arithmetics) or choosing between alternative strategies [18]. This preprocessor-level variability is a common technique in software libraries and software product lines [5], like the Linux Kernel [20].

Challenges for testing such highly configurable product lines are numerous, but they are mostly related to the vast number of possible SUT-s in the test process. This is particularly an issue at integration- and system testing levels, and could affect various test types including functional, non-functional, regression, maintenance, etc. An example situation is the following. Each concrete product uses a certain configuration of preprocessor parameters in the common core code. As part of the QA process, each product – and therefore the corresponding product configurations of the core code – are tested. This, however, does not help answering the following question: What is the risk of enabling feature F in product P? If F has not been used recently in a product, then the untested code of feature F suddenly becomes part of product P. On the other hand, if F is used in many products, the same core functionality is tested multiple times, individually for each product. Hence, product lines need targeted test strategies to handle variability [7], [25]. An optimal solution to this problem would be to test each and every configuration upon releasing any product (with a possibly updated list of enabled features). But, in many cases, this is unrealistic due to the prohibitively large number of possible configurations; hence, various selective retesting strategies are followed [19].

Product-line testing strategies based on feature models typically aim at producing high coverage in terms of features, which is, in a way, a black box approach [21]. However, it is usually not verified what is the effect of such strategies on the portion of the code base eventually included in testing. In this article, we offer a complementary approach, in which we select the configurations for testing on a white box-basis: by how much of the whole code base is included in the limited set of configurations. This way, we will be able to reduce the risk of untested features in any selected product.

We present two algorithms that automatically find configurations by maximizing preprocessor-level code coverage in C++-based projects. By extending the testing process with these optimal-coverage configurations, the amount of untested code drops significantly with moderate testing overhead. The algorithms can help reorganize test configurations so that the same functionalities are not tested unreasonably many times,

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)
978-1-4799-1885-0/15/$31.00

but originally undertested components receive enough focus in exchange. We also present the results of our experiments performed on iGO Navigation, a highly configurable and large scale system from the automotive and navigation domain.

The paper is organized as follows. Section II specifies our exact research goals, and describes the basic notions used in preprocessor-based variability analysis. Section III presents the two aforementioned algorithms, while Section IV shares the results of our experiments on the iGO Navigation system. We evaluate the results in Section V, and discuss related work in Section VI. Section VII concludes the paper.

II. Motivation and Approach

Our effort to develop a configuration search algorithm was motivated by testing challenges in the iGO Navigation product of NNG LLC [11], a large scale proprietary automotive system. The core system is written in C++ and is widely used in a vast amount of products from mobile to line-fit navigation, mainly in automotive projects with key brands. It has been developed for more than 10 years, and its size is more than 1.9M lines of code. The configuration of this system is very similar to that of the Linux kernel, where the set of built-in kernel functionalities are selected with preprocessor switches [29]. Many of the features can be modules, enabling run time load/unload. The navigation core is a library containing numerous different functionalities (such as address search, route calculation, map visualization, etc.), with both static (preprocessor-based) and run time variability management.

A specific iGO Navigation product configuration determines preprocessor switches to enable/disable different functionalities according to the target system environment and product design decisions. Besides these, there are preprocessor definitions that describe the target operating environment in detail, called platform properties. Module implementations use these platform properties to optimize their structure and execution for the actual needs. Over the years, challenges of high variability have been addressed in various ways. The system employs test automation solutions, however the thorough testing of one configuration still requires significant effort. In this environment, a relatively quick but extensive test would greatly increase the efficiency of testing efforts.

The main aim of our work was to devise a risk-based strategy in which the limited testing capacity for testing this system could be wisely used. This means, using minimal effort finding the most important configurations to test. We emphasize that although the method has been verified on this particular system, it is general enough to be applicable on other systems as well.

Finding the appropriate set of configurations and the search criteria in this project turned out to be difficult. Most of the previous approaches utilize the so called *Feature Model* (FM) approach [3], which represents all the possible products of SPL by declaring the constraints and relations between features. Usually the number of possible configurations grows exponentially with the number of features in the given FM leading to millions of possible products to deal with. Therefore, there was a need for methods which can automatically reduce the number of products that need to be tested. For example, generic combinatorial testing methods have been adapted to support SPL testing: the equivalent of combinatorial testing in the SPL environment is to use SAT solvers to generate products to test pairwise, or t-wise combinations of features [10].

While the FM approach provides a clean SPL representation, FM-based testing solutions have some limitations. First, feature models are very high level, specification oriented logical representations, which are not always well-documented or usable for the mentioned testing scenarios. The combinatorial testing approach needs the calculation of all t-wise sets, which is known to be NP-complete, so working with higher complexities ($t \geq 3$) is quite ineffective [13]. Furthermore, despite the popularity of pairwise testing in general, there are opinions that it is a poorly understood and ineffective technique [2]. Finally, the FM approach generally uses binary variables, where SAT solvers work efficiently, but these solutions cannot deal with complex variables as in our case.

Hence, we propose a complementary method which, instead of the black box-like approach of feature models, is code-oriented, i.e. white-box. The basic idea of the method is that in order to keep the overall code quality high with reasonable testing effort, we minimize the number of tested configurations while the source code covered by these configurations is kept as high as possible. Here, on 'covered source code lines' we mean physical code lines that belong to a given configuration (so, we do not deal with code coverage resulting from actually executing the test). In the following sections we provide details of the proposed general algorithms and how they have been implemented to work with the iGO Navigation system.

III. Search algorithms

A. Basic Concepts and Approach

In this section we seek for answers to the following problem: find N configurations that cover as much code as possible. Our base notions in this research are variables and configurations. We call *variables* all preprocessor macros appearing in conditional directives. A *configuration* is a set of variables with concrete values, where the value may be `undefined` as well. We call *configuration variables* preprocessor macros that are taking part in configuring the system; these are separated from other type of macros. We use a block-based model of the source code. Each (conditional) block starts with a preprocessor condition and contains all source code lines until a new conditional directive (`#if, #else, #elif` or `#endif`), or until the end of the file is reached. The content of the block is a black box from our point of view, only two important block properties are used: block condition (also called presence condition [15]) and block size measured in lines of code.

Considering conditional compilation, the notion of coverage is the following: a configuration *covers* a source code line if it satisfies the presence condition of the block containing the given line. Thus the total coverage of a given configuration can be determined by evaluating conditions of all blocks and computing the sum of block sizes of covered blocks. Note that we consider that block conditions contain not only the actual conditional expression of the local conditional directive, but all conditions accumulated which led to the actual condition from the beginning of the source file. For example, the nested block in the following

code snippet has $defined(B)$ as its local condition and $(\neg\, defined(A) \vee (defined(A) \wedge A < 5)) \wedge defined(B)$ as its accumulated global condition.

```
#if A >= 5
  ...
#else
  ...
  #if defined B
    ...
  #endif
#endif
```

We approach the problem via conditional blocks and individual configuration variables from two viewpoints in the following two ways:

a) Block-based approach: considering block conditions, we assign values to variables to cover largest blocks, and iteratively compose an output configuration.

b) Variable-based approach: we search the variable space and assign values to selected variables to increase coverage, and iteratively compose an output configuration.

We implemented one block-based and one variable-based algorithm. We expected long running time for the algorithms because of the complexity of the problem. Thus we designed greedy algorithms that make locally best decisions. Both algorithms work on pre-filtered blocks for speed optimization purposes. There are blocks with constant `true` or `false` conditions, these blocks are filtered out. There are blocks that contain `#error` directives (so called *error blocks*), which have to be avoided in configuration search, so they are also filtered out. Pre-filtering is done once at the beginning of the search process and requires one check for each block condition. Besides applying the error block filter, the algorithms regularly check error block conditions to skip any candidate configuration that covers any error blocks.

B. Block-based Algorithm

In this method each output configuration is composed iteratively. The pseudocode of the algorithm is listed in Figure 1. This greedy algorithm tries to cover the largest possible block in each iteration. It maintains a candidate configuration that contains the variable–value pairs already fixed. The condition of the largest uncovered block is examined and possible values of its variables are tried until candidate configuration extended with the new variable–value pairs (extended candidate) covers the given block. If the extended candidate does not cover any error blocks, then it is used as the new candidate and the set of covered blocks is updated.

When no new blocks can be covered, the candidate configuration is written to the output. The algorithm continues with composing a new configuration from the scratch, while maintaining the set of covered blocks from all previous iterations.

This algorithm heavily depends on the distribution of block sizes, since in each iteration the largest covered block determines the direction of the search.

C. Variable-based Algorithm

The variable-based algorithm selects appropriate variables to cover as much code as possible in a greedy manner. The pseudocode can be observed in Figure 2. Similarly to the block-based one, this algorithm composes the output configurations iteratively. In each iteration the algorithm tries to extend the candidate configuration by one variable–value pair. To find the best pair, the algorithm computes all possible extensions of the candidate configuration and chooses one pair with the highest code coverage. If the extended candidate does not cover any error blocks, then it is used as the new candidate and the set of covered blocks is updated.

When no more valid extensions can be found, the candidate configuration is written to the output. The algorithm continues with composing a new configuration from scratch, while maintaining the set of covered blocks from all previous iterations.

Input: N ▷ Number of output configurations
Input: S ▷ Set of code blocks
Output: C ▷ List of N output configurations

1: **algorithm** BLOCK-BASED ALGORITHM
2: $i \leftarrow 0$
3: $BLK \leftarrow$ SORT(FILTER($S, desc\_by\_size$))
4: $COV \leftarrow \emptyset$
5: **while** $i < N$ **do**
6: $X \leftarrow$ empty configuration
7: **for all** $b_{max} \in BLK \setminus COV$ **do**
8: **for all** $(v_1, x_1) \dots (v_n, x_n)$: $v_i \in$ COND(b_{max}), $x_i \in$ VAL(v_i) **do**
9: $X' \leftarrow X \cup (v_1, x_1) \dots (v_n, x_n)$
10: **if** (X' satisfies b_{max}) $\wedge$ (X' does not satisfy any error conditions) **then**
11: $X \leftarrow X'$
12: $COV \leftarrow COV \cup \{c \in BLK : c$ covered by $X'\}$
13: **break**
14: **end if**
15: **end for**
16: **end for**
17: $C_i \leftarrow X$
18: $i \leftarrow i + 1$
19: **end while**

Figure 1. Block-based configuration search algorithm

IV. EXPERIMENTS

A. Variability Properties of the iGO Navigation System

The iGO Navigation system has more than 60 active configurations that have been released and still maintained. The developers implemented a proprietary variability analyzer tool to perform the static analysis of preprocessor conditions. Most important features of the analyzer are that (1) block conditions can be simplified using heuristics; and (2) configurations and block conditions can be evaluated to query whether a block is part of a given configuration. The analyzer first produces a variability model with blocks and their conditions, which is then processed by the search algorithms via the analyzer API.

The iGO Navigation variability model contains 46,810 blocks, while in the conditions 1,200 configuration variables are used. The search algorithms filter out three type of blocks at the beginning (constant true, false and error blocks). Conditions of the remaining blocks contain both configuration variables and other variables. Since we use only configuration variables, from the coverage point of view three types of conditions can be distinguished. Configuration conditions contain only configuration variables, thus it is expected to cover these blocks. Mixed conditions contain both types of variables, so coverage is possible only if the condition can be satisfied

Input: N ▷ Number of output configurations
Input: V ▷ Set of configuration variables
Input: S ▷ Set of code blocks
Output: C ▷ List of N output configurations

algorithm VARIABLE-BASED ALGORITHM
  $i \leftarrow 0$
  $BLK \leftarrow$ FILTER(S)
  $COV \leftarrow \emptyset$
  while $i < N$ **do**
    $X \leftarrow$ empty configuration
    while $V \setminus$ VAR(X) $\neq \emptyset$ **do**
      $X_{max} \leftarrow X$
      $COV_{max} \leftarrow COV$
      for all $(v_i, x_i) : v_i \in V \setminus$ VAR(X), $x_i \in$ VAL(v) **do**
        $X' \leftarrow X \cup (v_i, x_i)$
        if X' does not satisfy any error conditions **then**
          $COV_{X'} \leftarrow COV \cup \{b \in BLK : b$ covered by $X'\}$
          if $|COV_{X'}| > |COV_{max}|$ **then**
            $X_{max} \leftarrow X'$
            $COV_{max} \leftarrow COV_{X'}$
          end if
        end if
      end for
      if $|COV_{max} \setminus COV| > 0$ **then**
        $X \leftarrow X_{max}$
        $COV \leftarrow COV_{max}$
      else
        break
      end if
    end while
    $C_i \leftarrow X$
    $i \leftarrow i + 1$
  end while

Figure 2. Variable-based configuration search algorithm

using configuration variables (for example there is a fortunately placed logical `or` in the condition). Other conditions contain only non-configuration variables, so these cannot be covered by our algorithms. The distribution of condition types in the system can be seen in Table I.

Table I. SYSTEM SIZE – DIVIDED INTO CONDITION TYPES

Condition type	Blocks	LOC
Filtered condition (`T, F, #error`)	11,847	682,300
Configuration condition	22,067	920,926
Mixed condition	10,085	271,710
Other condition	2,811	50,064
Total	46,810	1,925,000

Regarding the block sizes, the majority (46,184 blocks, 98.7%) of the blocks are below 500 LOC, while the most frequent block size with 12,931 occurrences is 2. The minimum and maximum block sizes are 1 and 10,428 LOC respectively, and the average is 41 LOC.

B. Experiments with 10 Configurations

First set of experiments were performed with $N = 10$, since 10 configurations may be reasonable to test and we also expected to see coverage trends in 10 iterations. In Figures 3 and 4 coverage of all 10 configurations can be observed along the x axis in the order they were produced by the block-based and variable algorithms respectively. The y axis shows lines of code values. In these diagrams the dashed lines of Total LOC and Config LOC have constant values as they represent system properties serving as baselines. *Total LOC* denotes the total lines of code belonging to blocks with configuration and mixed conditions. This level of coverage cannot be reached, because mixed conditions cannot be fully covered by configuration variables. *Config LOC* denotes the total lines of code contained by blocks with configuration condition, which level of coverage is addressed in our research.

The other three lines show actual coverage values in each iteration. *Delta coverage LOC* denotes the coverage added by the given configuration compared to the coverage provided by all previous configurations. *Total coverage LOC* line shows the overall coverage achieved by all configurations including the actual one. *Config coverage LOC* means the overall coverage considering only blocks with configuration condition.

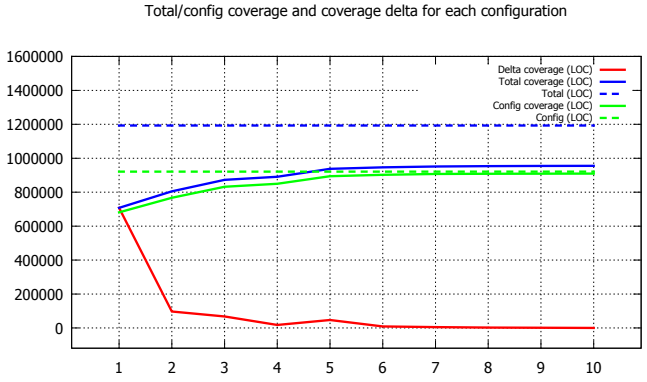

Figure 3. Coverage results of the block-based algorithm (lines of code)

Figure 3 shows that the first configuration of the block-based algorithm covers 74% of Config LOC itself (see green dashed and solid lines). From that point the coverage still increases, but after the fifth configuration we see modest increase only. Config coverage does not reach Config LOC in this experiment. Counting all 10 configurations, the Config coverage value is still **98.73%** of Config LOC. However, this is far above than what we expected. On the other hand, interesting is to observe the parallel curve of Total coverage and Config coverage lines. In the fifth iteration the Total coverage exceeds the Config LOC values, since the algorithm covers several mixed blocks as well.

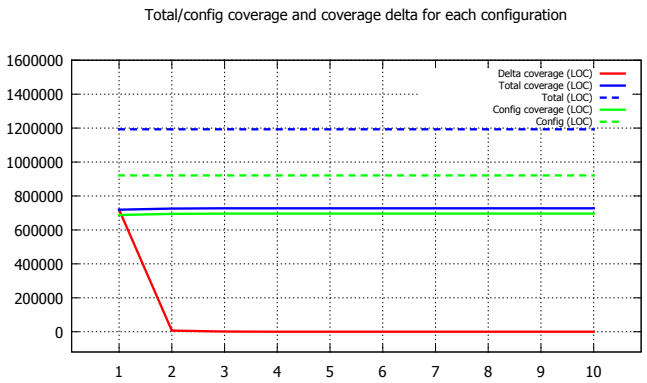

Figure 4. Coverage results of the variable-based algorithm (lines of code)

Results of the variable-based algorithm are presented in a similar diagram in Figure 4. The first configuration produces even slightly better coverage than the first one of the block-based algorithm. Unfortunately the coverage increase in the

subsequent configuration is too small. Starting from the 4th configuration, the algorithm does not cover any new source code lines. Thus the Config coverage result of this algorithm remains at the **75.56%** level.

The total running time of the tool consists of the time spent by the variability analyzer tool and the search algorithm. The analyzer produced the variability model in $\sim$5 hours. However, this model is used for other purposes as well, as such we consider it available for configuration search. The runtime performance of the algorithms was between 22 and 26 minutes, which is acceptable, since testing (and even building) 10 configurations takes much more time, so to find good configurations is far profitable.

C. Experiment with 50 Configurations

Given the acceptable search time for 10 configurations, in the second phase the block-based algorithm was performed to find $N = 50$ configurations. The variable based algorithm stopped to increase coverage after 3 configurations, so it was omitted from this experiment.

After the initial 10 configurations, only small parts of the code are covered in each step by the block-based algorithm. Figure 5 shows that Config coverage slowly approaches the Config LOC line (green solid and dashed lines). What we cannot observe in the diagram: after the 24th configuration in each step less than 100 (sometimes only 2-3) additional source code lines are covered. The total search time in this larger experiment was about 124 minutes, which means that we experienced roughly linear behaviour.

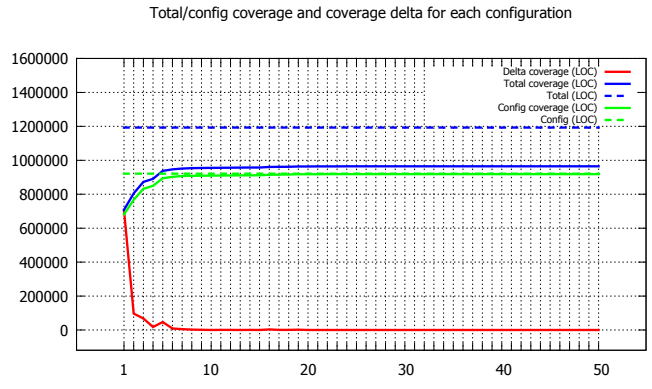

Figure 5. Coverage results of the block-based algorithm (LOC, $N = 50$)

V. Evaluation

A. Evaluation of Results

The block-based algorithm performed better in our experiments. Its average search time was below the time of the variable-based algorithm. Both algorithms run in an acceptable time frame, so more complex heuristics could be used in future versions. The first configurations found by the block-based algorithm cover slightly less lines of code than the ones found by the variable-based algorithm. On the other hand, the block-based algorithm is not limited to 3 configurations. Comparing the results after 10 configurations the **75.56%** coverage by the variable based algorithm is not acceptable, while the **98.73%** coverage of the block-based algorithm is a solid result which makes the search algorithm applicable to use in practice.

The advantage of the block-based algorithm is that in each step it looks for a combination of more than one variables at a time (as many variables as the block condition contains). On the contrary, the variable-based algorithm tries to find the best variable in each step. To find the best variable, it evaluates all free variables by computing possible coverages. This is the reason for the longer running time, unfortunately is seems that for the subject system this search does not pay off. To overcome this one variable at a time limitation we propose a solution in the next section. On the other hand, the advantage of this algorithm is its variable-based logic, that may help to avoid local maxima, which traps the other algorithm that greedily consumes largest blocks.

Besides the promising coverage results, we need to know whether we produce good configurations. For a resulting configuration it is not expected to be similar to an already released configuration of the product. Instead, the expectation is that the produced configurations have to be built successfully to enable testing. The development policy of the company forces the use of error conditions to exclude invalid configurations that produce build time or runtime errors. Developers create a conditional block for unsupported combinations of variables and their values. The body of these blocks contains an `#error` directive to prevent successful build. Hence testing build results is outside of the responsibility of the algorithms, they rely on well placed `#error` directives, giving the responsibility back to the developers. We expect that by improving the error markers the convergence to 100% coverage will be slower, but the quality of the produced configurations will be better.

B. Plans for Improvement

An important outcome of this research is the list of future ideas/actions to achieve much better results. We identified two major issues during the experiments: improvement needed for variable-based algorithm and resulting configurations need to be validated. The first issue is the one-variable problem with the variable-based algorithm. One way of improvement is to guess for more variables and values at a time. However, in this case the exhaustive search for best coverage gain in each step is becoming less feasible. Another way we propose is to develop a hybrid algorithm, which starts with a variable-based variant, and at the point when the variable-based search has no new results we switch to the block-based algorithm.

To improve the quality of the configurations we rely on error conditions. We plan to enhance the use of error markers in the code and check our algorithms on the changed program. To achieve better error markers we plan to use automatic configuration build tests in a controlled way. During the controlled build tests the configuration space is traversed and selected configurations are tried to build the code. In case of build failure the configuration is reported to developers to handle the issue and insert `#error` directive if necessary.

There are two less crucial but still desired ideas for future work. One direction is to further improve simplifier heuristics of the analyser to increase performance. The second direction is enabled by the fair running time of greedy algorithms: we plan to develop search algorithms that are not limited to

finding local maxima. In fact even 24 hours running time or an iterative, long running algorithm is acceptable if we gain on testing efforts.

C. Generalizability and Threats to Validity

We identified several threats that can affect the validity of our results. Our subject system uses typical preprocessor-based variability solutions. In addition, it holds the properties of a typical industrial SPL, the problems raised by the iGO Navigation system may be representative among preprocessor-based SPLs. On the other hand, iGO Navigation is a proprietary software of one concrete domain, since we cannot conclude that similar results can be achieved on systems with different variability policies.

Although our approach and the proposed algorithms are generally applicable, there are two restrictions introduced by iGO Navigation developers to decrease the complexity of preprocessor-based variability solution. First, configuration variables may not be function-like macros. There are several examples of such macros in large projects (like the GCC compiler). This issue can be addressed in the analyzer, and by introducing project-specific helper variables. Second, the domain of these variables is restricted to integer values only. In practice, most variables have only enabled and disabled states, only some of them may have more values. Since this restriction applies only to configuration variables, this integer-restricted model is a limitation only when configuration variables implement some kind of metaprogramming at the preprocessor level, which usage is rare in our experience. On the other hand, the usual approach utilizing feature models is more restricted using binary variables only. In addition, this limitation is due to optimization heuristics, which can be extended to handle a more general case as well. Another threat is that resulted configurations are not yet tested as the number of error blocks need to be increased. After the use of error markers is improved throughout the code, we plan re-run the experiment and measure the testing capabilities of resulting configurations.

VI. Related work

The C/C++ preprocessor enhances the C and C++ languages by lightweight meta-programming capabilities. It is widely assumed that the variability mechanism of the preprocessor tool is used quite frequently in the implementation of large variable software systems from different domains such as operating systems, database systems, or compilers [18]. But using preprocessor directives is not the only way of handling variability. We refer to the paper of Thüm *et al.* [30] for an in depth survey on variability solutions. The interested reader is also referred to the recent survey on search based software engineering for SPLs by Harman *et al.* [9].

Testing software product lines also requires special strategies that take care of variability properties of these systems. For an overview we refer to the works of Neto *et al.* [21] and Engström *et al.* [7]. Several researchers tackling the problems of SPL testing rely on classical approaches, e.g. feature models and t-wise testing [12]. However, t-wise testing of SPLs is a difficult task, hence constraint solving methods have been put to use. Perrouin *et al.* [23] proposed a solution based on a SAT solver. Oster *et al.* [22] improved the predictability of similar approaches by transforming the feature model and using CIT algorithms. As a recent result, SPLCAT [12] looks promising because it can handle larger FMs, its performance can be seen in the experiments conducted on the Linux kernel. An optimization of SPLCAT has been proposed by Johansen *et al.* [13], it performs well on FMs of all sizes but regarding t-wise testing it is limited to $t = 3$. Due to the computational complexity of t-wise testing some of the available solutions have weaknesses in terms of performance. Usually, the most prevalent issues are the following: scalability, flexibility, predictability of the generated solutions, restrictions of SAT solvers, and handling of large sized FMs and higher level of interactions during t-wise testing. Therefore, researchers are constantly aiming for different solutions, for example, prioritization or search-based heuristics. Henard *et al.* [10] proposed a combinatorial testing approach which is able to handle feature models with larger sizes. It utilizes flexible prioritization models based on a t-wise maximizing fitness model. It is also features a scalable search-based technique to generate products from the given FM.

Tartler *et al.* [29] presented an approach for determining the smallest set of configurations that can achieve almost full code coverage. The presented work is the most similar approach to ours. A naive and a greedy algorithm is proposed to find high coverage configurations in the Linux kernel. The algorithms are based on similar pirnciples to ours like preprocessor variables and conditional blocks. However, they have some limitations compared to our method, namely, they uses only binary variables (passed to the SAT solver), the granularity is at the block level, and there is no distinction between blocks based on their size. This approach is on the mid way between the feature models and our source code line level approach. A common outcome of their and our work could be a proposal of hybrid algorithms in future.

Kästner *et al.* developed a variability analyzer used for mining features [14], which is a promising direction of filling the gap between preprocessor configurations and feature models. Krone and Snelting [16] analysed the complex configuration structures created with directives and produced a graphical output of them. Concept lattices were used to help in reengineering configurations [27]. Latendresse [17] proposed a solution for finding the conditions required for a particular source line to get through the conditional compilation. This approach promises efficient symbolic evaluation algorithm with linear time complexity. The C-CLR tools are Eclipse plugins that provide source code views on user selected configurations [26]. The Sunifdef command line tool [28] attempts to eliminate or simplify conditional directives based on defined macro values. There are several other related ideas implemented to ease the understanding and handling preprocessor-based configurations. Baxter and Mehlich introduced a method for removing unnecessary conditional directives based on rewrite rules [4]. The Columbus preprocessor schema [31] defines both dynamic and static representation of directives. The latter could be used for variability analysis as well. Additional techniques to help handling configurations include IDE integration of analysis [6], background coloring of conditional [8], program slicing adopted on preprocessor macros [32], [33] or refactoring conditionals into aspects [1].

VII. Conclusions

In this paper we addressed an important issue of testing and maintaining software product lines, namely, how to produce a small number of configurations with high coverage of the code base. We presented two types of algorithms for the task and evaluated them on the iGO Navigation system, a large industrial project featuring a preprocessor-based variability solution with 1200 configuration variables. We found that the runtime performance of these greedy algorithms is acceptable, and that in the case of a block-based algorithm the 10 resulting configurations cover 98,73% of configurable source code lines. During the experiments we identified concrete improvement possibilities in our algorithms. Main future direction of our work is to improve the quality of the results through enhanced error conditions.

References

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 243–254, New York, NY, USA, 2009. ACM.

[2] J. Bach and P. J. Schroeder. Pairwise testing - a best practice that isn't. In *In Proceedings of the 22nd Pacific Northwest Software Quality Conference*, pages 180–196, 2004.

[3] D. Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin Heidelberg, 2005.

[4] I. D. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 281, Washington, DC, USA, 2001. IEEE Computer Society.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[6] R. Dévai, L. Vidács, R. Ferenc, and T. Gyimóthy. Service layer for IDE integration of C/C++ preprocessor related analysis. In *Computational Science and Its Applications - ICCSA 2014*, volume 8583 of *Lecture Notes in Computer Science*, pages 402–417. Springer International Publishing, Jun 2014.

[7] E. Engström and P. Runeson. Software product line testing–a systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[8] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.

[9] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 5–18, New York, NY, USA, 2014. ACM.

[10] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Softw. Eng.*, 40(7):650–670, July 2014.

[11] Homepage of NNG LLC. `http://www.nng.com/`, 2015.

[12] M. F. Johansen, O. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 638–652, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] M. F. Johansen, O. Haugen, and F. 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.

[14] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semiautomatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014.

[15] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: Toward type checking #ifdef variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.

[16] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of ICSE 1994, 16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society, 1994.

[17] M. Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 170–179. IEEE Computer Society, March 2003.

[18] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 105–114, New York, NY, 5 2010. ACM.

[19] S. Lity, M. Lochau, I. Schaefer, and U. Goltz. Delta-oriented model-based spl regression testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering*, PLEASE '12, pages 53–56, Piscataway, NJ, USA, 2012. IEEE Press.

[20] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the linux kernel variability model. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2010.

[21] P. A. D. M. S. Neto, I. do Carmo Machado, J. D. McGregor, E. S. De Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[22] S. Oster, F. Markert, and P. 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.

[23] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. 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.

[24] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[25] P. Runeson and E. Engström. Regression testing in software product line engineering. *Advances in Computers*, 86:223–263, 2012.

[26] N. Singh, C. Gibbs, and Y. Coady. C-CLR: a tool for navigating highly configurable system software. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 9, New York, NY, USA, 2007. ACM.

[27] G. Snelting. Software reengineering based on concept lattices. In *CSMR*, pages 3–10, 2000.

[28] Sunifdef homepage. `http://sf.net/projects/sunifdef/`, 2015.

[29] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. *SIGOPS Oper. Syst. Rev.*, 45(3):10–14, Jan. 2012.

[30] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):Article 6, 6 2014.

[31] L. Vidács, A. Beszédes, and R. Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering)*, pages 75–84. IEEE Computer Society, Mar. 2004.

[32] L. Vidács, A. Beszédes, and R. Ferenc. Macro impact analysis using macro slicing. In *Proceedings of ICSOFT 2007, International Conference on Software and Data Technologies*, pages 230–235, July 2007.

[33] L. Vidács, A. Beszédes, and T. Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, May 2009.