ABSTRACT

A Software Product Line (SPL) is a family of programs. Testing an SPL is a challenge because the number of programs to examine may be exponential in the number of features. However, there are features whose absence or presence has no bearing on the outcome of a test. We can ignore such irrelevant features and consider combinations of only the remaining features, thereby eliminating unnecessary test runs. In this paper, we propose a product line representation that enables a conventional static program analysis to be applied. We then present a classification of features that can be used to narrow down the search for relevant features. Conditions of relevance that a static analysis can check are outlined and a procedure that uses the set of relevant features to reduce the combinatorial number of programs to test is sketched.

1. INTRODUCTION

A Software Product Line (SPL) is a family of programs where each program is defined by a unique combination of features. Developing a set of programs with commonalities and variabilities in this way can significantly reduce both the time and cost of software development. However, as the number of programs may be exponential in the number of features, testing an SPL, the phase to which the majority of software development is dedicated, becomes especially challenging [12].

Indeed, scale is the biggest challenge in testing or checking the properties of programs in a product line. Even a product line with just 10 optional features has over a thousand ($2^{10}$) distinct programs. As an example of a situation where every program must be considered, suppose that every program of an SPL outputs a String that each feature might modify. Every possible feature combination must be checked to see if the output always conforms to a particular pattern.

The state-of-the-art often relies on sampling, checking feature combinations that have a higher chance of falsifying certain properties [6, 7, 11]. While this is practical, it may overlook critical combinations. Another approach is to apply traditional verification techniques directly – model checking [9, 16] or bounded exhaustive testing [3, 17] – on every product of an SPL. Unfortunately, feature combinatorics often render brute force impractical. Yet another complicating factor is that features often have no formal specifications; even contracts, a relatively lightweight form of specification, are typically unavailable.

Current approaches fail to exploit the defining characteristic of features, i.e., a feature is an increment in functionality. Features add new functionality, but typically do not invalidate existing functionality. We hypothesize that this is the key to reducing the number of configurations to test. For example, suppose that 8 of the 10 features in the example mentioned do not modify the output String and thus are irrelevant. We need only run the String test on only $2^2 = 4$ programs, rather than a thousand.

In this paper, we explain the concept of irrelevant features. We represent an SPL in a form where conventional program analyses can be applied and we outline conditions of relevance that allows a static analysis to identify the relevant features. We then describe how this information may be used to reduce the configurations to test without reducing the test’s ability to find bugs.

2. MOTIVATING EXAMPLE

Product Line. Figure 1 shows a product line of games where a player accumulates points by discovering treasure. Each feature’s code is given a distinct color. There are four features in the product line:

- **Base** (clear color) introduces Player class, which defines what happens when a treasure is found and how reward and penalty are computed, and SuperPlayer, which, despite the name, is actually just a player with more points than a normal player.

- **Novice** (blue) eases the game play by introducing bonus points (bonus field is introduced in lines 6-7 and incremented in lines 12-13) and reducing a SuperPlayer’s penalty (the feature introduces its own version of penalty() in lines 36-42 that overrides Base’s version).

- **Limit** (yellow) puts a ceiling on the return value of Player.reward() and SuperPlayer.penalty().

- **Fatigue** (red) considers finding a treasure a form of labor and subtracts a small number of points (lines 14-15).

SysGen Representation. There are different ways of representing a product line. In this paper, we use a SysGen
Figure 1: Example Product Line

Figure 2: Product Line Tests

tests for our product line. Test1 checks that there is no bonus when a super player is created. Test2 checks the penalty for a super player with 200 points. Test3 makes a regular player discover a treasure worth 100 units and checks that the player has at least that many points afterwards.

Although a test may check many functionalities, we assume a setting where it exercises a small portion of a product line, the way a unit test does. We consider a test to have all of its inputs (except the Boolean feature variables which are used to instantiate a program) fixed.

Feature Combinatorics. The key to making product line testing practical is eliminating unnecessary feature combinations and we do this by determining the features relevant to a test. The running example helps us understand intuitively what “relevance” is. For example, in Test1, only Base and Novice are relevant because only these features’ code is reachable from Test1.main(). The relevant features are less obvious for Test2 and Test3 but they can still be statically determined. We begin by presenting a classification of features that helps narrow down the search for relevant features.

3. CLASSIFICATION OF FEATURES

A feature is relevant if its code can influence the test outcome, meaning that we need to run the test once with the feature set to true and another time with the feature set to false as the test results may be different across these two runs (we will elaborate on this in Section 4). Although every feature’s code can be analyzed for relevance, there is no need to do so because from the SysGen program, the feature model and the test we are given, we can considerably reduce the set of features whose code needs to be analyzed. We classify features with this goal in mind.

A feature is bound if its truth value is fixed for a given test. Before determining bound features, a feature model is specialized, i.e. its feature combinations are reduced, by having implementation constraints added according to [15] to ensure that the test will compile. The feature model may be further specialized by a tester adding test constraints to the feature model to require that certain features must always be present or absent when running the test. The com-
Figure 3: Classification of Features

plete set of bound features are then determined by mapping the specialized feature model to a propositional formula [2] and using a SAT solver to propagate constraints. Unbound features, whose truth values are not fixed, are just the complement of bound features.

Of the unbound features, only the features whose code is reachable (i.e. executable) from the test’s entry point (main method) need to be checked for relevance. An off-the-shelf static analysis that computes a call graph and determines the transitive callees of each statement of the main method can be used to identify reachable features.

Figure 3 shows the classification of features. Ineffective features are reachable features that are not relevant. An irrelevant feature is any feature that is not relevant (i.e. ineffective, unreachable, and bound) and thus we need only consider one truth value for it when running the test. In other words, whether the test passes or fails is independent of whether an irrelevant feature is present or not. We give ideas on how to identify relevant features in Section 4 but for now, it is apparent that:

• In Test1, the test references SuperPlayer (which belongs to Base) and Playerbonus (Novice), so Base and Novice are bound to true. Note that Base is mandatory anyway due to the feature model. Fatigue and Limit are unbound. Also, these two features are unreachable as their code is not executed through the test.

• In Test2, only Base (true) is bound. The test references SuperPlayer.penalty() of Novice, but the method definition need not exist as Base provides Player.penalty(). Therefore, Novice is unbound. However, if the tester wanted to test only the former method definition, she could add the test constraint Novice=true to the feature model. Novice and Limit are reachable features.

• For Test3, only Base (true) is bound. All the three unreachable features are executable. For example, Limit is reachable as reward() is called by Novice.

Suppose \( n \) is the number of unbound features in the entire product line, \( u \) the number of unbound features in the test, \( r \) the number of reachable features, and \( R \) the number of reachable features. The number of programs to test is at most \( 2^R \), where \( 2^R \leq 2^u \leq 2^n \).

4. CONDITIONS FOR RELEVANCE

A reachable feature is relevant to a test if it influences the test result. Conversely, an ineffective feature is simply a reachable feature that is not relevant. A feature will be ineffective if it does not change the control-flow or data-flow of another reachable feature’s code. Control-flow is represented using a control-flow graph (CFG), a directed graph whose nodes are basic blocks that consist of straight-line code. An ineffective feature adds code to existing basic blocks without introducing edges between them, thereby preserving the shape of the graph itself. Data-flow is represented using a graph of def-use pairs [1]. A feature preserves def-use pairs if it writes only to its own variables. For example:

• For Test1, because there is no reachable feature as explained before, there is no relevant feature and consequently, only one configuration with Base and Novice bound to true, such as \{Base=true, Novice=true, Fatigue=false, Limit=false\}, needs to be run for the test.

• For Test2, Novice and Limit, the two reachable features, are relevant as the former changes the CFG by replacing a called method with its own method and the latter adds an edge to a CFG to exit early.

• For Test3, Limit, Fatigue, and Novice are reachable. Fatigue is relevant because it alters a variable (points) of another feature (Base). Limit is relevant because it changes control-flow of reward() called from line 13. Novice is relevant as it allows code of another relevant feature (Limit) to be reached. With three relevant features, Test3 must be run on up to seven configurations.

Further details are given in [10].

5. CONFIGURATIONS TO TEST

We can identify the configurations to run the test on using the relevant features and the feature model specialized for the test. We can use an off-the-shelf SAT solver like SAT4J [14] to enumerate all combinations of the relevant features, treating the rest of the features as don’t-cares. Configurations disallowed by the feature model are not considered. Finally, for each of the configurations to test, we create a concrete program corresponding to it from the SysGen program, and run the test against the concrete program.

Examples. For each of the three example tests, without our technique, all seven configurations in the original feature model would have to be tested. With our technique, for Test1, just one configuration, \{Base=true,Novice=true,-Fatigue=false,Limit=false\}, must be examined. For Test2, configurations representing the four combinations of the relevant features Novice and Limit must be tested. For Test3, all seven configurations must be tested as every feature except Base is considered relevant.
6. RELATED WORK

Model-Checking. Classen et al.[4] recently proposed a technique to check a temporal property against a product line that is in the form of Feature Transition Systems (FTS), which is a preprocessor-like representation like SysGen. Our technique and their technique are different in that theirs works on a representation (transition systems) and setting (verifying temporal properties) different from those (object-oriented programs and testing) that ours works on and thus the two techniques are complementary.

Sampling. Sampling exploits domain knowledge, rather than program analysis results, to select configurations to test [6, 7, 11]. While these approaches can miss problematic configurations, our work cannot as we exhaustively, but without redundancy, examine the feature combinations.

Feature Interactions. [5, 8] present static analyses that check whether a feature modifies behavior of another feature, which is clearly similar to (ir)relevance. However, these techniques work on conventional aspect-oriented programs, where all modules are required for the program to work, which is sharply different from SPLs. Indeed, the prior works performed analysis more for modular reasoning, rather than for reducing combinatorics in product line testing.

Reducing Testing Effort. There is also related work on reducing testing, typically using output from some analysis, although such work is not in the context of product lines. For example, a regression testing technique like [13] identifies a subset of existing tests to run given a program change or a feature. We address the opposite problem, i.e. we identify a subset of existing features to run given a test. The two techniques are complementary as both settings can occur.

7. CONCLUSIONS

Software Product Lines (SPLs) represent a fundamental approach to the economical creation of a family of related programs. Testing SPLs is more difficult than testing conventional programs because of the combinatorial number of programs to examine. Our insight is that every test is designed to evaluate one or more properties of SPL programs. A feature might alter any number of properties. In SPL testing, determining whether a particular feature is relevant to a property (test) or not is the critical problem.

We sketched a procedure to test a product line. Given a test, we determine the features that need to be bound for it to compile. This already reduces configurations to test. Of the unbound features, we determine the features reachable from the entry point of the test, further reducing the configurations. And of the reachable features, we determine the features that affect the properties being evaluated, reducing the configurations further.

Acknowledgements. Kim is supported by an NSERC Postgraduate Scholarship. Kim and Batory are supported by NSF’s Science of Design Project #CCF-0724979. Khurshid is supported by NSF #CCF-0845628.

8. REFERENCES


