

Reducing Combinatorics in Testing Product Lines

Chang Hwan Peter Kim
University of Texas at Austin
Austin, TX 78712 USA
chpkim@cs.utexas.edu

Don Batory
University of Texas at Austin
Austin, TX 78712 USA
batory@cs.utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712 USA
khurshid@ece.utexas.edu

ABSTRACT

A *Software Product Line (SPL)* is a family of programs where each program is defined by a unique combination of *features*. Testing or checking properties of an SPL is hard as it may require the examination of a combinatorial number of programs. In reality, however, features are often *irrelevant* for a given test — they augment, but do not change, existing behavior, making many feature combinations unnecessary as far as testing is concerned. In this paper we show how to reduce the amount of effort in testing an SPL. We represent an SPL in a form where conventional static program analysis techniques can be applied to find irrelevant features for a test. We use this information to reduce the combinatorial number of SPL programs to examine.

1. INTRODUCTION

A *Software Product Line (SPL)* is a family of programs where each program is defined by a unique combination of features. By developing a set of programs with commonalities and variabilities in a systematic way, SPLs can significantly reduce both the time and cost of software development. But at the same time, SPLs require software engineering techniques distinct from those for conventional programs. In particular, testing, the phase to which the majority of software development is dedicated, becomes especially challenging [26].

The most obvious challenge in testing or checking the properties of programs in an SPL is scale: an SPL with only 10 optional features has over a thousand (2^{10}) distinct programs. The need to assume the worst-case and test all programs is evident in the following scenario: suppose that every program of an SPL outputs a String that each feature might modify. To see if the output always conforms to a particular pattern, every possible feature combination must be tested.

Current practice often focusses on feature combinations that are believed to have a higher chance of falsifying certain properties [10][11][25]. In light of no other information, this

is reasonable but critical combinations may be overlooked. Another approach is to apply traditional verification techniques directly — model checking [16][35] or bounded exhaustive testing [7][37] — on every product of the SPL. Again, feature combinatorics render brute force impractical. Yet another complicating factor is that features often have no formal specifications; even contracts are typically unavailable.

Given this dismal situation, it is still possible to improve the state-of-the-art by leveraging the semantics of *features*, i.e. increments in functionality. It is well-known that there are features whose absence or presence has no bearing on the outcome of a test. Such features are *irrelevant* — they augment, but do not invalidate, existing behavior. To illustrate potential benefits, suppose we determine that 8 of the 10 features in the above example do not modify the output String and thus are irrelevant. We can confidently run the String output test on only $2^2 = 4$ programs to analyze the entire product line, instead of a thousand.

In this paper, we explore the concept of irrelevant features to reduce SPL testing. We find features that do not influence the result of a given test (these features are irrelevant). We accomplish this by representing an SPL in a form where conventional program analyses can be applied, determining the features that are irrelevant for a given test, and pruning the space of such features to reduce the number of SPL programs to examine for that test without reducing its ability to find bugs. In a poster paper [19], we introduced the notion of analyzing features to reduce the effort of testing a product line. This paper expands the poster paper substantially by making the following new contributions:

Technique. We precisely define (ir)relevance in terms of changes that a feature can make to a program. We modify off-the-shelf static analyses for object-oriented programs to check for relevance.

Implementation. We implement our technique as an Eclipse plugin that uses *Soot*[30], a popular static analysis framework for Java, and *SAT4J*[31], an off-the-shelf SAT solver.

Evaluation. We demonstrate the effectiveness of our technique on concrete product lines and tests.

2. MOTIVATING EXAMPLE

Product Line. Suppose that we have the product line in Figure 1 that represents bank accounts where one can add money and be rewarded for being a valuable customer. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

March 2125, 2011, Pernambuco, Brazil.

Copyright 2011 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

Figure 6: Algorithm to Find Test Con gurations

Table 1: Con gurations to Test

Test1	Test2	Test3	Base	Loyalty	Fee	Ceiling
No	No	Yes	1	0	0	1
No	Yes	Yes	1	0	1	0
No	Yes	Yes	1	0	1	1
Yes	Yes	Yes	1	1	0	0
No	Yes	Yes	1	1	0	1
No	No	Yes	1	1	1	0
No	No	Yes	1	1	1	1

irrelevant features as don't-cares. More specifically, we find a solution to the specialized feature model and add it to the configurations to test (lines 6-7). We then ensure that the configuration's assignments to the relevant features do not appear again by creating a *blocking clause* [31] consisting of the assignments and conjoining the negation of the clause to the feature model (lines 9-16). We then check if there is another configuration and repeat the process until there are no more configurations.³

Once the configurations to test have been identified, a *test runner*, shown in Figure 3, goes through each configuration, creating a concrete program corresponding to the configuration from the SysGen program and running the test against that program.

Examples. Table 1 shows the results of analyzing our running example. Without analysis, each row, a configuration in the original feature model, would have to be executed for each test. However, with our analysis, given a test, only the rows with 1 entries in the column corresponding to the test need to be examined. For *Test1*, as stated in Section 5, there are no relevant features and thus the enumeration algorithm returns just one configuration, *Base*, to test. For *Test2*, four combinations of the relevant features *Base* and *Loyalty* must be tested. For *Test3*, all seven configurations must be tested.

7. CASE STUDIES

We implemented our technique as an Eclipse plugin and evaluated it on three product lines: *Graph Product Line*

³A simple variation of our algorithm terminates after collecting *k* configurations, in case there is a huge number of configurations to test.

(*GPL*), which is a set of programs that implement different graph algorithms [23]; *notepad*, a Java Swing application with functionalities similar to Windows Notepad; and *jak2java*, which is a feature-configurable tool that is part of the AHEAD Tool Suite [2].

Multiple tests were considered for each product line. Each test, essentially a unit test, creates and calls the product line's objects and methods corresponding to the functionality being tested. We ran our tool on a Windows XP machine with Intel Core2 Duo CPU with 2.4 GHz and 1024 MB as the maximum heap space. Note that although the product lines were created in-house, they were created long before this paper was conceived (*GPL* and *jak2java* were created over 5 years ago and *notepad* was created 2 years ago). In fact, these product lines were originally written in *Jak* [4] and for the purpose of this paper, we developed a *Jak-to-SysGen* translator to convert them into the SysGen representation. Our plugin, the examined product lines and tests, as well as the detailed evaluation results are available for download [18].

7.1 Graph Product Line (GPL)

Table 2 shows the results for *GPL*, which has 1713 LOC with 18 features and 156 configurations. Variations arise from algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted) that are used. We report two representative tests below.

CycleTest. 10 features are unbound. Applying the static analysis, we find that 7 out of the 10 are reachable. Out of these 7, only 1 feature, *nr*, is relevant. *nr* is relevant because it fails the data-flow check by adding an extra edge for every existing edge against the graph which was created by the *nr* feature. The other reachable features perform I/O operations on their own data are not considered to be relevant (see Section 8.1 for a discussion on I/O). With no analysis, the test would have to be run on 156 configurations, the number of programs in the product line. By specializing the feature model for this test and determining bound and unbound features, we reduce that number to 40. By applying the static analysis, we reduce the number to 2. The time taken to specialize the feature model is negligible. The static analysis takes less than a minute and a half.

Our technique achieves a useful reduction in the configurations to test. Such a reduction pays dividends in two ways. First, there is a good chance that it takes less time to perform the static analysis (1.20 minutes) and run the test on the reduced set (2) of configurations than to run the test on the original set (156) of configurations. But more importantly, redundant test results are eliminated and need not be analyzed by the tester. As far as the tester is concerned, there is no extra information in the other 154 test results and any information related to success or failure of the test can be obtained from these 2 configurations.

StronglyConnectedTest. This test requires a number of features to be bound for compilation, leaving only 4 features unbound. Out of those 4, 3 are reachable, but none are relevant. Just determining the unbound features already reduces the number of configurations, and applying the static analysis returns the best possible outcome, i.e. running the test on just 1 configuration. Like the previous test, the static analysis takes just over a minute.

Table 2: GPL Results

Lines of code	1713
Features	18
Configurations	156
CycleTest	
Unbound features	10
Reachable features	7
Relevant features	1: Undirected (data-flow)
Configurations with unbound features	40
Configurations to test	2
Duration of static analysis	72 sec. (1.20 min.)
StronglyConnectedTest	
Unbound features	4
Reachable features	3
Relevant features	0
Configurations with unbound features	16
Configurations to test	1
Duration of static analysis	72 sec. (1.20 min.)

Table 3: Notepad Results

Lines of code	2074
Features	25
Configurations	7057
PersistenceTest	
Unbound features	22
Reachable features	3
Relevant features	1: UndoRedo (data-flow)
Configurations with unbound features	5256
Configurations to test	2
Duration of static analysis	2856 sec. (47.60 min.)
PrintTest	
Unbound features	22
Reachable features	4
Relevant features	2: UndoRedo (data-flow) Persistence (introduction)
Configurations with unbound features	5256
Configurations to test	4
Duration of static analysis	2671 sec. (44.51 min.)

7.2 Notepad

Table 3 shows the results for `Notepad`, which has 2074 LOC with 25 features and 7056 configurations. Variations arise from the different permutations of functionalities, such as saving/opening files and printing, and user interface support for them (each functionality can have an associated toolbar button, menubar button, or both). We wrote tests for the example functionalities mentioned.

`PersistenceTest`. Binding still leaves 22 features unbound, but static analysis cuts down that number to 3 reachable features and only one relevant feature. The `undo` feature is relevant because it fails the data-flow check by attaching an event listener to the text area, which is allocated by another feature. Binding reduces 7057 configurations to 5256 and this is reduced to 2 configurations after running the analysis. Although Notepad is not large, it uses Java Swing, whose very large call-graph must be included in order for application call-back methods to be analyzed. This substantially raised the analysis time to 45 minutes. A common solution to this problem is to skip over certain method calls, especially those that are deep, in the framework, but this must be done with great care as doing so could prevent call-back methods from being reached. Reducing analysis time is a subject for further work.

`PrintTest`. The numbers are similar to the previous test, but this time, `print` is also found to be relevant because one of its methods overrides a method of an off-the-shelf file filter class in the Swing framework. Still, we only have to test 4 configurations rather than 5256. The duration is long for the same reason as mentioned previously.

7.3 jak2java

Table 4 shows the results for `jak2java`, which has 26,332 LOC with 17 features and 5 configurations. Despite the large code base and the number of features, there are only five configurations total because of the many constraints in the feature model. We wrote tests to execute the methods that we know are modified by other features. We aimed to find out whether these modifications would render these other features relevant to the method being executed. Here are some representative results.

`ReduceToJavaTest`. Features `add` and `remove` are relevant because they introduce methods that override meth-

ods of another feature. Feature `add` is relevant because it fails the control-flow check by returning early from the method of another feature. Unfortunately, all the configurations in the product line must be tested. The reason for this is that calling `add`, the method being tested, is very much like calling the `add` method of a product line, which reaches a large portion of the product line’s code base. Because there is a large amount of code to analyze, the static analysis takes 4.24 minutes. All the configurations have to be tested because a large fraction of the product line’s interactions are reachable.

`ArgInquireTest`. This test fares better because the method being called, `argInquire`, does not reach a large portion of the product line, taking 1.6 minutes to determine that only 1 configuration needs to be tested.

8. DISCUSSION

We now discuss assumptions and limitations, the effectiveness of our work, testing missing functionality, threats to validity, and a perspective.

8.1 Assumptions and Limitations

Off-the-shelf program analyses have well-known limitations. Indeed, the first three assumptions below are not unique to our work but the last assumption is.

Reflection. Any change to the code base, including the addition of a class member, can change the outcome of reflection. We assume that reflection is not used. Another possibility is to check if reflection is used in the control-flow of the test and consider *any* unbound feature to be relevant. Related work, such as [15], also do not consider reflection.

Native Calls. It is hard to determine if a native call, such as an I/O operation, has a side-effect using Soot. Rather than making the overly conservative assumption that every native call has a side-effect, we assume that native calls have no side-effect. Consequently, features can perform reads/writes to files or standard input/output without being considered relevant.

Timing. If a test uses the duration of its execution as an outcome, any feature that adds instructions to the test will be considered relevant. Rather than checking if a test indeed uses such a timer, we assume that it does not.

Local Variables. A method can declare variables local to it. We assume a feature’s modification does not reference or modify local variables introduced by other features. Features are written in a dedicated language like *Jak* [4] that restricts a feature’s modifications in this way. We assume this restriction holds and we use an off-the-shelf side-effect analysis, which, by definition of “side-effect” of a method, need not consider writes to local variables. Our benchmarks satisfy this assumption as they were translated from n to n representation using a translator. It would not require much effort to remove this limitation.

8.2 Effectiveness

Our technique works because there are tests that exercise a small portion of the product line involving a few features. Even just binding features can cut down many configurations. Further reductions are possible as not many features are reachable from a test and even fewer are relevant. Determining reachable and relevant features cannot be done manually and requires a dedicated program analysis like ours. Although a highly precise program analysis can significantly reduce false positives, our case studies illustrate that a context-insensitive analysis suffices because only a small set of classes and methods, relevant to the functionality being tested, are instantiated and invoked.

8.3 Testing Missing Functionality

Suppose feature n should modify the data-flow of a method n . The feature’s author forgets to make the modification, which causes our analysis to report that n is irrelevant when testing n . n is irrelevant because it is missing functionality, rather than having an orthogonal functionality as previous example features did. Without a specification, e.g. that n is supposed to be relevant to n ,

burden of detecting missing functionality rests on the alertness of testers; no program analysis could detect this error. This is a general problem of testing and is not limited to our work. In fact, our work helps in that it reports information on feature (ir)relevance, which may provide a clue to such errors.

8.4 Threats to Validity

Our technique can take longer than running the test on all the configurations, as is the case with R for u since there is no reduction in configurations. But we believe this case is an outlier and the static analysis is worth running to achieve even a small reduction for several reasons. First, testing a product requires it to be synthesized, which takes non-negligible time. Second, it takes time to run the tests themselves. Third, a configuration’s test result may be redundant with another configuration’s test result due to an irrelevant feature between the two, yet the tester will have to waste time analyzing both configurations’ results. Further experience with our analysis will bear out these points.

8.5 Perspective

Initially, our belief was that existing analyses for conventional programs could be directly applied to a SysGen program. However, we discovered that analyzing a SysGen program was much more challenging than we had anticipated. Consider the following example. While some parts of our analysis, including reachability and data-flow check, are performed using a backend abstraction like 3-address code, other parts of our analysis, notably the control-flow check, must be performed on a frontend abstraction like Abstract Syntax Tree because branch statements like r and n are often optimized away on the backend [6]. This presents the technical challenge of developing a bridge between frontend and backend analyses. For example, we only want to perform control-flow checks on the reachable methods, but these methods are determined by a backend analysis. Currently, we provide a string representation that the frontend and the backend abstractions both map to. Developing a more robust intermediate abstraction may be necessary in the future.

9. RELATED WORK

9.1 Product Line Testing and Verification

There is a considerable amount of research in product line testing and verification (see [24] for a survey). We discuss research most closely related to ours.

Model-Checking. Classen et al.[8] 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, but for transition systems. Their technique composes the product line’s FTS with the automaton of the temporal property’s negation and reports violating configurations. Although we both tackle the general problem of checking a property against a product line, they work on a representation (transition systems) and setting (verifying temporal properties) different from ours (object-oriented programs and testing), making the two techniques complementary.

Sampling. Sampling exploits domain knowledge, rather than program analysis results, to select configurations to

test. An SPL tester may choose a set of features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested [10][11][25]. Sampling approaches can miss problematic configurations, whereas we use a program analysis to safely prune feature combinations.

Test Construction. Instead of generating tests from a complete specification of a program, tests are generated incrementally from feature specifications [34]. There is also research on constructing a product line of tests so that they may be reused [5][27]. We address the different problem of minimizing test execution for a single given test.

9.2 Program Slicing

Determining relevant features is closely related to *backwards program slicing* [36], which uses a dataflow analysis to determine the minimal subset of a program that can affect the values of specified variables at a specified program point. Our definition of relevance is more conservative than a “slice” [36] but at the same time, requires less precision: our goal is to reduce feature combinations to test by determining features, not statements, for which we need only assign one truth value.

9.3 Feature Interactions

There is a large body of work on detecting feature interactions using static analysis [28][32][13][9][22], of which *harmless advice* [13] and *Modular Aspects with Ownership (MAO)* [9] are the most relevant. Harmless advice introduces a type system in which aspects can terminate control-flow but cannot produce data-flow to the base program. MAO relies on contracts to determine if an aspect changes the control-flow or data-flow of another module. Our analysis was inspired by MAO, but is technically closer to harmless advice, as both perform an inter-procedural analysis and do not rely on contracts. But unlike harmless advice, our approach does not require every feature to be harmless or irrelevant.

More importantly, MAO and harmless advice assume a setting where all modules (i.e. aspects/features) are required for the program to work, which is sharply different from SPLs. Indeed, related work in feature interactions perform analysis more for modular reasoning of a single program, rather than for reducing combinatorics in product line testing.

9.4 Compositional Analysis and Verification

Currently, we perform a static analysis for each test. With multiple tests, it is possible that the same classes and methods of the product line will be analyzed multiple times. It may be possible to analyze the product line once and combine the result against that of analyzing each test using compositional static analysis [12] and verification [14][22].

9.5 Reducing Testing Effort

Reducing testing effort for a single program, typically using output from some analysis, has a long history. [29] identifies a subset of existing tests to run given a program change. We address a complementary problem, namely to identify a subset of existing features that are relevant for a given test in a product line.

10. 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 test in an SPL.

Features are a fundamental, but unconventional, form of modularity. Combinations of features yield different programs in an SPL and each program is identified by a unique combination of features. Features impose a considerable amount of structure on programs (that is why features are composable in combinatorial numbers of ways), and exploiting this structure has been the focus of our paper.

Our key insight is that every SPL test is designed to evaluate one or more properties of a program. A feature might alter any number of properties. In SPL testing, a particular feature may be relevant to a property (test) or it may not. Determining whether a feature is relevant for a given test is the critical problem.

We presented a framework for testing an SPL. 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 configurations. And of the reachable features, we determine the features that affect the properties being evaluated, reducing configurations even more.

Several case studies were presented that showed meaningful reductions in the number of configurations to test, and more importantly, lends credence to the folk-tale that many features of a product line add new behavior without affecting existing behavior. We demonstrated the idea of leveraging such features to exhaustively but efficiently test product lines. Our work is a step forward in practical reductions in SPL testing.

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.

11. REFERENCES

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [2] D. Batory. Ahead tool suite.
- [3] D. Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin, Texas, Mar. 2005.
- [4] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *In Proceedings Fifth International Conference on Software Reuse*, pages 143–153. IEEE.
- [5] A. Bertolino and S. Gnesi. Pluto: A test methodology for product families. In F. van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2003.
- [6] E. Bodden. Private and Soot newsgroup correspondence, 2010.
- [7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In

- ISSTA '02*, July 2002.
- [8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines (to appear). In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*. IEEE, 2010. Acceptance rate: 13.7
- [9] C. Clifton, G. T. Leavens, and J. Noble. MAO: Ownership and effects for more effective reasoning about aspects. In *ECOOP'07*.
- [10] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*. ACM, 2006.
- [11] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM.
- [12] P. Cousot and R. Cousot. Modular static program analysis. In *Proceedings of Compiler Construction*, pages 159–178. Springer-Verlag, 2002.
- [13] D. S. Dantas and D. Walker. Harmless advice. *SIGPLAN Not.*, 41(1):383–396, 2006.
- [14] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02*.
- [15] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA'01*.
- [16] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
- [17] M. Janota. Do sat solvers make good configurators? In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 191–195. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
- [18] C. H. P. Kim. Reducing combinatorics in product line testing: Tool and results. Available from www.usenix.org/conference/ussw/2010, 2010.
- [19] C. H. P. Kim, D. Batory, and S. Khurshid. Eliminating Products to Test in a Software Product Line. In *ASE2010 (Tentatively Accepted Poster Session Paper)*. Available from www.usenix.org/conference/ussw/2010.
- [20] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in jit optimizations. In *Compiler Construction*, volume 3443 of *LNCS*, 2005.
- [21] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
- [22] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *SIGSOFT Softw. Eng. Notes*, 27(6):89–98, 2002.
- [23] R. E. Lopez-herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. 2001 Conf. Generative and Component-Based Software Eng*, pages 10–24. Springer, 2001.
- [24] R. Lutz. Survey of product-line verification and validation techniques. Technical report, Jet Propulsion Laboratory, NASA, May 2007.
- [25] J. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, CMU/SEI, Mar. 2001. Available from www.sei.cmu.edu.
- [26] C. Nebut, Y. L. Traon, and J.-M. Jézéquel. System testing of product lines: From requirements to test cases. In *Software Product Lines*, pages 447–478. Springer-Verlag, 2006.
- [27] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.
- [28] C. Prehofer. Semantic reasoning about feature composition via multiple aspect-weavings. In *GPCE'06*.
- [29] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996.
- [30] Sable Group. Soot: a Java optimization framework.
- [31] SAT4J. SAT4J. www.sat4j.org.
- [32] G. Snelling and F. Tip. Semantics-based composition of class hierarchies. In *ECOOP'02*.
- [33] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.
- [34] E. Uzuncaova, D. Garcia, S. Khurshid, and D. S. Batory. Testing software product lines using incremental test generation. In *ISSRE'08*.
- [35] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.
- [36] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [37] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE'04*.