A Technique for Agile and Automatic Interaction Testing for Product Lines

Martin Fagereng Johansen[1,2], Øystein Haugen[1], Franck Fleurey[1], Erik Carlson[3], Jan Endresen[3], and Tormod Wien[3]

[1] SINTEF ICT, Pb. 124 Blindern, 0314 Oslo, Norway
{Martin.Fagereng.Johansen,Oystein.Haugen,Franck.Fleurey}@sintef.no
[2] Institute for Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
[3] ABB, Bergerveien 12, 1375 Billingstad, Norway
{erik.carlson,jan.endresen,tormod.wien}@no.abb.com

Abstract. Product line developers must ensure that existing and new features work in all products. Adding to or changing a product line might break some of its features. In this paper, we present a technique for automatic and agile interaction testing for product lines. The technique enables developers to know if features work together with other features in a product line, and it blends well into a process of continuous integration. The technique is evaluated with two industrial applications, testing a product line of safety devices and the Eclipse IDEs. The first case shows how existing test suites are applied to the products of a 2-wise covering array to identify two interaction faults. The second case shows how over 400,000 test executions are performed on the products of a 2-wise covering array using over 40,000 existing automatic tests to identify potential interactions faults.

Keywords: Product Lines, Testing, Agile, Continuous Integration, Automatic, Combinatorial Interaction Testing.

1 Introduction

A product line is a collection of products with a considerable amount of hardware or code in common. The commonality and differences between the products are usually modeled as a feature model. A product of a product line is given by a configuration of the feature model, constructed by specifying whether features are including or not. Testing product lines is a challenge since the number of possible products grows exponentially with the number of choices in the feature model. Yet, it is desirable to ensure that the valid products function correctly.

One approach for testing product lines is combinatorial interaction testing [1]. Combinatorial interaction testing is to first construct a small set of products, called a covering array, in which interaction faults are most likely to show up and then to test these products normally. We have previously advanced this approach by showing that generating covering arrays from realistic features models is tractable [2] and by providing an algorithm that allows generating covering arrays for product lines of the size and complexity found in industry [3].

In its current form, the application of combinatorial interaction testing to testing product lines is neither fully automatic nor agile; a technique for automatic and agile testing of product lines based on combinatorial interaction testing is the contribution of this paper, presented in Section 3. The technique is evaluated by applying it to test two industrial product lines, a product line of safety devices and the Eclipse IDEs; this is presented in Section 4.

In Section 4.1 it is shown how the technique can be implemented using the Common Variability Language (CVL) [4] tool suite. (CVL is the language of the ongoing standardization effort of variability languages by OMG.) Five test suites were executed on 11 strategically selected products, the pair-wise covering array, of a product line of safety devices to uncover two unknown and previously undetected bugs.

In Section 4.3 it is shown how the technique can be implemented using the Eclipse Platform plug-in system. More than 40,000 existing automatic tests were executed on 13 strategically selected products, the pair-wise covering array, of the Eclipse IDE product line, producing more than 400,000 test results that reveal a multitude of potential interaction faults.

2 Background and Related Work

2.1 Product Lines

A product line [5] is a collection of products with a considerable amount of hardware or code in common. The primary motivation for structuring one's products as a product line is to allow customers to have a system tailored for their purpose and needs, while still avoiding redundancy of code. It is common for customers to have conflicting requirements. In that case, it is not even possible to ship one product for all customers.

The Eclipse IDE products [6] can be seen as a software product line. Today, Eclipse lists 12 products (which configurations are shown in Table 1a[1]) on their download page[2].

One way to model the commonalities and differences in a product line is using a feature model [7]. A feature model sets up the commonalities and differences of a product line in a tree such that configuring the product line proceeds from the root of the tree. Figure 1 shows the part of the feature model for the Eclipse IDEs that is sufficient to configure all official versions of the Eclipse IDE. The figure uses the common notation for feature models; for a detailed explanation of feature models, see Czarnecki and Eisenecker 2000 [8].

2.2 Product Line Testing

Testing a product line poses a number of new challenges compared to testing single systems. It has to be ensured that each possible configuration of the product

[1] `http://www.eclipse.org/downloads/compare.php`, retrieved 2012-04-12.
[2] `http://eclipse.org/downloads/`, retrieved 2012-04-12.

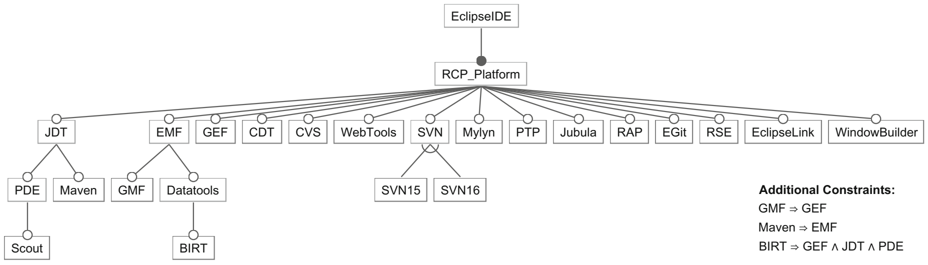


Fig. 1. Feature Model for the Eclipse IDE Product Line

line functions correctly. One way to validate a product line is through testing, but testing is done on a running system. The software product line is simply a collection of many products. One cannot test each possible product, since the number of products in general grows exponentially with the number of features in the product line. For the feature model in Figure 1, there are $356,352$ possible configurations.

Reusable Component Testing. In a survey of empirics of what is done in industry for testing software product lines [9], we found that the technique with considerable empirics showing benefits is *reusable component testing*. Given a product line where each product is built by bundling a number of features implemented in components, reusable component testing is to test each component in isolation. The empirics have later been strengthened; Ganesan et al. 2012 [10] is a report on the test practices at NASA for testing their Core Flight Software System (CFS) product line. They report that the chief testing done on this system is reusable component testing [10].

Interaction Testing. There is no single recommended approach available today for testing interactions between features in product lines efficiently [11], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [1], discussed below; a technique called ScenTED, where the idea is to express the commonalities and differences on the UML model of the product line and then derive concrete test cases by analyzing it [12]; and incremental testing, where the idea is to automatically adapt a test case from one product to the next using the specification of similarities and differences between the products [13]. Kim et al. 2011 [14] presented a technique where they can identify irrelevant features for a test case using static analysis.

Combinatorial Interaction Testing: Combinatorial interaction testing [1] is one of the most promising approaches. The benefits of this approach is that it deals directly with the feature model to derive a small set of products (a covering array) which products can then be tested using single system testing techniques, of which there are many good ones [15].

Table 1. Eclipse IDE Products, Instances of the Feature Model in Figure 1

(a) Official Eclipse IDE products

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
RCP_Platform	X	X	X	X	X	X	X	X	X	X	X	X
CVS	X	X	X	X	X	X	X	X	X	X	X	X
EGit	-	-	X	X	X	X	-	-	-	-	-	-
EMF	X	X	-	-	-	X	X	-	-	-	-	-
GEF	X	X	-	-	-	X	X	-	-	-	-	-
JDT	X	X	-	-	X	X	X	-	X	-	-	X
Mylyn	X	X	X	X	X	X	X	X	X	X	X	-
WebTools	-	X	-	-	-	-	X	-	-	-	X	-
RSE	-	X	X	X	-	-	X	X	-	-	-	-
EclipseLink	-	X	-	-	-	-	X	-	-	X	-	-
PDE	-	X	-	-	X	X	X	-	X	-	-	X
Datatools	-	X	-	-	-	-	X	-	-	-	-	-
CDT	-	-	X	X	-	-	-	X	-	-	-	-
BIRT	-	-	-	-	-	-	X	-	-	-	-	-
GMF	-	-	-	-	-	X	-	-	-	-	-	-
PTP	-	-	-	-	-	-	-	X	-	-	-	-
Scout	-	-	-	-	-	-	-	-	X	-	-	-
Jubula	-	-	-	-	-	-	-	-	-	X	-	-
RAP	-	-	-	-	X	-	-	-	-	-	-	-
WindowBuilder	X	-	-	-	-	-	-	-	-	-	-	-
Maven	X	-	-	-	-	-	-	-	-	-	-	-
SVN	-	-	-	-	-	-	-	-	-	-	-	-
SVN15	-	-	-	-	-	-	-	-	-	-	-	-
SVN16	-	-	-	-	-	-	-	-	-	-	-	-

(b) Pair-wise Covering Array

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12	13
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X	X
RCP_Platform	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS	-	X	-	X	-	X	-	-	X	-	-	-	-
EGit	-	X	-	-	X	X	X	-	-	X	-	-	-
EMF	-	X	X	X	X	-	-	X	X	X	X	X	-
GEF	-	-	X	X	X	-	X	X	X	-	-	X	-
JDT	-	X	X	X	X	-	X	-	X	X	-	X	-
Mylyn	-	X	-	X	-	-	X	X	-	-	-	-	-
WebTools	-	-	X	X	X	-	X	-	-	X	X	-	-
RSE	-	X	X	-	X	X	-	-	-	-	-	-	-
EclipseLink	-	X	X	-	-	X	-	X	X	-	-	-	-
PDE	-	X	-	X	X	-	X	-	X	-	-	X	-
Datatools	-	X	X	X	X	-	-	-	X	-	X	X	-
CDT	-	-	X	X	-	X	-	X	X	-	-	-	-
BIRT	-	-	-	X	X	-	-	-	X	-	-	X	-
GMF	-	-	X	X	X	-	-	X	X	-	-	-	-
PTP	-	-	X	-	X	X	-	X	X	-	-	-	-
Scout	-	X	-	X	-	-	X	-	X	-	-	-	-
Jubula	-	-	X	X	-	X	-	X	-	X	-	-	-
RAP	-	X	X	-	-	X	X	-	X	-	-	-	-
WindowBuilder	-	X	-	X	-	X	-	X	-	-	-	-	-
Maven	-	X	X	-	-	-	-	-	X	X	-	-	-
SVN	-	X	-	-	X	X	X	X	X	-	X	-	X
SVN15	-	X	-	-	X	-	-	X	-	-	-	-	X
SVN16	-	-	-	-	-	X	X	-	X	-	X	-	-

There are three main stages in the application of combinatorial interaction testing to a product line. First, the feature model of the system must be made. Second, the t-wise covering array must be generated. We have developed an algorithm that can generate such arrays from large features models [3][3]. These products must then be generated or physically built. Last, a single system testing technique must be selected and applied to each product in this covering array.

Table 1b shows the 13 products that must be tested to ensure that every pair-wise interaction between the features in the running example functions correctly. Each row represents one feature and every column one product. 'X' means that the feature is included for the product, '-' means that the feature is not included. Some features are included for every product because they are core features, and some pairs are not covered since they are invalid according to the feature model.

Testing the products in a pair-wise covering array is called 2-wise testing, or pair-wise testing. This is a special case of t-wise testing where $t = 2$. t-wise testing is to test the products in a covering array of strength t. 1-wise coverage means that every feature is at least included and excluded in one product, 2-wise coverage means that every combination of two feature assignments are in the covering array, etc. For our running example, 3, 13 and 40 products is sufficient to achieve 1-wise, 2-wise and 3-wise coverage, respectively.

[3] See [3] for a definition of covering arrays and for an algorithm for generating them.

Empirical Motivation. An important motivation for combinatorial interaction testing is a paper by Kuhn et al. 2004 [16]. They indicated empirically that most bugs are found for 6-wise coverage, and that for 1-wise one is likely to find on average around 50%, for 2-wise on average around 70%, and for 3-wise around 95%, etc.

Garvin and Cohen 2011 [17] did an exploratory study on two open source product lines. They extracted 28 faults that could be analyzed and which was configuration dependent. They found that three of these were true interaction faults which require at least two specific features to be present in a product for the fault to occur. Even though this number is low, they did experience that interaction testing also improves feature-level testing, that testing for interaction faults exercised the features better. These observations strengthen the case for combinatorial interaction testing.

Steffens et al. 2012 [18] did an experiment at Danfoss Power Electronics. They tested the Danfoss Automation Drive which has a total of 432 possible configurations. They generated a 2-wise covering array of 57 products and compared the testing of it to the testing all 432 products. This is possible because of the relatively small size of the product line. They mutated each feature with a number a mutations and ran test suites for all products and the 2-wise covering array. They found that 97.48% of the mutated faults are found with 2-wise coverage.

3 Proposed Technique

We address two problems with combinatorial interaction testing of software product lines in our proposed technique. A generic algorithm for automatically performing the technique is presented in Section 3.2, an evaluation of it is presented in Section 4 and a discussion of benefits and limitations presented in Section 5.

- **The functioning of created test artifacts is sensitive to changes in the feature model:** The configurations in a covering array can be drastically different with the smallest change to the feature model. Thus, each product must be built anew and the single system test suites changed manually. Thus, plain combinatorial interaction testing of software product lines is not agile. This limits it from effectively being used during development.
- **Which tests should be executed on the generated products:** In ordinary combinatorial interaction testing, a new test suite must be made for a unique product. It does not specify how to generate a complete test suite for a product.

3.1 Idea

Say we have a product line in which two features, A and B, are both optional and mutually optional. This means that there are four situations possible: Both A and B are in the product, only A or only B is in the product and neither is in the product. These four possibilities are shown in Table 2a.

Table 2. Feature Assignment Combinations

(a) Pairs

Feature\Situation	1	2	3	4
A	X	X	-	-
B	X	-	X	-

(b) Triples

Feature\Situation	1	2	3	4	5	6	7	8
A	X	X	X	X	-	-	-	-
B	X	X	-	-	X	X	-	-
C	X	-	X	-	X	-	X	-

If we have a test suite that tests feature A, $TestA$, and another test suite that tests feature B, $TestB$, the following is what we expect: (1) When both feature A and B are present, we expect $TestA$ and $TestB$ to succeed. (2) When just feature A is present, we expect $TestA$ to succeed. (3) Similarly, when just feature B is present, we expect $TestB$ to succeed. (4) Finally, when neither feature is present, we expect the product to continue to function correctly. In all four cases we expect the product to build and start successfully.

Similar reasoning can be made for 3-wise and higher testing, which cases are shown in Table 2b. For example, for situation 1, we expect $TestA$, $TestB$ and $TestC$ to pass, in situation 2, we expect $TestA$ and $TestB$ to pass, which means that A and B work in each other's presence and that both work without C. This kind of reasoning applies to the rest of the situations in Table 2b and to higher orders of combinations.

3.2 Algorithm for Implementation

The theory from Section 3.1 combined with existing knowledge about combinatorial interaction testing can be utilized to construct a testing technique. Algorithm 1 shows the pseudo-code for the technique.

The general idea is, for each product in a t-wise covering array, to execute the test suites related to the included features. If a test suite fails for one configuration, but succeeds for another, we can know that there must be some kind of interaction disturbing the functionality of the feature.

In Algorithm 1, line 1, the covering array of strength t of the feature model FM is generated and the set of configurations are placed in CA_t. At line 2, the algorithm iterates through each configuration. At line 3, a product is constructed from the configuration c. PG is an object that knows how to construct a product from a configuration; making this object is a one-time effort. The product is placed in p. If the construction of the product failed, the result is placed in the result table, *ResultTable*. The *put* operation on *ResultTable* takes three parameters, the result, the column and the row. The row parameter can be an asterisk, '*', indicating that the result applies to all rows.

If the build succeeded, the algorithm continues at line 7 where the algorithm iterates through each test suite, *test*, of the product line, provided in a set $Tests$. At line 8, the algorithm takes out the feature, f, that is tested by the test suite *test*. The algorithm finds that in the object containing the Test-Feature-Mapping, TFM. At line 9, if this feature f is found to be included in the current

Algorithm 1. Pseudo Code of the Automatic and Agile Testing Algorithm

```
1: $CA_t \leftarrow GenerateCoveringArray(FM, t)$
2: **for** each configuration $c$ in $CA_t$ **do**
3:    $p \leftarrow PG.GenerateProduct(c)$
4:    **if** $p$'s build failed **then**
5:       $ResultTable.put(''buildfailed'', c, *)$
6:    **else**
7:       **for** each test $test$ in $Tests$ **do**
8:          $f \leftarrow TFM.getFeatures(test)$
9:          **if** $c$ has features $f$ **then**
10:            $result \leftarrow p.runTest(test)$
11:            $ResultTable.put(result, c, f)$
12:         **end if**
13:      **end for**
14:   **end if**
15: **end for**
```

configuration, c, then, at line 10, the test suite is run. The results from running the test is placed in the result table[4], line 11.

3.3 Result Analysis

Results stored in a result table constructed by Algorithm 1 allow us to do various kinds of analysis to identify the possible causes of the problems.

Attributing the Cause of a Fault. These examples show how analysis of the result can proceed:

- If we have a covering array of strength 1, CA_1, of a feature model FM: If a build fails whenever f_1 is not included, we know that f_1 is a core feature.
- If we have a covering array of strength 2, CA_2, of a feature model FM in which feature f_1 and f_2 are independent on each other: If, $\forall c \in CA_2$ where both f_1 and f_2 are included, the test suite for f_1 fails, while where f_1 is included and f_2 is not, then the test suite of f_1 succeeds, we know that the cause of the problem is a disruption of f_1 caused by the inclusion f_2.
- If we have a covering array of strength 2, CA_2, of a feature model FM in which feature f_1 and f_2 are not dependent on each other: If, $\forall c \in CA_2$ where both f_1 and f_2 are included, the test suites for both f_1 and f_2 succeed, while where f_1 is included and f_2 is not, then the test suite of f_1 fails, we know that the cause of the problem is a hidden dependency from f_1 to f_2.

These kinds of analysis are possible for all the combinations of successes and failures of the features for the various kinds of interaction-coverages.

[4] Two examples of result tables are shown later, Tables 4b and 5b.

Of course, if there are many problems with the product line, then several problems might overshadow each other. In that case, the tester must look carefully at the error given by the test case to find out what the problem is. For example, if every build with f_1 included fails that will overshadow a second problem that f_2 is dependent on f_1.

Guarantees. It is uncommon for a testing technique to have guarantees, but there are certain errors in the feature model that will be detected.

- Feature f is not specified to be a core feature in the feature model but is in the implementation. This is guaranteed to be identified using a 1-wise covering array: There will be a product in the covering array with f not included that will not successfully build, start or run.
- Feature f_1 is not dependent on feature f_2 in the feature model, but there is a dependency in the code. This is guaranteed to be identified using a 2-wise covering array. There will be a product in the 2-wise covering array with f_1 included and f_2 not included that will not pass the test suite for feature f_1.

4 Evaluation with Two Applications and Results

4.1 Application to ABB's "Safety Module"

About the ABB Safety Module. The ABB Safety Module is a physical component that is used in, among other things, cranes and assembly lines, to ensure safe reaction to events that should not occur, such as the motor running too fast, or that a requested stop is not handled as required. It includes various software configurations to adapt it to its particular use and safety requirements.

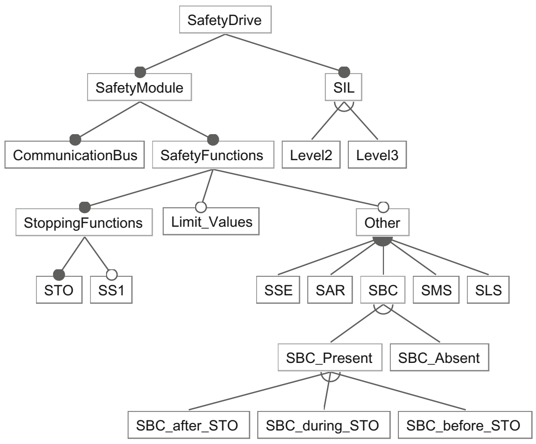
A simulated version of the ABB Safety Module was built—independently of the work in this paper—for experimenting with testing techniques. It is this version of the ABB Safety Module which testing is reported in this paper.

Basic Testing of Sample Products. Figure 2a shows the feature model of the Safety Module. There are in total 640 possible configurations. Three of these are set up in the lab for testing purposes during development. These are shown in Figure 2b and are, of course, valid configurations of the feature model of the ABB Safety Module, Figure 2a.

The products are tested thoroughly before they are delivered to a customer. Five test suites are named in the left part of Table 3a; the right side names the feature that the test suite tests.

We ran the relevant tests from Table 3a. The results from running the relevant test suite of each relevant product are shown in Table 3b. The table shows a test suite in each row and a product in each column. When the test suite tests a feature not present in the product, the entry is blank. When the test suite tests a feature in the product, the error count is shown. All test runs gave zero errors, meaning that they were successful for the three sample products. This is also what we expected since these three test products have been used during development of the simulation model to see that it functions correctly.

(a) Feature Model



(b) Sample Products

Feature\Product	a1	a2	a3
SafetyDrive	X	X	X
SafetyModule	X	X	X
CommunicationBus	X	X	X
SafetyFunctions	X	X	X
StoppingFunctions	X	X	X
STO	X	X	X
SS1	X	X	X
Limit_Values	-	-	-
Other	X	X	X
SSE	X	X	X
SAR	X	X	-
SLS	X	X	X
SBC	X	X	X
SBC_Present	X	-	X
SBC_during_STO	-	-	-
SBC_after_STO	-	-	X
SBC_before_STO	X	-	-
SBC_Absent	-	X	-
SMS	X	X	X
SIL	X	X	X
Level2	-	X	-
Level3	X	-	X

Fig. 2. ABB Safety Module Product Line

Table 3

(a) Feature-Test Mapping

Unit-Test Suite	Feature
GeneralStartUp	SafetyDrive
Level3StartUpTest	Level3
TestSBC_After	SBC_after_STO
TestSBC_Before	SBC_before_STO
TestSMS	SMS

(b) Test errors

Test\Product	a1	a2	a3
GeneralStartUp	0	0	0
Level3StartUpTest	0		0
TestSBC_After			0
TestSBC_Before	0		
TestSMS	0	0	0

Testing Interactions Systematically. The three sample products are three out of 640 possible products. Table 4a shows the 11 products that need to be tested to ensure that every pair of features is tested for interaction faults; that is, the 2-wise covering array of Figure 2a.

We built these products automatically and ran the relevant automatic test suite on them. Table 4b shows the result from running each relevant test suite on each product of Table 4a. If the features interact correctly, we expect that there would be no error.

As we can see, products 2, 3, 7 and 8 did not compile correctly. This proved to be because for certain configurations, the CVL variability model was built incorrectly, producing a faulty code that does not compile.

For product 9, the test suite for the SMS ("Safe Maximum Speed") feature failed. This is interesting, because it succeeded for product 4 and 5. We investigated the problem, and found that the SMS feature does not work if the break is removed from the ABB Safety Module. This is another example of an interaction

Table 4. Test Products and Results for Testing the Safety Module

(a) 2-wise Covering Array

Feature\Product	0	1	2	3	4	5	6	7	8	9	10
SafetyDrive	X	X	X	X	X	X	X	X	X	X	X
SafetyModule	X	X	X	X	X	X	X	X	X	X	X
CommunicationBus	X	X	X	X	X	X	X	X	X	X	X
SafetyFunctions	X	X	X	X	X	X	X	X	X	X	X
StoppingFunctions	X	X	X	X	X	X	X	X	X	X	X
STO	X	X	X	X	X	X	X	X	X	X	X
SS1	-	-	X	X	-	X	-	X	X	X	-
Limit_Values	-	X	-	X	-	X	X	X	-	-	X
Other	-	X	X	-	X	X	X	X	X	X	X
SSE	-	-	X	-	-	X	X	X	X	-	-
SAR	-	X	-	-	X	X	X	-	-	-	X
SBC	-	X	X	-	X	-	X	X	X	X	X
SBC_Present	-	X	X	-	X	-	-	X	X	-	X
SBC_after_STO	-	-	-	-	X	-	-	X	-	-	-
SBC_during_STO	-	X	-	-	-	-	-	-	X	-	-
SBC_before_STO	-	-	X	-	-	-	-	-	-	-	X
SBC_Absent	-	-	-	-	-	-	X	-	-	X	-
SMS	-	-	X	-	X	X	-	-	X	X	-
SLS	-	X	X	-	-	X	-	X	-	X	-
SIL	X	X	X	X	X	X	X	X	X	X	X
Level2	-	-	X	X	-	-	X	X	X	-	-
Level3	X	X	-	-	X	X	-	-	-	X	X

(b) Test Errors

Test\Product	0	1	2	3	4	5	6	7	8	9	10
GeneralStartUp	0	0	-	-	0	0	0	-	-	0	0
Level3StartUpTest	0	0			0	0				0	0
TestSBC_After					0			-			
TestSBC_Before			-								0
TestSMS			-		0	0			-	1	

fault. It occurs when SMS is present, and the break is absent. The inclusion of *SBC_Absent* means that there is no break within in the implementation.

4.2 Implementation with CVL

The pseudo-algorithm for implementing the technique with the CVL [4] tool suite is shown as Algorithm 2. It is this implementation that was used to test the ABB Safety Module[5]. The algorithm assumes that the following is given to it: a CVL variability model object, VM; a coverage strength, t; a list of tests, *tests*; and that a mapping between the tests and the features, TFM.[6]

The algorithm proceeds by first generating a t-wise covering array and setting them up as resolution models in the CVL model, VM. The CVL model contains bindings to the executable model artifacts for the ABB Safety Module. Everything that is needed is reachable from the CVL model. It can thus be used to generate the executable product simulation models; the set of product models is placed in P. The algorithm then loops through each product p. For each product, it sees if the build succeeded. If it did not, that is noted in *resultTable*. If the build succeeded, the algorithm runs through each test from the test set provided. If the feature the test tests is present in the product, run the test and record the result in the proper entry in *resultTable*. The result table we got in the experiment with the ABB Safety Module is shown in Table 4b.

[5] The source code for this implementation including its dependencies is found on the paper's resource website: `http://heim.ifi.uio.no/martifag/ictss2012/`

[6] All these are available on the paper's resource website.

Algorithm 2. Pseudo Code of CVL-based version of Algorithm 1

```
1:  $VM.GenerateCoveringArray(t)$
2:  $P \leftarrow VM.GenerateProducts()$
3:  for each product $p$ in $P$ do
4:      if $p$ build failed then
5:          $resultTable.put("buildfailed", p, *)$
6:      else
7:          for each test $test$ in $tests$ do
8:              $f \leftarrow TFM.getFeatures(test)$
9:              if $p$ has features $f$ then
10:                 $result \leftarrow p.runTest(test)$
11:                 $resultTable.put(result, p, f)$
12:             end if
13:         end for
14:     end if
15: end for
```

4.3 Application to the Eclipse IDEs

The Eclipse IDE product line was introduced earlier in this paper: The feature model is shown in Figure 1, and a 2-wise covering array was shown in Table 1b.

The different features of the Eclipse IDE are developed by different teams, and each team has test suites for their feature. Thus, the mapping between the features and the test suites are easily available.

The Eclipse Platform comes with built-in facilities for installing new features. We can start from a new copy of the bare Eclipse Platform, which is an Eclipse IDE with just the basic features. When all features of a product have been installed, we can run the test suite associated with each feature.

We implemented Algorithm 1 for the Eclipse Platform plug-in system and created a feature mapping for 36 test suites. The result of this execution is shown in Table 5b. This experiment[7] took in total 10.8 GiB of disk space; it consisted of 40,744 tests and resulted in 417,293 test results that took over 23 hours to produce on our test machine.

In Table 5b, the first column contains the results from running the 36 test suites on the released version of the Eclipse IDE for Java EE developers. As expected, all tests pass, as would be expected since the Eclipse project did test this version with these tests before releasing it.

The next 13 columns show the result from running the tests of the products of the complete 2-wise covering array of the Eclipse IDE product line. The blank cells are cells where the feature was not included in the product. The cells with a '-' show that the feature was included, but there were no tests in the test setup for this feature. The cells with numbers show the number of errors produced by running the tests available for that feature.

[7] The experiment was performed on Eclipse Indigo 3.7.0. The computer on which we did the measurements had an Intel Q9300 CPU @2.53GHz, 8 GiB, 400MHz RAM and the disk ran at 7200 RPM.

Table 5. Tests and Results for Testing the Eclipse IDE Product Line, Figure 1, Using the 2-wise Covering Array of Table 1b

(a) Tests

Test Suite	Tests	Time(s)
EclipseIDE	0	0
RCP_Platform	6,132	1,466
CVS	19	747
EGit	0	0
EMF	0	0
GEF	0	0
JDT	33,135	6,568
Mylyn	0	0
WebTools	0	0
RSE	0	0
EclipseLink	0	0
PDE	1,458	5,948
Datatools	0	0
CDT	0	0
BIRT	0	0
GMF	0	0
PTP	0	0
Scout	0	0
Jubula	0	0
RAP	0	0
WindowBuilder	0	0
Maven	0	0
SVN	0	0
SVN15	0	0
SVN16	0	0
Total	40,744	14,729

(b) Results, Number of Errors

Feature\Prod.	JavaEE	1	2	3	4	5	6	7	8	9	10	11	12	13
EclipseIDE	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RCP_Platform	0	17	90	94	0	0	90	0	91	87	7	0	0	10
CVS	0		0		0		0			0				
EGit			-			-	-	-			-			
EMF	-		-	-	-	-			-	-	-	-	-	
GEF	-			-	-	-		-	-	-			-	
JDT	0		11	8	0	0	0	0		5	3		0	
Mylyn	-		-		-			-	-					
WebTools	-			-	-	-		-			-	-		
RSE	-		-	-		-	-							
EclipseLink	-		-	-			-		-	-				
PDE	0		0		0	0		0		0			0	
Datatools	-		-	-	-	-				-		-	-	
CDT				-	-		-		-	-				
BIRT					-	-				-			-	
GMF	-			-	-	-			-	-				
PTP				-		-	-		-	-				
Scout			-		-			-		-				
Jubula				-	-		-		-		-			
RAP			-	-			-	-		-				
WindowBuilder			-		-		-		-					
Maven			-	-						-	-			
SVN			-			-	-	-	-	-		-		-
SVN15			-			-			-					-
SVN16							-	-		-		-		

Products 4–5, 7 and 11–12 pass all relevant tests. For both features CVS and PDE, all products pass all tests. For product 2–3 and 9–10, the JDT test suites produce 11, 8, 5 and 3 error respectively. For the RCP-platform test suites, there are various number of errors for products 1–3, 6, 8–10 and 13.

We executed the test several times to ensure that the results were not coincidental, and we did look at the execution log to make sure that the problems were not caused by the experimental set up such as file permissions, lacking disk space or lacking memory. We did not try to identify the concrete bugs behind the failing test cases, as this would require extensive domain knowledge that was not available to us during our research.[8]

4.4 Implementation with Eclipse Platform's Plug-in System

Algorithm 3 shows the algorithm of our testing technique for the Eclipse Platform plug-in system[9].

[8] We will report the failing test cases and the relevant configuration to the Eclipse project, along with the technique used to identify them.

[9] The source code for this implementation including its dependencies is available through the paper's resource website, along with the details of the test execution and detailed instructions and scripts to reproduce the experiment.

Algorithm 3. Pseudo Code of Eclipse-based version of Algorithm 1

```
1: $CA \leftarrow FM.GenerateCoveringArray(t)$
2: for each configuration $c$ in $CA$ do
3:    $p \leftarrow GetBasicEclipsePlatform()$
4:    for each feature $f$ in $c$ do
5:       $p.installFeature(f)$
6:    end for
7:    for each feature $f$ in $c$ do
8:       $tests \leftarrow f.getAssociatedTests()$
9:       for each test $test$ in $tests$ do
10:         $p.installTest(test)$
11:         $result \leftarrow p.runTest(test)$
12:         $table.put(result, c, f)$
13:      end for
14:   end for
15: end for
```

The algorithm assumes that the following is given: a feature model, FM, and a coverage strength, t.

In the experiment in the previous section we provided the feature model in Figure 1. The algorithm loops through each configuration in the covering array. In the experiment, it was the one given in Table 1b. For each configuration, a version of Eclipse is constructed: The basic Eclipse platform is distributed as a package. This package can be extracted into a new folder and is then ready to use. It contains the capabilities to allow each feature and test suite can be installed automatically using the following command: `<eclipse executable> -application org.eclipse.equinox.p2.director -repository <repository1,...> -installIU <feature name>` Similar commands allow tests to be executed.

A mapping file provides the links between the features and the test suites. This allows Algorithm 3 to select the relevant tests for each product and run them against the build of the Eclipse IDE. The results are put into its entry in the result table. The results from the algorithm are in a table like the one given in the experiment, shown in Table 5b.

5 Benefits and Limitations

Benefits

– **Usable**: The technique is a fully usable software product line testing technique: It scales, and free, open source algorithms and software exists for doing all the automatic parts of the technique.[10]
– **Agile**: The technique is agile in that once set up, a change in a part of the product line or to the feature model will not cause any additional manual

[10] Software and links available on the paper's resource website: `http://heim.ifi.uio.no/martifag/ictss2012/`

work. The product line tests can be rerun with one click throughout development. (Of course, if a new feature is added, a test suite for that feature should be developed.)
- **Continuous Integration**: The technique fits well into a continuous integration framework. At any point in time, the product line can be checked out from the source code repository, built and the testing technique run. For example, the Eclipse project uses Hudson [19] to check out, build and test the Eclipse IDE and its dependencies at regular intervals. Our technique can be set up to run on Hudson, and every night produce a result table with possible interaction faults in a few hours on suitable hardware.
- **Tests the feature model**: The technique tests the feature model in that errors might be found after a change of it. For example, that a mandatory relationship is missing causing a feature to fail.
- **Automatic**: The technique is fully automatic except making the test suites for each feature, a linear effort with respect to the number of features, and making the build-scripts of a custom product, a one-time effort.
- **Implemented**: The technique has been implemented and used for CVL-based product lines and for Eclipse-based product lines, as described in Section 4.
- **Run even if incomplete**: The technique can be run even if the product line test suites are not fully developed yet. It supports running a partial test suite, e.g. when only half of the test suites for the features are present, one still gets some level of verification. For example, if a new feature is added to the product line, a new test suite is not needed to be able to analyze the interactions between the other features and it using the other feature's test suites.
- **Parallel**: The technique is intrinsically parallel. Each product in the covering array can be tested by itself on a separate node. For example, executing the technique for the Eclipse IDE could have taken approximately 1/13th of the time if executed on 13 nodes, taking approximately in 2 hours instead of 23.

Limitations

- **Emergent features**: Emergent features are features that emerge from the combination of two or more features. Our technique does not test that an emergent feature works in relation to other features.
- **Manual Hardware Product Lines**: Product line engineering is also used for hardware systems. Combinatorial interaction testing is also a useful technique to use for these products lines [20]; however, the technique described in this paper is not fully automatic when the products must be set up manually.
- **Quality of the automated tests**: The quality of the results of the technique is dependent on the quality of the automated tests that are run for the features of the products.
- **Feature Interactions**: A problem within the field of feature interaction testing is how to best create tests as to identify interaction faults occur between two or more concrete features, *the feature interaction problem* [21].

Although an important problem, it is not what our technique is for. Our technique covers all simple interactions and gives insight into how they work together.

6 Conclusion

In this paper we presented a new technique for agile and automatic interaction testing for product lines. The technique allows developers of product lines to set up automatic testing as a part of their continuous integration framework to gain insight into potential interaction faults in their product line.

The technique was evaluated by presenting the results from two applications of it: one to a simulation model of the ABB safety module product line using the CVL tool suite, and one to the Eclipse IDE product lines using the Eclipse Platform plug-in system. The cases show how the technique can identify interaction faults in product lines of the size and complexity found in industry.

Acknowledgments. The work presented here has been developed within the VERDE project ITEA 2 - ip8020. VERDE is a project within the ITEA 2 - Eureka framework.

The authors would like to thank the anonymous reviewers for their helpful feedback.

References

1. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering 34, 633–650 (2008)
2. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 638–652. Springer, Heidelberg (2011)
3. Johansen, M.F., Haugen, Ø., Fleurey, F.: An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In: Alves, V., Santos, A. (eds.) Proceedings of the 16th International Software Product Line Conference (SPLC 2012). ACM (2012)
4. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, pp. 139–148. IEEE Computer Society, Washington, DC (2008)
5. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
6. Rivieres, J., Beaton, W.: Eclipse Platform Technical Overview (2006)
7. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)

8. Czarnecki, K., Eisenecker, U.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
9. Johansen, M.F., Haugen, Ø., Fleurey, F.: A Survey of Empirics of Strategies for Software Product Line Testing. In: O'Conner, L. (ed.) Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011, pp. 266–269. IEEE Computer Society, Washington, DC (2011)
10. Ganesan, D., Lindvall, M., McComas, D., Bartholomew, M., Slegel, S., Medina, B., Krikhaar, R., Verhoef, C.: An analysis of unit tests of a flight software product line. Science of Computer Programming (2012)
11. Engström, E., Runeson, P.: Software product line testing - a systematic mapping study. Information and Software Technology 53(1), 2–13 (2011)
12. Reuys, A., Reis, S., Kamsties, E., Pohl, K.: The scented method for testing software product lines. In: Käkölä, T., Dueñas, J.C. (eds.) Software Product Lines, pp. 479–520. Springer, Heidelberg (2006)
13. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Transactions on Software Engineering 36(3), 309–322 (2010)
14. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing combinatorics in testing product lines. In: Proceedings of the Tenth International Conference on Aspect-oriented Software Development, AOSD 2011, pp. 57–68. ACM, New York (2011)
15. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
16. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30(6), 418–421 (2004)
17. Garvin, B.J., Cohen, M.B.: Feature interaction faults revisited: An exploratory study. In: Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE 2011) (November 2011)
18. Steffens, M., Oster, S., Lochau, M., Fogdal, T.: Industrial evaluation of pairwise spl testing with moso-polite. In: Proceedings of the Sixth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2012 (January 2012)
19. Moser, M., O'Brien, T.: The Hudson Book. Oracle Inc. (2011)
20. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 269–284. Springer, Heidelberg (2012)
21. Bowen, T., Dworack, F., Chow, C., Griffeth, N., Herman, G., Lin, Y.: The feature interaction problem in telecommunications systems. In: Seventh International Conference on Software Engineering for Telecommunication Switching Systems, SETSS 1989, pp. 59–62. IET (1989)