

# Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines

Martin Fagereng Johansen<sup>1,2</sup>, Øystein Haugen<sup>1</sup>, Franck Fleurey<sup>1</sup>,  
Anne Grete Eldegard<sup>3</sup>, and Torbjørn Syversen<sup>3</sup>

<sup>1</sup> SINTEF ICT, Pb. 124 Blindern, 0314 Oslo, Norway

{Martin.Fagereng.Johansen, Øystein.Haugen, Franck.Fleurey}@sintef.no

<sup>2</sup> Institute for Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway

<sup>3</sup> Tomra Systems ASA, Drengsrudhagen 2, 1385 Asker, Norway  
{Anne.Grete.Eldegard, Torbjorn.Syversen}@tomra.no

**Abstract.** Combinatorial interaction testing is an approach for testing product lines. A set of products to test can be set up from the covering array generated from a feature model. The products occurring in a partial covering array, however, may not focus on the important feature interactions nor resemble any actual product in the market. Knowledge about which interactions are prevalent in the market can be modeled by assigning weights to sub-product lines. Such models enable a covering array generator to select important interactions to cover first for a partial covering array, enable it to construct products resembling those in the market and enable it to suggest simple changes to an existing set of products to test for incremental adaption to market changes. We report experiences from the application of weighted combinatorial interaction testing for test product selection on an industrial product line, TOMRA's Reverse Vending Machines.

**Keywords:** Product Lines, Software, Hardware, Testing, Combinatorial Interaction Testing, Evolution.

## 1 Introduction

A product line is a collection of systems with a considerable amount of software or hardware components in common [14]. The commonality and differences between the systems are usually modeled as a feature model [12]. Testing product lines is a challenge since the number of possible configurations generally grows exponentially with the number of features in the feature model. Yet, one has to ensure that any valid product will function correctly. There is no consensus yet on how to efficiently test product lines, but there are a number of suggested approaches [7].

A first level of product line testing is testing the software or hardware components in isolation to ensure that they function correctly on their own, a technique seen in industry [9]. Still there may be errors in the interaction between the features. Combinatorial interaction testing [4] is a promising approach for performing interaction testing between the features of a product line.

Based on this, we decided to try out combinatorial interaction testing on TOMRA's product line of reverse vending machines. Reverse vending machines handle the return of deposit beverage containers at retail stores such as supermarkets, convenience stores and gas stations. The feature model for the part of their product line we study has 68 features that potentially combine to 435,808 different configurations.

At TOMRA Verilab they are responsible for testing these machines. They already have a set of test products and were interested in applying new theory and techniques from product line testing research to understand the quality of their current test process and to improve it.

We found that their existing test lab covered a high percentage of the possible simple feature interactions. But, when we generated a new test lab from scratch of the same size as the current test lab, we encountered a problem. Even though the generated machines were valid machines that could be constructed, and even though they did test more of the simple interactions between features with the same number of products, they neither resembled any realistic machine that would be found in the market nor did any subset of products cover the most prevalent interactions.

A solution to these problems was to partition the machines in the market into sub-product lines, partially configured feature models; and to assign weights on them reflecting the number of products that are instances of this particular sub-product line. Modeling weights and sub-product lines proved to be a simple and intuitive way to capture relevant domain-knowledge in a feature model. It is close to the way the domain experts reason about the market and what is most important to be verified.

By generating covering arrays by prioritizing interactions according to their weight, we generated products that resemble the products in the market and that covered as many simple interactions as possible. This caused fewer interactions to be covered, but those interactions that were covered were more relevant according to the market situation.

The weighted sub-product line models also gave us an unexpected benefit. It enabled us to set up an evolution process for the test lab to incrementally adapt it to a continually changing market situation.

In addition to the application to TOMRA's product line, we briefly show how the technique can be used on the Eclipse IDE<sup>1</sup> software product line.

The generation, analysis and evolution based on weighted sub-product line models have been implemented in a fully functional tool freely available as open source on the paper's resource website<sup>2</sup>. The generation of covering arrays in the tool is done using the ICPL algorithm, an algorithm we have developed to generate covering arrays from large feature models [10,11].

---

<sup>1</sup> The Eclipse IDE is provided by the Eclipse Foundation and is independent from the TOMRA case.

<sup>2</sup> <http://heim.ifi.uio.no/martifag/models2012/>. Two example models, covering arrays and weighted sub-product line models are also available on this website.

This paper is structured as follows. In Section 2 we cover relevant background information and related work. In Section 3 we introduce models of weighted sub-product lines that enable covering array generation algorithms to select and evolve collections of products to test. In Section 4 we present the models and experiences from applying the techniques to an industrial product line at TOMRA and in Section 5 briefly describe the applicability to testing Eclipse IDEs. The paper ends with the conclusion, Section 6.

## 2 Background and Related Work

### 2.1 Product Lines

As stated in the introduction, a product line is a collection of systems with a considerable amount of hardware or software in common. The primary motivation for structuring one's systems as a product line is to allow customers to have a system tailored for their purpose and needs, while still avoiding redundancy of hardware or code. It is not uncommon for customers to have conflicting requirements. In that case, it is not even possible to ship one system for all customers. In the case of hardware, it would be uneconomical to ship unused components.

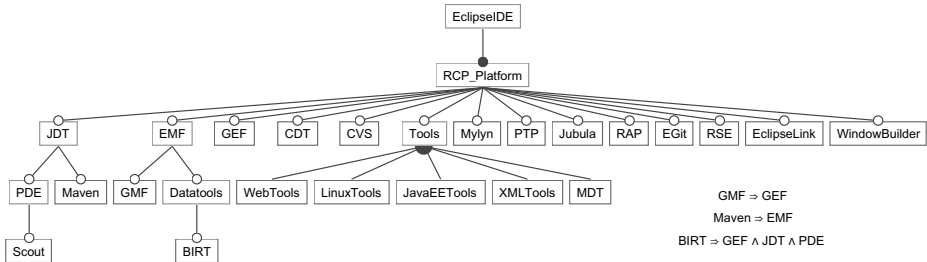
The Eclipse IDE products [2] can be seen as a software product line. Today, the Eclipse project lists 12 products on their download page<sup>3</sup>. The configurations of these products are shown in Table 1a<sup>4</sup>. These products share many components, but all components are not offered together as one single product. The reason is that the download would be unnecessary large, since, for example, a C++ systems programmer usually does not need to use the PHP-related features. It would also bloat the system by giving the user many unnecessary alternatives when, for example, creating a new project. Some products contain early developer releases of some components, such as Eclipse for modeling. Including these would compromise the stability for the other products. Thus, it should be clear why offering specialized products for different use cases is good.

One way to model the commonalities and differences in a product line is using a feature model [12]. 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. Please refer to an example of a feature model for a subset of Eclipse in Figure 1.

Proceeding from the root, configuring the product line consists of making a decision for each node in the tree. Each node represents a feature of the product line. The nature of this decision is modeled as a decoration on the edges going from a node to another. For example, in Figure 1, a filled circle means that the feature is mandatory, and an empty circle means that it is optional. A filled semi-circle on the outgoing edges means that at least one of the features underneath must be selected. An empty semi-circle means that one and only one must be

<sup>3</sup> <http://eclipse.org/downloads/> as of 2012-03-09.

<sup>4</sup> The original version of this table was found at <http://www.eclipse.org/downloads/compare.php>, 2012-03-28.



**Fig. 1.** Feature model for a significant part of the Eclipse IDE product line supported by the Eclipse Project

selected. In addition, constraints not effectively modeled on the tree are written underneath the model as propositional constraints; for example, when GMF is selected, it implies that GEF must also be selected.

The parts that can be different in the products of a product line are usually called its *variability*. One particular product in the product line is called a *variant* and is specified by a configuration of the feature model. A configuration consists of specifying whether each feature is included or not.

## 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 line functions correctly. One way to verify a product line is through testing, but testing is done on a running system. The product line is simply a collection of many products. The number of possible configurations generally grows exponentially with the number of features in the feature model. For the feature model in Figure 1, the number of possible configurations is 1,900,544, and this is a relatively simple product line.

There is no single recommended approach available today for testing product lines efficiently [7], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [4], discussed below; reusable component testing, which we have seen in industry [9], but which does not test for interaction faults in the product line; 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 [15]; 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 [16].

## 2.3 Combinatorial Interaction Testing for Product Lines

*Combinatorial interaction testing* [4] 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 subset of products which can then be tested using single system testing techniques, of which there are many good ones [3]. The idea is to select a small subset of products where the interaction faults are most likely to occur. For example, we can select the subset of all possible products where each pair of features is present. This includes the cases where both features are present; when one is present and when none of the two are present. Table 1b shows the 12 products that must be tested to ensure that every interaction between two features in the running example functions correctly, a 2-wise covering array. 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 mandatory, and some pairs are not covered since they are invalid according to the feature model.

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

(a) 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	-	-	-	-	-	-	-
EMF	X	X	-	-	X	X	-	-	-	-	-	-
GEF	X	X	-	-	X	X	-	-	-	-	-	-
JDT	X	X	-	-	X	X	-	-	-	X	-	-
Mylyn	X	X	X	X	X	X	X	X	X	X	-	-
Tools	X	X	X	X	X	X	X	X	X	X	-	-
WebTools	-	X	-	-	-	X	-	-	-	X	-	-
LinuxTools	-	-	X	X	-	-	X	-	-	-	-	-
JavaEETools	-	X	-	-	-	X	-	-	-	-	-	-
XMLTools	X	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	-	-	-	-	-
MDT	-	-	-	-	-	X	-	-	-	-	-	-
Scout	-	-	-	-	-	-	-	X	-	-	-	-
Jubula	-	-	-	-	-	-	-	-	X	-	-	-
RAP	-	-	-	-	X	-	-	-	-	-	-	-
WindowBuilder	X	-	-	-	-	-	-	-	-	-	-	-
Maven	X	-	-	-	-	-	-	-	-	-	-	-

(b) Complete 2-wise Covering Array												
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	-	-
EGit	X	X	X	-	-	X	X	-	-	-	-	-
EMF	X	X	X	-	X	-	X	X	-	X	X	-
GEF	X	X	-	X	-	X	X	X	-	X	-	-
JDT	X	-	X	X	X	-	X	X	-	-	-	-
Mylyn	-	X	X	X	-	-	X	-	-	-	-	-
Tools	X	X	X	X	X	X	X	X	-	-	-	-
WebTools	X	-	X	X	-	X	-	X	-	-	-	-
LinuxTools	X	-	X	X	-	X	X	X	-	-	-	-
JavaEETools	X	-	X	-	X	-	X	-	-	-	-	-
XMLTools	-	X	X	-	X	-	X	-	-	-	-	-
RSE	-	X	X	-	X	-	X	-	X	-	-	-
EclipseLink	-	X	X	-	X	-	X	X	-	-	-	-
PDE	X	-	X	X	-	X	X	-	-	-	-	-
Datatools	X	X	-	X	-	X	-	-	-	-	-	X
CDT	X	-	X	-	X	X	-	-	-	-	-	-
BIRT	X	-	X	-	X	-	X	X	-	-	-	-
GMF	X	X	-	X	-	X	-	-	X	-	X	-
PTP	-	X	-	X	X	X	-	-	-	-	-	-
MDT	X	-	X	X	-	X	X	-	-	-	-	-
Scout	-	-	-	X	X	-	X	X	-	-	-	-
Jubula	-	-	X	X	-	X	-	X	X	-	-	-
RAP	X	-	X	-	X	-	X	-	-	-	-	-
WindowBuilder	X	-	X	-	X	-	-	X	X	X	-	-
Maven	X	-	X	-	X	-	-	X	-	-	-	-

2-wise covering arrays are a special case of  $t$ -wise covering arrays where  $t = 2$ . 1-wise coverage means that every feature is at least included and excluded in at least one product. 3-wise coverage means that every combination of three features are present, etc. For our running example, 2, 12 and 37 products are sufficient to achieve 1, 2 and 3-wise coverage, respectively.

An important motivation for combinatorial interaction testing is a paper by Kuhn et al. 2004 [13]. 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. Kuhn et al.

2004 result, however is not about combinations of features, but about combinations of program input. A recent study by Garvin and Cohen 2011 [8] checked whether Kuhn et al. 2004’s result also holds for feature interaction faults. They investigated 250 faults of two real-world, open source systems. Of these faults 28 were found to be configuration dependent and three to be true interaction faults. In addition, they conclude that exercising feature interactions traverses more of the product line’s behavior. This indicates that Kuhn et al. 2004’s result is also applicable for feature interaction faults.

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 subset of products must be generated. We have developed an algorithm that can generate such arrays from large features models [11]. 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.


### 3 Weighted Combinatorial Interaction Testing

As explained earlier, in order to successfully apply combinatorial interaction testing at TOMRA, we had to extend the technique by developing and using weighted sub-product line models. In this section, we describe the models and how they are used on a simple example, before discussing the more complex details and evaluations for the application at TOMRA in the next section.

**Ordinary Combinatorial Interaction Testing.** Figure 2a shows a simple feature model with 6 features. When applying ordinary combinatorial interaction testing, we would, for example, generate a 2-wise covering array (as in Table 2b), build the products and test them individually. That way, we know that all interactions between pairs of features have been tested.

**Table 2.** A Simple Example

(a) Feature Model



```

graph TD
    R[R] --- A((A))
    R --- B((B))
    R --- C((C))
    A --- D[D]
    A --- E[E]
    style A fill:#fff,stroke:#000
    style B fill:#fff,stroke:#000
    style C fill:#fff,stroke:#000
    style D fill:#fff,stroke:#000
    style E fill:#fff,stroke:#000
        
```

(b) Complete 2-wise covering array

	1	2	3	4	5	6
R	X	X	X	X	X	X
A	X	X	X	X	-	-
B	-	X	-	X	-	X
C	-	X	-	X	X	-
D	X	-	-	X	-	-
E	X	X	-	-	-	-

(c) Weighted Sub-product lines

	1	2
#Weight	100	10
R	X	X
A	-	?
B	X	X
C	?	X
D	-	?
E	-	?

**Initial Problems.** At this stage of the application of combinatorial interaction testing at TOMRA we faced a few problems:

- There were no ways to select a subset of the products to test that included the most important interactions. One could of course select a subset that covered as many interactions as possible, but it might not include some important interactions.
- The covering array generation algorithm generates a different result each time it is run. Since there are many equally good products, why not select the ones that look like some of the larger classes of sold products?

**A Solution.** These experiences revealed to us that an assumption of ordinary combinatorial interaction testing, that all interactions are equal, is not entirely the case in practice.

For TOMRA, there were certain market segments where the products are similar. It is important for TOMRA that a product containing the commonalities of these segments are tested. Thus, we decided: Let us model the products in each market segment as a sub-product line, and assign a weight to it according to how many products of that kind are in the market.

This will enable us to assign weights to each interaction in the sub-product line. These weights can then be used by a covering array generator to select the interactions with the most weight to cover first. This would cause the results to be similar each time the covering array algorithm is run and the first products produced would contain the most important interactions.

**Weighted Sub-product Lines.** For our simple example, there are two major market segments, modeled in Table 2c. These two sub-product lines are as follows: The first has R and B included and A, D and E excluded. The feature C is optional within this segment. The second segment has features R, B and C included while A, D and E are optional within this segment. Now, in this second example, the limitations imposed on the products by the feature model in Figure 2a still apply, of course. So, even though A, D and E are marked with a question mark, the products with A excluded also has D and E excluded. The first segment have 100 instances in the market while the second segment has only 10.

Now, what about legal products that are not in any market segment? For this example, a product with R included and B excluded is not found in the market. As we experienced at TORMA, the reason that they are not found is that they do not make sense even though they are structurally valid. This is valuable domain knowledge, and is of great value to a covering array generator: It enables it not to focus on the interactions that are of no practical importance.

**Algorithms.** An example should clarify how the covering array generation algorithm can use the weighted sub-product line models. This example uses 1-wise covering arrays since an example with 2-wise covering arrays would fill up several

pages due to the combinatorial explosion of interactions. This is not a problem for modern computers to deal with however.

We will use the terminology from a previous paper of ours [11]: An *assignment* is a pair with a feature name and a boolean. A *t-set* is a set of  $t$  assignments. A *configuration* is a set of assignments in which all features of the product line are given an assignment. The *universe* to cover is the set of all valid t-sets,  $U_t$ . Thus, a *t-wise covering array* is a set of configurations,  $C_t$ , such that  $\forall e \in U_t, \exists c \in C_t : e \subseteq c$ . A t-set can be written as for example  $\{(F1, X), (F2, -)\}$ , a 2-set with  $F1$  included and  $F2$  excluded.

For our simple example, the following 11 t-sets need to be in a product to achieve 1-wise coverage:  $\{(R, X)\}, \{(A, X)\}, \{(A, -)\}, \{(B, X)\}, \{(B, -)\}, \{(C, X)\}, \{(C, -)\}, \{(D, X)\}, \{(D, -)\}, \{(E, X)\}, \{(E, -)\}$ . Note that the assignment with  $R$  excluded is not present since in feature modeling the root must always be included.

Now, we can assign weights to each t-set. In Table 2c, whenever a t-set is present in a sub-product line, the weight is added to the t-set. If there is a question-mark, half the weight is given to each assignment. For example  $(A, X)$  gets 5 because it is not present in the first sub-product line and only as an option in the second. One t-set is not present in any sub-product line and therefore gets the weight zero:  $\{(\{(R, X)\}, 110), (\{(A, X)\}, 5), (\{(A, -)\}, 105), (\{(B, X)\}, 110), (\{(B, -)\}, 0), (\{(C, X)\}, 60), (\{(C, -)\}, 50), (\{(D, X)\}, 5), (\{(D, -)\}, 105), (\{(E, X)\}, 5), (\{(E, -)\}, 105)\}$ .

These t-sets can now be ordered according to their weights:  $\{(\{(R, X)\}, 110), (\{(B, X)\}, 110), (\{(A, -)\}, 105), (\{(E, -)\}, 105), (\{(D, -)\}, 105), (\{(C, X)\}, 60), (\{(C, -)\}, 50), (\{(A, X)\}, 5), (\{(D, X)\}, 5), (\{(E, X)\}, 5), (\{(B, -)\}, 0)\}$ .

Now, the circumstances warrants two kinds of coverages. The ordinary type of coverage is *t-set coverage*. The goal of combinatorial interaction testing is to cover as many simple interactions as possible, and ultimately a t-set coverage of 100%, that is, cover all t-sets. Since we have introduced weights for each t-set, talking about *weight coverage* makes sense. The goal of weight coverage is then to cover the most weight possible, and ultimately cover all the t-sets with weight, that is, achieve 100% weight coverage.

Just as t-set coverage is found by taking the number of covered t-sets and dividing it by the total number of valid t-sets,  $|U_t|$ , similarly weight coverage is found by taking the covered weight divided by the total weight.

The total weight of our example is the sum of all the weights of all the t-sets: 660. Now, if we were to generate a single product to test from the weighted t-sets, we would get the following:  $\{(R, X)\}, \{(A, -)\}, \{(B, X)\}, \{(C, X)\}, \{(D, -)\}, \{(E, -)\}$ . The weight covered by this product is the sum of all weights of the covered t-sets: 595. Thus, the weight coverage of this single product is  $595/660 \approx 90\%$ . This number can be contrasted with the t-set coverage of this product which is  $6/11 \approx 55\%$ . The high weight coverage indicates to us that we have most of the important t-sets (given the current market situation) are in the product. Any valid product would, however, give the same t-set coverage, but only that product would give such a high weight coverage.

Note that 100% weight coverage can mean that we have less than 100% t-set coverage since some t-sets can have zero weight. To ensure that 100% weight



coverage means 100% t-set coverage, include a sub-product line with all question marks and a non-zero weight.

**Evolution of Test Products.** A goal of testing is to gain confidence in the products that are sold to or used by the customers. Weights on sub-product lines can be set up to reflect the market situation, but could also include expected sales. When the market situation or the expectations change, the weights and the sub-product lines can change. This does not mean that the test lab or the feature model is changed. It will, however, mean that the weight coverage of the products that are currently being tested changes.

A simple algorithm can suggest simple changes to a set of test products. By calculating the coverage of the current test products, and then the new coverage given 1, 2 or 3 (or even more) changes of it, a list of possible changes can be made<sup>5</sup>. If the best changes are applied to the lab incrementally, the test products can evolve over time to converge on the current market situation, even if it changes during the evolution.

A special case of this is the introduction of a new feature in the feature model. The expected sales of this new feature can be added to the weighted sub-product line models. Including it for at least one of the products in the set of test products will probably be the best way to increase weight coverage. Thus, this decision is automated by our approach.

**Sub-Product Lines, Related Work.** Czarnecki et al. 2004 [5] introduced the idea of staged configuration. The stages are the production of a new sub-product line from a previous one. The difference between their work and ours is that we apply sub-product lines to modeling the market situation for use in testing, while they use it during product line development.

Their ideas are further developed in Czarnecki et al. 2005 [6]. Our view is similar to theirs in that the sub-product lines are specializations of the complete feature model to certain market segments. It is on the basis of these specializations that domain experts do their daily work.

Batory 2005 [1] integrates the idea of staged configurations with the formalization of feature models as propositional constraints. He formalizes feature models as propositional formulas. In his work, a sub-product line is a propositional formula where some variables, representing features, set to 'true' for included, some set to 'false' for excluded, and the unset features set to 'unknown'. The 'unknown' classification is the same as the questions-marks in our sub-product line models.

## 4 Industrial Application: TOMRA

In this section, we report from an industrial application of our technique at TOMRA Verilab.

---

<sup>5</sup> An implementation of this algorithm is available on the paper's resource website. It supports searching for 1–3 changes for improving 1–3 wise coverage.

**About TOMRA Verilab.** TOMRA Verilab is the part of TOMRA that is responsible for testing TOMRA's reverse vending machines (RVMs). Reverse vending machines handle the return of deposit beverage containers at retail stores such as supermarkets, convenience stores and gas stations. In Norway, customers are required by law to pay an amount for each container they buy which is given back to them if they decide to return the container.

**RVMs.** The RVMs are delivered all over the world and the market is expanding. However, individual market requirements and the needs of TOMRA's customers within the different markets can vary significantly. TOMRA's reverse vending portfolio therefore offers a high degree of flexibility in terms of how a specific installation is configured.

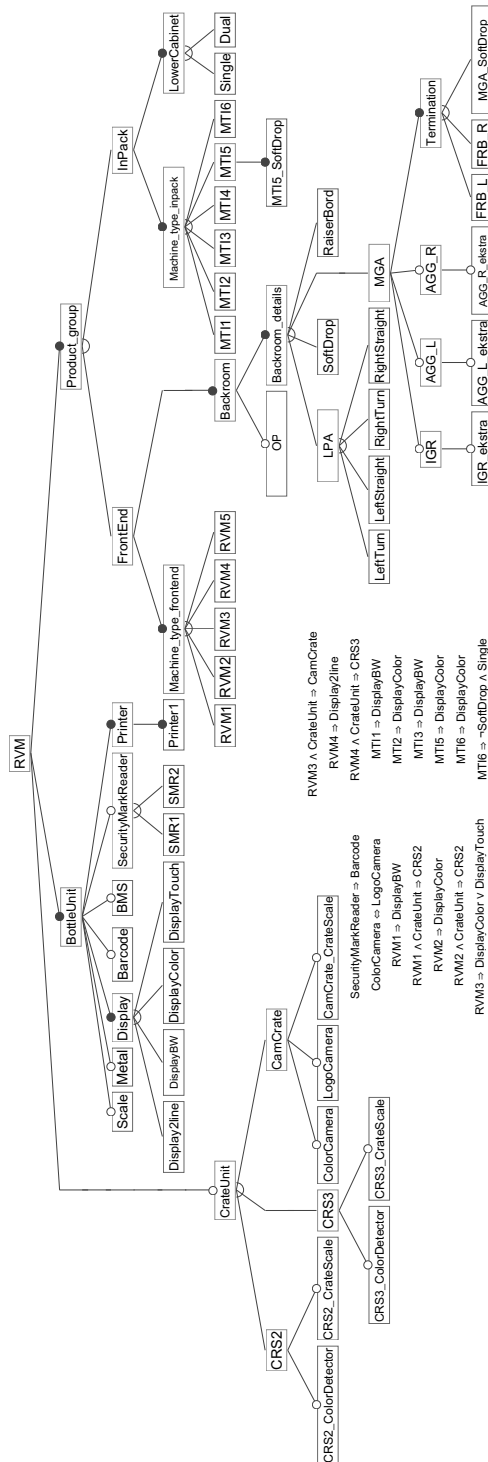
Figure 2 shows a part of the feature model for TOMRA RVMs. The feature model has 68 features, and a huge number of possible configurations (435,808, to be exact). Variation can include such things as the quality of the display used (e.g. black and white, color, touch-screen interface), the type of storing and sorting facilities the system has, as well as different container recognition technologies utilized for identifying container security marks, material and other characteristics.

**Test Lab.** In order to test the RVMs, TOMRA Verilab has set up a test lab of machines configured by hand to ensure the quality of the machines in the market. Parts of these product configurations are shown in Table 3. These products are both automatically and manually tested. The software is partly tested automatically by installing and running test suites on the machines. The manual tests are run by, for example, inserting bottles of various kinds in various ways, orders and magnitudes.

**Sub-Product Lines and Weights.** As discussed earlier, the results produced by ordinary covering array generation was not suited for TOMRA for various reasons. To solve these, we modelled the weighted sub-product lines as partly shown in Table 4.

**Existing Coverages.** The first experiment was to measure the t-set and weight coverage of the existing test lab, shown in part in Table 3. Recall that coverage is measured by taking the covered valid t-sets and then dividing either their number or their weight by the total number or t-sets or the total weight, respectively.

Figure 3a shows the 1–3-wise, t-set and weight coverage for the existing test lab. The weight coverage is consistently higher for the weight coverage than for the t-set coverage. This is consistent with the fact that the developers paid attention to the market situation when designing the test lab manually. It also suggests that weighted sub-product line models are a guide to test product selection for TOMRA Verilab.



**Fig. 2.** Part of the Feature model of TOMRA's Product Line of Reverse Vending Machines

**Table 3.** Part of TOMRA’s Actual Test Lab

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
RVM	X	X	X	X	X	X	X	X	X	X	X	X
CrateUnit	X	-	X	-	-	-	X	X	X	-	-	X
...												
BottleUnit	X	X	X	X	X	X	X	X	X	X	X	X
Display	X	X	X	X	X	X	X	X	X	X	X	X
Display2line	-	-	X	-	-	-	-	-	-	-	-	-
DisplayBW	X	X	-	X	X	-	-	-	-	-	-	-
DisplayColor	-	-	-	-	-	X	X	X	X	X	-	-
DisplayTouch	-	-	-	-	-	-	-	-	-	-	-	X
Scale	-	X	X	X	X	X	X	X	X	X	X	X
Metal	X	X	-	X	X	X	X	X	X	X	X	X
Barcode	X	X	-	X	X	X	X	X	X	X	X	X
BMS	-	-	-	X	-	-	-	-	-	-	-	-
SecurityMarkReader	-	X	-	X	-	-	X	-	-	-	X	X
SMR1	-	X	-	-	-	-	-	-	-	-	-	-
SMR2	-	-	-	X	-	-	X	-	-	-	X	X
Printer	X	X	X	X	X	X	X	X	X	X	X	X
Printer1	X	X	X	X	X	X	X	X	X	X	X	X
Product-group	X	X	X	X	X	X	X	X	X	X	X	X
FrontEnd	X	X	X	-	-	-	X	X	X	-	-	X
...												
Backroom	X	X	X	-	-	-	X	X	X	-	-	X
Backroom_details	X	X	X	-	-	-	X	X	X	-	-	X
OP	-	-	-	-	-	-	-	X	-	-	-	-
LPA	-	X	-	-	-	-	-	-	-	-	-	-
...												
SoftDrop	-	-	-	-	-	-	-	-	-	-	-	-
...												
RaiserBord	X	-	X	-	-	-	-	X	X	-	-	-
...												

**Generated Coverages.** The second experiment was to generate a partial covering array from scratch using both t-set and weight coverage and compare them to each other. The results are shown in Figure 3b. We can see that for 1–3-wise covering arrays, the weighted covering arrays are consistently smaller than t-set-based covering arrays. This suggests that either it would have been beneficial to used weighted covering array generation from the start, or that the current test lab is outdated with respect to the current market situation.

**Suggesting Improvements.** The third experiment was to find small modifications that can be done on the existing test lab at TOMRA to increase the weight coverage. Some such suggestions are shown in Table 5.

The search for improvements is done by flipping a set of assignments and, if the new configuration is valid, recalculating the new weight coverage. If the coverage is better than the original one, the changes and the new coverage are recorded.

In Table 5, we have recorded some suggestions for improving the 2-wise weight coverage of the existing test lab. The original 2-wise weight coverage was 95.8% (Table 3). We can achieve an increase of 0.2 pp by excluding feature *Metal* for product 11, an increase of 0.7 pp by including feature *SoftDrop* and excluding feature *RaiserBord* for product 1. Finally, we achieve an increase of 0.8 pp by excluding *Metal*, including *SoftDrop* and excluding *RaiserBord* for product 1.

**Table 4.** Part of TOMRA’s Sub-Product Lines and Their Weights

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
#Weight	478	478	1140	500	500	333	333	257	5120	581	818	1525	1001	500	1225	125			
RVM	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CrateUnit	-	X	?	X	X	-	X	-	X	?	?	?	-	X	-	-	-	-	-
...																			
BottleUnit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Display	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Display2line	-	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-
DisplayBW	?	?	?	?	?	-	-	-	-	-	-	-	-	-	?	?	?	?	?
DisplayColor	?	?	?	?	?	X	X	?	?	?	?	?	-	X	?	?	?	?	X
DisplayTouch	-	-	-	-	-	-	-	?	?	?	?	?	-	-	-	-	-	-	-
Scale	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Metal	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Barcode	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
BMS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SecurityMarkReader	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
SMR1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	-
SMR2	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
Printer	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Printer1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Product_group	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
FrontEnd	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
...																			
Backroom	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
Backroom_details	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
OP	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	-	-	-	-
LPA	-	-	X	-	-	?	?	-	-	X	-	-	?	?	-	-	-	-	-
...																			
SoftDrop	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	?	?	?	-
...																			
RaiserBord	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	-	-	-	-
...																			

Generating the suggestions on our machine<sup>6</sup> took 49s, 141s and 1,090s for an improvement in 2-wise covering of 1, 2 and 3 suggestions of changes respectively.

The next set of runs (4–6) is on an improved version of the original test lab. We chose to apply suggestion 1, to exclude feature *Metal* for product 11, and got a new test lab of 2-wise weight coverage of 96.0%. The improvements can be read in the same way as the previous set of suggestions.

The next set of runs (7–9) is on another improved version of the original test lab. We chose to apply suggestion 2 to the original lab, to include feature *SoftDrop* and exclude feature *RaiserBord* for product 1, and got a new test lab of 2-wise weight coverage of 96.5%.

Finally, we applied suggestions 1 and 2 to the original test lab to get a coverage of 96.3%. This produces suggestions 10–12.

The best coverage was achieved by changing four features by applying suggestions 2 and 9 to get a new weight coverage of 97.2%, up 1.4pp from 95.8%.

<sup>6</sup> Its specifications: Intel Q9300 CPU @2.53GHz and 8 GiB, 400MHz RAM. Each execution ran in parallel in 4 threads, as the computer had 4 logical processors.

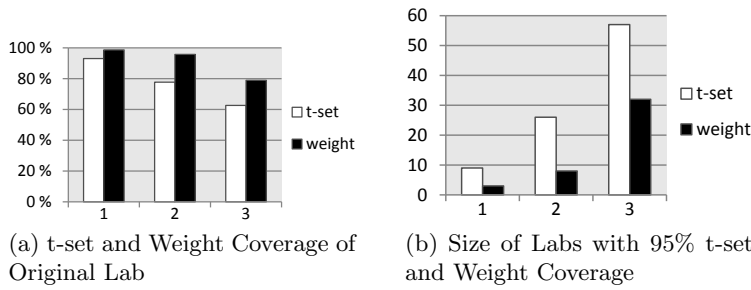


Fig. 3. Results of Two Experiments

Table 5. Simple changes to TOMRA’s test lab, Table 3, that produce higher coverage of the product line, Figure 2, based on the current market situation, Table 4

Change Suggestion	New Coverage	Product	Feature 1	Set	Feature 2	Set	Feature 3	Set
Starting from the lab machines with coverage 95.8%								
1	96.0%	11	Metal	-				
2	96.5%	1	SoftDrop	X	RaiserBord	-		
3	96.6%	1	Metal	-	SoftDrop	X	RaiserBord	-
Starting from lab machines with suggestion 1, with coverage 96.0%								
4	96.3%	10	Barcode	-				
5	96.7%	1	SoftDrop	X	RaiserBord	-		
6	96.9%	1	Barcode	-	SoftDrop	X	RaiserBord	-
Starting from lab machines with suggestion 2, with coverage 96.5%								
7	96.7%	11	Metal	-				
8	97.0%	11	Metal	-	Scale	-		
9	97.2%	10	Metal	-	Scale	-	Barcode	-
Starting from lab machines with suggestion 1 and 4, with coverage 96.3%								
10	96.5%	11	Scale	-				
11	97.0%	1	SoftDrop	X	RaiserBord	-		
12	96.5%	3	Scale	-	SoftDrop	X	RaiserBord	-

5 Applicability to the Eclipse IDEs

As an indication of the generality of our approach, we did an experiment to see if it also made sense for the product line of Eclipse IDEs. The Eclipse IDE can be seen as a product line; it was introduced in Section 2. The actual products offered on the Eclipse website was shown in Table 1a.

One source of information about what the users of the Eclipse IDE have is the download statistics reported on the Eclipse project’s download pages<sup>7</sup>. These can be used as weights<sup>8</sup>. The weights can be assigned to the product configurations themselves, which were previously shown in Table 1a. Table 6 shows the downloads as weights linked to each of the Eclipse products in Table 1a. In this table we can clearly see that some products are more downloaded than others.

<sup>7</sup> <http://eclipse.org/downloads/> as of 2012-03-09.  
<sup>8</sup> A better source of weights and sub-product lines is the data acquired by the Eclipse Usage Data Collector (UDC) that “[...] collects information about how individuals are using the Eclipse platform,” available online at [eclipse.org/org/usagedata/](http://eclipse.org/org/usagedata/).

**Table 6.** Eclipse IDE Product Configurations and Their Downloads as of 2012-03-09

Product	Name	Weight	Product	Name	Weight
1	Java	282,220	7	Reporting	33,813
2	JavaEE	856,493	8	Parallel	10,441
3	C/C++	58,720	9	Scout	1,130
4	C/C++ Linux	58,720	10	Testers	8,953
5	RCP/RAP	16,610	11	JavaScript	35,750
6	Modeling	22,060	12	Classic	651,616

By generating a 2-wise covering array using the feature model in Figure 1 with the weights from Table 6 on the product configurations in Table 1a, we found that just 4 products give more than 95% weight coverage. This clearly shows that our approach is most likely applicable outside the scope of the TOMRA industrial case.

## 6 Conclusion

In this paper we showed how an additional type of model was needed in order to effectively apply combinatorial interaction testing to an industrial product line. The new model captures relevant domain knowledge in a form that is close to the way domain experts reason about their domain and enables additional benefits to be derived from combinatorial interaction testing:

- Cover the interactions found in the market or planned to be in future products first.
- Generate products that both cover many simple interactions and resemble products found in the market.
- Incrementally evolve the test products for the continually changing market situation.
- Covering array generation is more deterministic.

We described our experiences of applying this to a product line of industrial size and complexity, the TOMRA RVMS.

The algorithms that implement these features in addition to the ordinary combinatorial interaction testing features are available on the paper's resource website as free and open source software.

**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.

## References

1. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

2. Beaton, W., Rivieres, J.: Eclipse platform technical overview. Tech. rep., The Eclipse Foundation (2006)
3. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
4. 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)
5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) *SPLC 2004*. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
6. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice* 10(2), 143–169 (2005)
7. Engström, E., Runeson, P.: Software product line testing - A systematic mapping study. *Information and Software Technology* 53(1), 2–13 (2011)
8. Garvin, B., Cohen, M.: Feature interaction faults revisited: An exploratory study. In: *IEEE 22nd International Symposium on Software Reliability Engineering (IS-SRE)*, pp. 90–99 (29 2011–December 2 2011)
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. 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)
11. 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)
12. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
13. 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)
14. Pohl, K., Böckle, G., van der Linden, F.J.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus (2005)
15. Reuys, A., Reis, S., Kamsties, E., Pohl, K.: The scented method for testing software product lines. In: Käkölä, T., Duenas, J.C. (eds.) *Software Product Lines*, pp. 479–520. Springer, Heidelberg (2006), doi:10.1007/978-3-540-33253-4\_13
16. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. *IEEE Transactions on Software Engineering* 36(3), 309–322 (2010)