Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing

Doctoral Dissertation by

Martin Fagereng Johansen

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree Philosophiae Doctor (PhD) in Computer Science

July 2013

Contents

Abstract

Due to varying demands by customers, some software systems need to be configurable and need to have optional features. Customers then configure their system according to their special needs and select the features they need. A problem the developers of such systems face is the possibility of latent faults awakened by some of its set-ups. A system used by thousands, if not by millions, cannot fail for many before it becomes a major problem for the developers. Indeed, if many systems fail, the general view of the quality of the system might be beyond repair. For example, the Eclipse IDEs are popular software systems; they are also highly customizable. There are probably tens of thousands of unique set-ups of the Eclipse IDEs in use.

This is one example of product line development, and the field that studies the construction and development of product lines is product line engineering. One way of gaining confidence in the quality of a product line—and any set-ups of it—is through testing. Product line testing, the strategic exercising of the product line in order to gain confidence in the quality of the product line and any configuration of it, is the subject of this thesis.

The strategy followed today with documented results is reusable component testing, the testing of the product line's common parts in isolation: If a common part fails in isolation because of an internal fault, it will likely cause failures in any product of which it is a part. Testing something in isolation will not, naturally, rule out interaction faults between these parts.

Combinatorial interaction testing (CIT) can test interactions and is—as of today—the technique closest to being realistically applicable in industrial settings. This technique is the starting point of the contributions of this thesis. CIT starts by sampling some products of the product line. The products are selected such that all combinations of a few features are in at least one of the products. This selection criterion is such that faults are more likely to show up in these products than in randomly selected products. As executable systems, they can be tested.

One significant problem with CIT is the lack of an efficient algorithm for the selection of products. The existing algorithms scale insufficiently for the larger product lines. This effectively rules it out in industrial settings. A contribution of this thesis is an algorithm (called ICPL) that can perform the selection process even for product lines of industrial size. A preliminary analysis of the problem was a necessary prior contribution contributed before the algorithm could be developed. These two contributions taken as a whole provides an efficient algorithm for this one bottle-neck of the application of CIT.

Having a scalable algorithm for selection enables the efficient application of CIT. To establish and demonstrate the usefulness of CIT, three advancements of CIT with applications were contributed.

Firstly, a common situation with product lines is portability across various hardware architectures, operating systems, database systems, etc. One construct that deals effectively with such situations is a homogeneous abstraction layer. It provides a uniform way of accessing these differing systems. Usually, only one implementation of a homogeneous abstraction layer can be present in a product. This deteriorates the performance of CIT. The selection criterion used by CIT is to strategically select simple combinations of features. Only being allowed to activate one and one feature is limiting to the algorithms. It was noticed that products differing only in their implementation of homogeneous abstraction layers can be tested using the same test suite. This enables a voting oracle or a gold standard oracle to be set up. This contribution thus turns the aforementioned problem into a strength.

Secondly, the selection criterion of CIT ensures all simple interactions are exercised. In some cases, this leads to redundant testing. A product line's market situation might exclude large classes of feature combinations or indicate then some combinations are more common than others. These things can be captured in a weighted sub-product line model, which, along with associated algorithms, is another contribution of this thesis.

Thirdly, one way to fully automate the application of CIT is to create the test cases before testing and then strategically allocate and automatically run those test cases in each application of CIT. Such a technique and a full application of it on the Eclipse IDEs using its existing test cases is also a contribution of this thesis.

In summary, this thesis contributes an algorithm that enables the application of CIT in industrial settings. Further, three advancements of CIT with applications were contributed to advance and demonstrate CIT as a product line testing technique.

Acknowledgments

This work has been conducted as a part of the Norwegian VERDE project funded by the Research Council of Norway. The Norwegian VERDE project participated in the international VERDE project ITEA 2 - ip8020. The international VERDE is a project within the ITEA 2 - Eureka framework.

First of all, I would like to thank my two supervisors Øystein Haugen and Franck Fleurey; they were naturally the closest cooperators during my PhD work on all issues. They provided me with the guidance I needed to get safely through this process, they were always available to answer my questions and for a good discussion. I could not have done it without their valuable guidance.

I have during the PhD studies been employed by SINTEF ICT as a research fellow at the initially named Department of Cooperative and Trusted Systems later renamed to the Department of Networked Systems and Services. I owe my thanks to the research director Bjørn Skjellaug and all the colleagues for providing a very pleasant and motivating work environment.

I would like to thank Anders Emil Olsen who, while working for TOMRA, introduced me and SINTEF to a highly interesting industrial case, the testing of TOMRA's reverse vending machines; it was not initially a part of the Norwegian VERDE project. It was a great experience working together with TOMRA Verilab's Anne Grete Eldegaard and Thorbjørn Syversen on this case study.

I also had the great opportunity to work together with Frank Mikalsen from Finale Systems on the Finale case study, and primarily Erik Carlson, Jan Endresen and Tormod Wien from ABB Corporate Research on the ABB case studies.

The VERDE project provided me with everything I wanted from a PhD. I got a firsthand experience with both the national and international research communities, and I got the firsthand experience of working together with people from three interesting companies on their industrial cases. Each gave me invaluable experience and provided a great start of a research career.

Last but not the least, I would like to thank my girlfriend Hilde Galleberg Johnsen for her continuous support on all matters during the years of my PhD work.

List of Original Publications

1. Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey, "Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible," *Model Driven Engineering Languages and Systems – 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16–21, 2011. Proceedings*, Springer, 2011, p. 638.

2. Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey, "An Algorithm for Generating t-wise Covering Arrays from Large Feature Models," *Software Product Lines – 16th International Conference, SPLC 2012, Salvador, Brazil, September 2–7, 2012. Proceedings – Volume 1*, ACM, 2012, p. 46.

3. Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey, "Bow Tie Testing – A Testing Pattern for Product Lines," *Pattern Languages of Programs – 16th European Conference, Proceedings*, ACM, 2012, p. 9:1.

4. Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen, "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines," *Model Driven Engineering Languages and Systems – 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings*, Springer, 2012, p. 269.

5. Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien, "A Technique for Agile and Automatic Interaction Testing for Product Lines," *Testing Software and Systems – 24th IFIP International Conference, ICTSS '12, Aalborg, Denmark, November 19–21, 2012. Proceedings*, Springer, 2012, p. 39.

Part I

Overview

Chapter 1

Introduction

"one element in the price of every article is the cost of its verification."
– Charles Babbage

One size does not always fit all; or one software system does not always fit all users. There are many situations, however, where they almost fit all. In these cases it might be better to make a configurable system, instead of several single systems. Indeed, the cost of single systems can be decreased, and the efficiency of developing them can be greatly increased by making a configurable system.

A company that produces, for example, t-shirts need to produce them of varying sizes for obvious reasons, or with a varying color or motives. Still, much of the production of these t-shirts is similar: They can use the same fabric, they can be dyed similarly and their motives applied with the same equipment. This is surely more efficient than having a separate production process for each kind of t-shirt.

A similar example could easily be mentioned for cars.

Historically, when products are offered in this manner, they were offered lined up, the customer browsing the products to find one that specifically suits his or her needs; thus, the term *product line*.

These notions were brought into software engineering; thus, the term *software product lines*. Their development has been studied in the field of software product line engineering (SPLE).

To continue the t-shirt analogy, because t-shirts of various sizes, colors and motives are made from the same fabric, dyes and motives, no one would be surprised to learn that several t-shirts made with the same quality fabric all are of good quality with respect to the fabric. This commonality, once verified, gives us a lot of information about the quality of the completed t-shirts. In the same way, a wrongly mixed dye might cause all t-shirts colored with it to drastically fade when washed. Washing one t-shirt of each dye will give us a lot of information about the other t-shirts colored with the same dyes.

The same analogy holds for software, but just how to gain confidence in the quality of any product of a product line, remains an open question. How to use testing for quality assurance of software product lines is the general problem of this thesis.

Testing the common parts out of which products are composed is already an established practice. As with the t-shirts, this gives us some level of confidence in the composed products.

It does not, however, tell us about the interaction between the parts. Various techniques are being researched to achieve this goal. Among the suggested techniques, this thesis advances one particular technique, *combinatorial interaction testing*. This technique is to select a small set of products such that each combination of t options (typically 1, 2 or 3) occurs in at least one of the products. For the t-shirts, it would mean producing t-shirts such that ever combination of, for example, 2 features are in at least one t-shirt: every combination of color and motif, every combination of motif and size, etc. These products are called the *t-wise* products. The t-wise criterion ensures that interaction faults are significantly more likely to occur in these selected products than in the same number of randomly selected products.

Counter-intuitively, this t-wise selection process is computationally expensive. It has even been argued that the selection is so difficult that it will remain infeasible for practical purposes. It is thus a bottle-neck in the application of combinatorial interaction testing. Contribution 1 and 2 of this thesis culminates in an algorithm that performs this selection efficiently enough to make combinatorial interaction testing usable in practice.

Having resolved this bottle-neck, the utilization of combinatorial interaction testing in industrial settings still requires issues to be addressed. The existence of a scalable means of selecting the products to test, however, opens up for and encourages further developments of the technique. Three such developments are the further contributions of this thesis.

1.1 Contributions

The main contributions of this thesis all relate to the testing of product lines using combinatorial interaction testing (CIT). Figure 1.1 shows the CIT testing process with the five contributions of this thesis marked as 1–5. The general technique of using CIT to test product lines has been studied for some time [42]. It starts with three assets, two of which are ubiquitous in software projects: Asset B is the implementation of a software system, and Asset C is test cases for this system. Asset A is the specification of variability in the system. Software product line engineering recommends maintaining a feature diagram, Asset A; however, this is not required to maintain a product line. The act of configuring and building a system can be done manually by domain experts using tooling not built on product line concepts. This was the case for all our four industrial case studies. However, making a feature diagram (and the bindings) is neither hard nor time consuming; it was done for all four case studies.

When Asset A–C are available, CIT can be carried out as follows: Configurations can be sampled from the feature diagram; Asset D shows the sampled configurations out of the swarm of possible configurations. Now, given the small number of selected configurations, it is feasible to build all their corresponding products; the configurations are applied to the implementation assets to produce a small number of actual, runnable products, Asset E, products P1–4. These runnable products can then be tested using the test cases, Asset C, to produce a test report, Asset F. Thus, with CIT, we start with the implementation and end up with a report on its quality.

The five papers, on which this thesis is based, present results that contribute to this process. Their points of contribution are marked as 1–5 respectively on Figure 1.1.

Contributions 1, 2, 4) The contributions of Paper 1 [78], 2 [79] and 4 [82] all relate to the sampling of configurations for testing purposes. The sampling stage was, prior to the contribu-

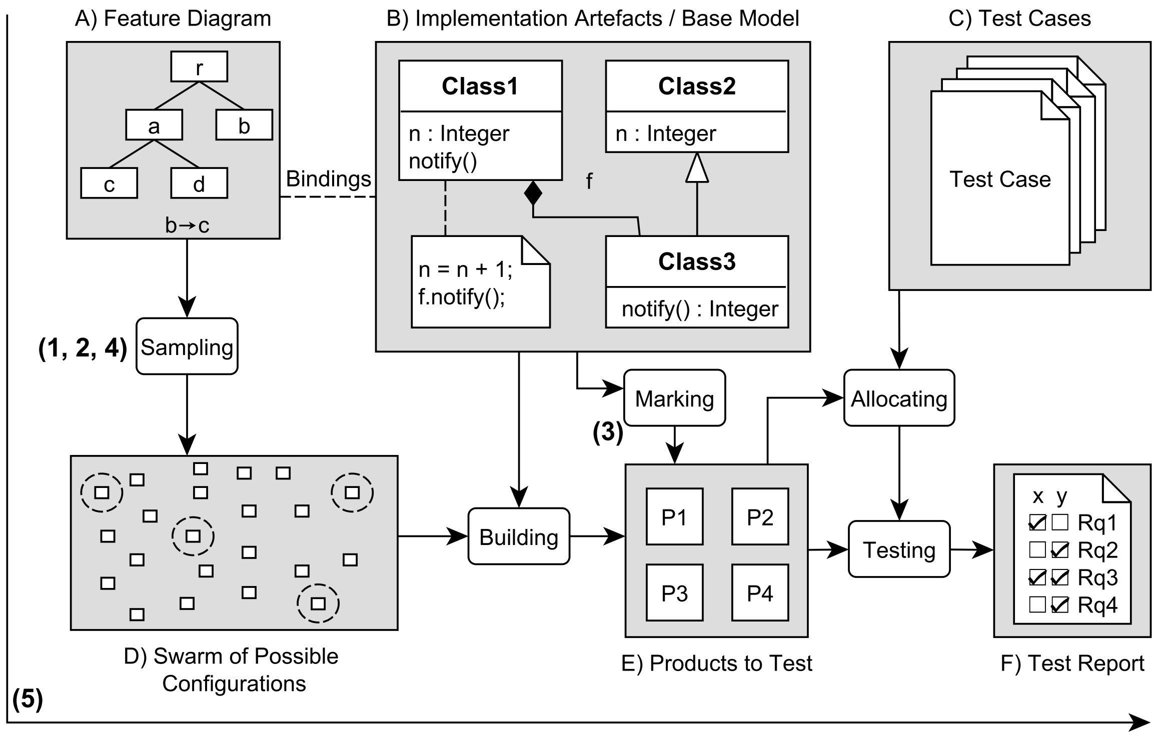


Figure 1.1: Contributions 1, 2, 3, 4 and 5 as Parts of a Whole Test Process

tions of this thesis, considered too computationally expensive to be done in industrial settings. Indeed, no scalable algorithm or tool existed. The primary contribution of Paper 1 is an argument for why this stage is feasible in practice: Providing a single valid configuration of a feature diagram is classified as NP-hard; however, having a product line without any products does not make sense in practice. Thus, it is argued that realistic feature diagrams are easily configurable, something that is backed up by a study of realistic feature models.

Paper 2 contributes a scalable algorithm (and tooling) for performing the sampling, thereby consolidating the argument of Paper 1. The contributed algorithm, ICPL, was and still is the fastest algorithm for CIT product sampling for product lines of industrial size, and the first to be able to sample products from the largest available product lines.

Paper 4 also contributes to the sampling stage. A premise of CIT is that all possible simple interactions must be exercised to evaluate the quality of a product line. However, depending on the market of the product line, not all interactions can occur, or seldomly occur. A *weighted sub-product line model*, a type of model contributed in Paper 4, allows the market situation to be captured. Associated algorithms, also contributed, then enable the product sampling to take the market situation into account to sample a smaller set of more market-relevant configurations.

Contribution 3) Often, the products of a product line need to work with different implementations of what is essentially the same thing: for example, a file system, network interface or a database system. A homogeneous abstraction layer provides a uniform interface which the product line assets can interact with instead of having one asset for each concrete implementation. There is, then, one implementation of the interface for each supported variant; for example, for each variant of file systems, network interfaces or database systems. Each implementation of the abstraction layer is mutually exclusive. One implementation of an abstraction layer suffice in these situations. This causes the number of products selected in CIT product sampling to increase because the sampling algorithm is restricted to choosing one implementation per product where it elsewhere can strategically select multiple features.

The contribution of Paper 3 [80] is to turn this problem into a benefit: Such homogeneous abstraction layers enable the reuse of existing test suites across many products selected in CIT product sampling. This step is not necessary to perform CIT; it is noted in Figure 1.1 as a marking of the configurations sampled. Those sampled configurations that only differ in the implementation of homogeneous abstraction layers are marked. These marks can then be utilized to make a voting oracle or a gold standard oracle as a part of the "Testing" stage.

Contribution 5) The contribution of Paper 5 [81] is noted in Figure 1.1 as an arrow going from the upper-left to the lower-right. It is a technique to fully automate the application of CIT: Given a feature diagram with bindings to the implementation and a set of test cases, it produces a test report in one click.

Among other things, it required an automatic allocation of the existing test cases: This was accomplished in the "Allocating" stage of Figure 1.1 by allocating the tests for the products sampled to those in which the test cases' required features were present.

A large-scale application this technique was done to the part of the Eclipse IDEs supported by the Eclipse Project. It was and still is the largest documented and reproducible application of a fully automatic testing of a software product line of industrial size.

Tool Support) All the contributions are supported by complete implementations in SPLCATool (Software Product Line Covering Array Tool) v0.3 and v0.4 and the Automatic CIT tool suite, all written by the authors of this thesis and freely available as open source. Technical details and a user manual for these tools are available as Appendix D.

1.2 Overview of the Industrial Cases

We applied parts of the contributions of this thesis to three commercial, industrial cases. In addition, we included one open source system. This was for the purpose of being able to work with the source code, publish the results and provide reproducible experiments. This open source case study will be described first followed by the three commercial, industrial cases.

1.2.1 The Eclipse IDEs

Eclipse is a collection of integrated development environments (IDEs) developed by the Eclipse Project [132]. An IDE is a program in which developers work with (primarily) software. They usually have editors especially designed to write certain programming languages. The IDE does various analyses in the background to give the users direct feedback or help when working. The IDE also keeps the system built at any time so that the developer can try the system they are developing with a single click. There are usually several editors open at any time, each specialized for their language or data format, each with help and guidance. All these tools are integrated into a complete environment for development, thereby the name *integrated development environment.*

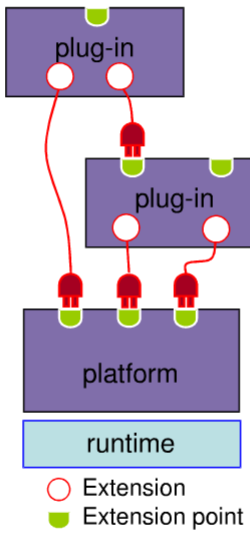


Figure 1.2: Overview of the Eclipse Plug-in System from [51]

The Eclipse IDEs are widely used, especially among Java developers. It is a competitor to Microsoft's Visual Studio which is primarily used for Windows and .NET programming in C++ and C#.

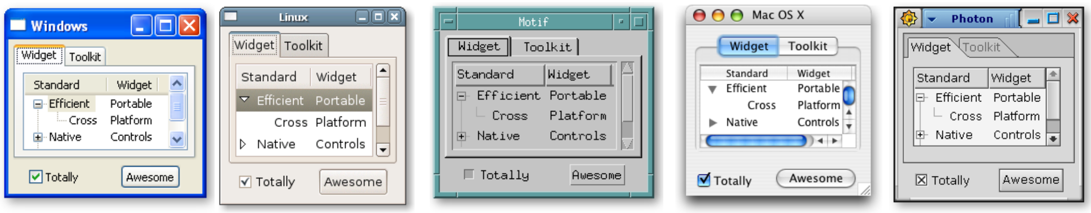
Eclipse is developed openly, is open source and free. The source code repositories and bug tracking systems used by the developers are online [51].

Eclipse is developed by, among others, Erich Gamma, the main author of the influential first book on design patterns [52]. Figure 1.2 shows Gamma's own overview of the design and how variability is supported in Eclipse. The Eclipse platform exposes extension-points that plug-ins can implement. These plug-ins can then expose new extension points, etc. Eclipse has a large community of plug-in developers.

Eclipse is also the recipient of the 2011 ACM Software System Award.

All of this made it a prime candidate as a case study for this thesis: It is open source, all results found can be published and documented, and the experiments can be reproduced by other researchers.

We studied v3.7.0 (Indigo) because it was the latest release in the summer of 2011. Twelve products were, at the release, configured by the Eclipse Project and offered for download; users were free to configure their own version of the IDE using the built-in tools of the basic platform.



(a) Windows (b) Linux, GTK (c) Linux, Motif (d) Mac (e) Mac, Photon

Figure 1.3: The Same Eclipse-based System Running in Different Windowing Systems (From `http://www.eclipse.org/swt/`, May 2011)

1.2.2 TOMRA's Reverse Vending Machines (RVMs)

TOMRA's reverse vending machines (RVMs) [156] handle the return of deposit beverage containers at retail stores such as supermarkets, convenience stores and gas stations. In Norway, were TOMRA resides, 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. In Norway, all vendors that sell beverage containers are required by law to take them back and return the money upon request.

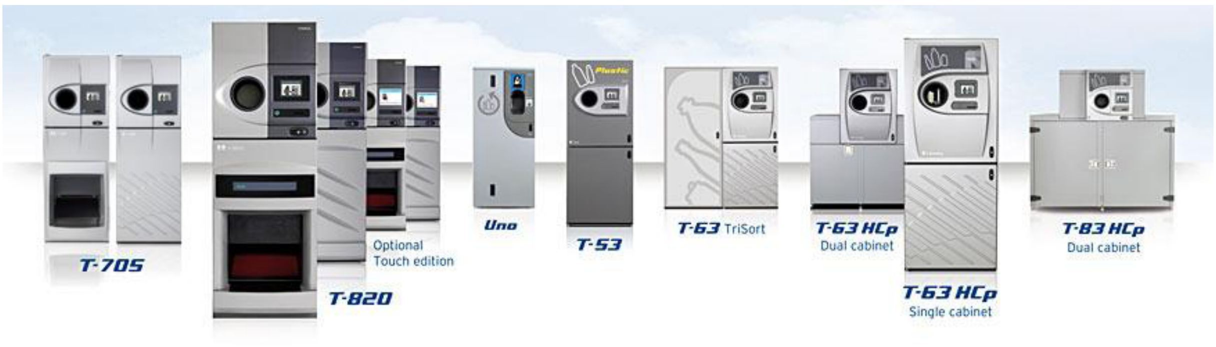


Figure 1.4: Several Variants of TOMRA RVMs

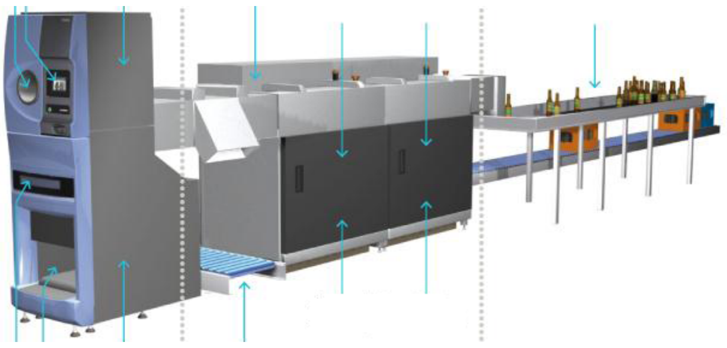


Figure 1.5: Behind the Front of TOMRA RVMs

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.

At TOMRA Verilab they are responsible for testing these machines. Their existing test setup was a set of manually configured test products. These test products were tested using manually written tests that were executed both automatically and manually.

1.2.3 ABB's Safety Modules

ABB is a large international company working with power and automation technologies. One of ABB's divisions develops a device called a *Safety Module* [1].

The ABB Safety Module is a physical component that is used in, among other things, cranes, conveyor belts and hoisting machines. Its task is to ensure safe reaction to problematic events,

Figure 1.6: The ABB Safety Module

such as the motor running too fast, or if 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 thesis—for experimenting with modeling tools and their interoperability. It is this version of the ABB Safety Module which testing is reported in this thesis.

1.2.4 Finale's Financial Reporting Systems

Finale Systems AS is a provider of software for financial reporting and tax returns. Their portfolio consists of nine main products that automate many tasks or subtasks in the reporting of accounting data to the public authorities [4]. For example, according to Finale, approximately 6,300 auditors and accountants use one of their systems to produce 130,000 annual settlements.

Their nine products defer many configuration options to users, making the number of possible configurations high. Companies use a wide range of accounting systems that Finale's systems need to interact with. They interact with these systems though a homogeneous abstraction layer. This abstraction layer is implemented concretely for each specific accounting system.

1.2.5 Additional Studies

In addition to these four case studies, 19 realistic feature models were gathered from research and industry; they are listed in Table 1.1. They include three large feature models extracted from three large systems by She et al. 2011 [141]: the Linux kernel, FreeBSD and eCos. Only the feature models of these studies were used, and they were used as is.

Four realistic feature models were made for each of our industrial case studies, but these were not included in some experiments to remain unbiased.

These 19 feature models were used in Contribution 1, the analysis of the covering array generation from realistic feature models, and in Contribution 2, the development of an algorithm for covering array generation.

1.3 Structure of the Thesis

The Faculty of Mathematics and Natural Sciences at the University of Oslo recommends that a dissertation is presented either as a monograph or as a collection of research papers. We have chosen the latter.

Table 1.1: Models and Sources

System Name	Model File Name	Source
X86 Linux kernel 2.6.28.6	2.6.28.6-icse11.dimacs	[141]
Part of FreeBSD kernel 8.0.0	freebsd-icse11.dimacs	[141]
eCos 3.0 i386pc	ecos-icse11.dimacs	[141]
e-Shop	Eshop-fm.xml	[95]
Violet, graphical model editor	Violet.m	`http://sourceforge.net/projects/violet/`
Berkeley DB	Berkeley.m	`http://www.oracle.com/us/products/database/berkeley-db/index.html`
Arcade Game Maker Pedagogical Product Line	arcade_game_pl_fm.xml	`http://www.sei.cmu.edu/productlines/ppl/`
Graph Product Line	Graph-product-line-fm.xml	[103]
Graph Product Line Nr. 4	Gg4.m	an extended version of the Graph Product line from [103]
Smart home	smart_home_fm.xml	[165]
TightVNC Remote Desktop Software	TightVNC.m	`http://www.tightvnc.com/`
AHEAD Tool Suite (ATS) Product Line	Apl.m	[157]
Fame DBMS	fame_dbms_fm.xml	`http://fame-dbms.org/`
Connector	connector_fm.xml	a tutorial [163]
Simple stack data structure	stack_fm.xml	a tutorial [163]
Simple search engine	REAL-FM-12.xml	[105]
Simple movie system	movies_app_fm.xml	[122]
Simple aircraft	aircraft_fm.xml	a tutorial [163]
Simple automobile	car_fm.xml	[166]

The thesis is based on a collection of five research papers and is structured into two main parts plus an appendix. Part I provides the context and an overall view of the work. Part II contains the collection of research papers. The purpose of Part I is to explain the overall context of the results presented in the research papers and to explain how they fit together. Part I is organized into the following chapters:

- **Chapter 1 - Introduction** introduces the problem, contributions and industrial cases.
- **Chapter 2 - Background and Related Work** presents the relevant background and other work related to the contributions of this thesis.
- **Chapter 3 - Contributions** presents an overview of the contributions and how they fit together.
- **Chapter 4 - Discussion** presents threats to validity and provides ideas for future work.
- **Chapter 5 - Conclusion** presents the conclusions of this thesis.

Each research paper in Part II is meant to be self-contained and can be read independently of the others. The papers therefore overlap to some extent with regard to explanations and definitions of the basic terminology. We recommend that the papers are read in the order they appear in the thesis.

- **Chapter 6 - Overview of Research Papers** is the first chapter of Part II. It provides publication details and a summary of each research paper.

Part III contains a series of appendices. The interested readers are referred to them throughout Part I for additional details.

Chapter 2

Background and Related Work

This chapter introduces the relevant background on product line engineering and testing, in Section 2.1 and 2.2, respectively, followed by related work: First, the related work in product line testing is presented in Section 2.3. Then, because the contributions of this thesis is based on an efficient algorithm for t-wise covering array generation (called ICPL), other algorithms for the same problem are presented, discussed and related to it in detail in Section 2.4.

2.1 Background: Product Line Engineering

A product line [125] is a collection of systems with a considerable amount of hardware and/or code in common. The primary motivation for structuring 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 and/or code. It is common for customers to have conflicting requirements. In that case, it is not even possible to ship one system for all customers.

The Eclipse products [132] can be seen as a software product line. When Eclipse v3.7.0 was released, the Eclipse website listed 12 products on their download page[1]. 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.

2.1.1 Feature Models

One way to model the commonalities and differences in a product line is using a feature model [86]. 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 2.1 is an example of a feature model for a subset of Eclipse. 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

[1] `http://eclipse.org/downloads/`

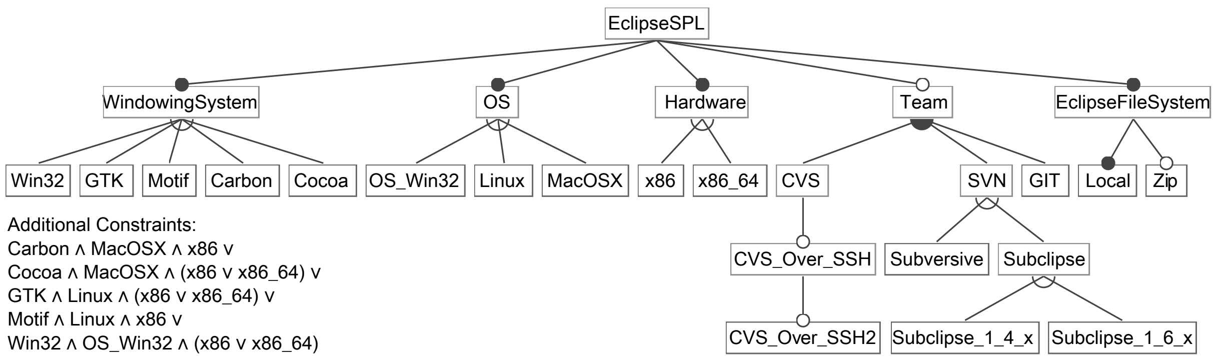


Figure 2.1: Feature model for a subset of Eclipse

on the edges going from a node to one or more nodes. For example, in Figure 2.1, one has to choose one windowing system which one wants Eclipse to run under. This is modeled as an empty semi-circle on the outgoing edges. When choosing a team functionality provider, at least one or all can be chosen. This is modeled as a filled semi-circle. The team functionality itself is marked with an empty circle. This means that that feature is optional. A filled circle means that the feature is mandatory. The feature model is configured from the root, and a feature can only be included when the preceding feature is included. For example, supporting CVS over SSH requires that one has CVS.

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 might be called a *variant* or simply a *product* and is specified by a configuration of the feature model. Such a configuration consists of specifying whether each feature is included or not.

Basic Feature Models

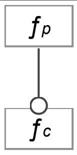
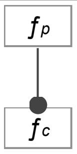
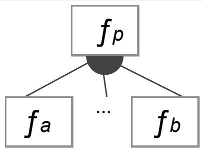
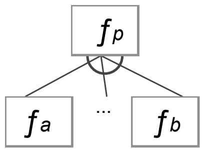
The contributions of this thesis use basic feature models. There are other kinds of feature models with more complex mechanisms. The work presented in this thesis can still apply meaningfully to more complex feature modeling mechanisms, but then a basic feature model must first be extracted for the purpose of testing.

Basic feature models are widely known and were originally introduced in Kang 1990 [86]. Batory 2005 [11] showed that a basic feature model can easily be converted to a propositional formula, and this is precisely the formalism of feature modeling that will be used in this thesis.

A basic feature model [139] has of a set of n features, $\{r, f_2, f_3, ..., f_n\}$. The first feature is a special feature called the *root feature*; it must always be included in a valid configuration. The other features may or may not be included depending on what the feature model specifies. The means of specifying variability in a basic feature model is listed in Table 2.1. The first column shows the most popular feature diagram notation, described in the second column. The last column shows the respective semantics as a propositional constraint. The P in the last row is for writing a propositional constraint below the tree structure. The primitives of this propositional constraint must be the feature names in the tree.

To convert a feature diagram into a propositional constraint, iterate through the diagram and convert each part into the corresponding propositional constraint. Combine them with $\wedge$. What are valid solutions of the feature diagram are also valid solutions to the resulting propositional constraint and *vice versa*.

Table 2.1: Overview of Basic Feature Models [139]

Diagram	Description	Semantics
	r is the root	r
	f_c is an optional sub-feature of f_p	$f_c \Rightarrow f_p$
	f_c is a mandatory sub-feature of f_p	$f_c \Leftrightarrow f_p$
	$f_a, \ldots, f_b$ are "or" sub-features of f_p	$(f_a \vee \cdots \vee f_b) \Leftrightarrow f_p$
	$f_a, \ldots, f_b$ are alternative sub-features of f_p	$((f_a \vee \cdots \vee f_b) \Leftrightarrow f_p) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$
P	Propositional formula P	P

2.1.2 Approaches to Product Line Engineering

Several approaches are in use or have been proposed for dealing with and working with the variability of a product line.

Preprocessors

Preprocessors can be used for product line engineering. When a feature model has been configured, the included and excluded features can be mapped to symbols that are issued to, for example, the C preprocessor. It will then conditionally compile source code according to how these symbols are set.

For example, if a feature called X is mapped to a symbol $XFeature$ issued to the build system, the code inside the ifdefs gets included in the product:

```
...
#ifdef XFeature
f();
#endif
...
```

This is done in all systems that use the *kbuild* build system [127]. kbuild is used for many open source product lines [13]. It takes its name from the Linux Kernel which it is used to configure and build (the Kernel Build system); the Linux Kernel is one of the largest product lines openly available.

Feature-Oriented Programming

Feature-oriented programming (FOP) [7] is the general idea that a product can be built by adding separately developed features to a base system. For example, with object-oriented programming, a feature can be developed as a new sub-class which refines a class by adding new fields, overriding existing methods or adding new methods. These are ways of refining a base program in a step-wise manner [10] to include more and more features, which may be defined as increments in functionality.

Components, Frameworks and Plug-ins

A product line can be developed by bundling features as components or plug-ins that are installed into a framework. Feature names can be mapped to fully qualified component names, as is done in the Eclipse Plug-in System [132]. To derive a product from a configuration of a feature model, start with the minimum set of required components (or the base framework) and then simply install the components whose corresponding feature is set to included in the feature model. Thus, you end up with the product that corresponds with the configuration.

When a product line is developed in this way, the components are units which may be executed by another program, such as a unit test. Such components can be tested as is, and no generation or configuration is necessarily needed in order to exercise a part of the product line.

Orthogonal Variability Modeling (OVM)

Where product line approaches like preprocessors or component-based approaches require the embedding of variability in the system implementation itself, various approaches propose having the variability modeled orthogonally to the implementation.

An approach for implementing variability orthogonally is the *orthogonal variability model* (OVM) of Pohl et al. 2005 [125]. Figure 2.2 shows a feature diagram in OVM syntax. A feature model in the popular syntax that captures the basic meaning of this is shown in Figure 2.3. In OVM, *variation points* are differentiated from *variants*. A variation point (tagged VP) is something that can vary, and a variant (tagged V) is what it can vary as. These tags are not included in Figure 2.3. Figure 2.3 also have two nodes without names. This is because these

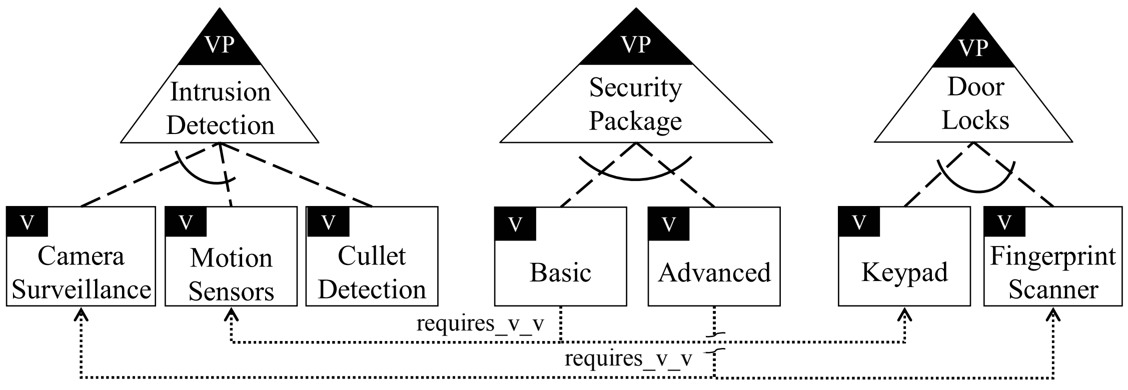


Figure 2.2: Orthogonal Variability Model — From [125]

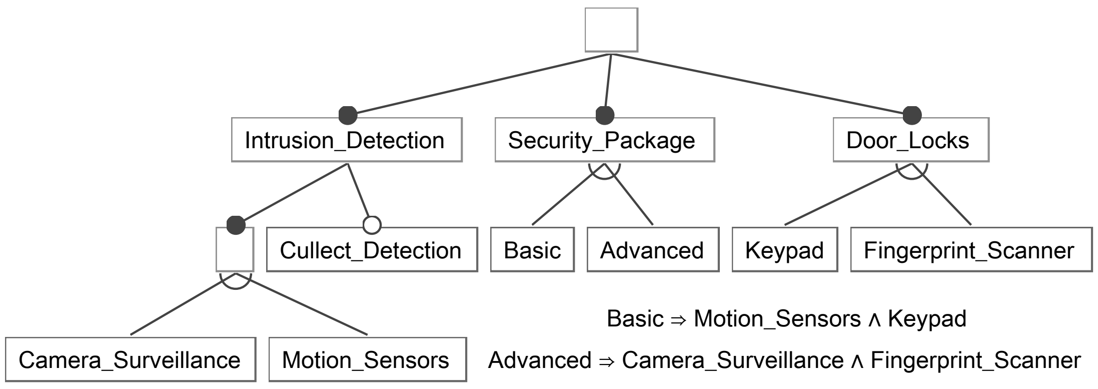


Figure 2.3: FODA Syntax Equivalent of Figure 2.2

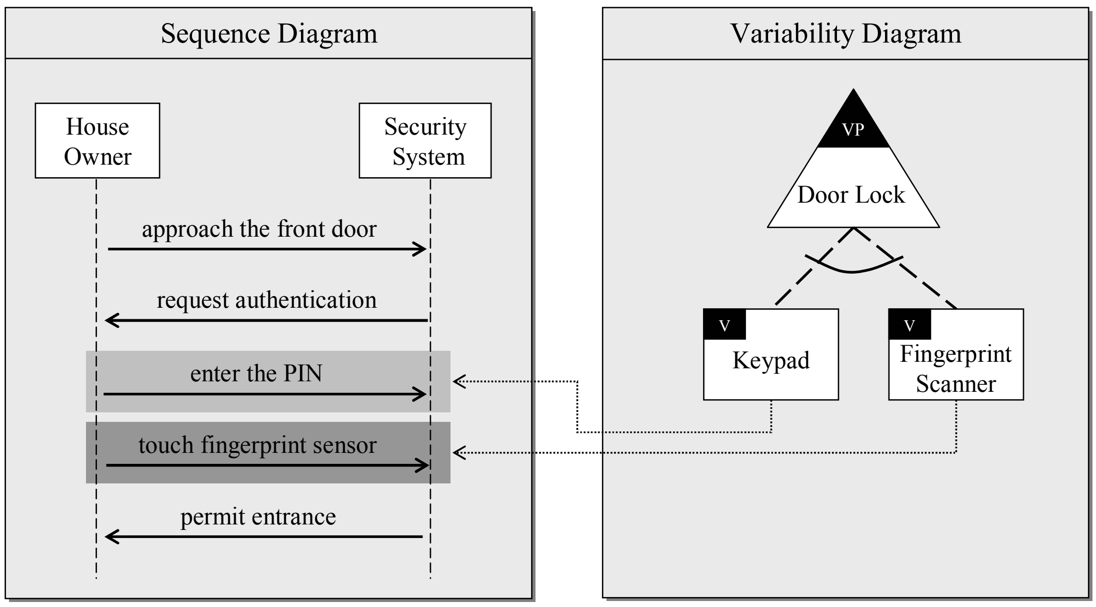


Figure 2.4: Orthogonal Variability Model and the Bindings to the Model — From [125]

nodes are unnecessary in OVM syntax, and therefore do not need to be named. Figure 2.3 also has textual propositional constraints instead of the arrows used in OVM.

Figure 2.4 shows how OVM can be used to model variability. The sequence diagram on the left is a part of the implementation or the test model of a system. The variants on the right side are linked to a segment of the model. This is what makes OVM orthogonal, in that the variation is modeled separately and not as a part of the implementation or test model. As for product realization, if the keypad is chosen, for example, the "enter pin"-transition remains in the sequence diagram, and, as the fingerprint scanner is therefore not included, the "touch fingerprint sensor"-transition is removed.

Common Variability Language (CVL)

OVM covered the general idea of an orthogonal approach, but it did not specify how this is to be achieved technically. One suggestion was provided by the *Common Variability Language* (CVL). It was initially proposed in Haugen et al. 2008 [69], and its most recent version is documented in OMG 2012 [71] as a part of the standardization process of CVL being carried out by OMG. Figure 2.5 shows the same feature diagram as in Figure 2.2 and Figure 2.3, but this time in CVL syntax.

A feature diagram is in CVL called a *Variability Specification Tree* (VSpec tree). Instead of the semi-circle in the popular syntax, it uses cardinalities on a triangle placed under a feature. The top-node in CVL is an instance of a *Configurable Unit* (a concept not further discussed here). It is unnamed in this example. The constraints are written in the textual *Basic Constraint Language* (BCL), a proposed constraint language as a part of CVL.

CVL realizes the idea proposed in OVM and shown in Figure 2.4 by requiring models to be implemented according to MOF [111] stored as XMI [112]. Specifically, implementations of CVL target Ecore, the Eclipse version of the Essential MOF (EMOF) standard. CVL supports defining *fragments* of models that are instances of an Ecore meta-model.

For example, Eclipse has defined an Ecore meta-model of UML2. We might use this to make a UML2 model of the sequence diagram shown in Figure 2.4. We can create fragments of the two highlighted areas around the transitions. We can then make a *fragment substitution* that substitutes the transition with an empty fragment when one of the features is not included. Such fragment substitutions are linked with the features in a CVL feature model.

Figure 2.6 shows how fragments are modeled and how fragment substitutions are performed in CVL. A fragment is modeled by recording its boundary elements. Then, any other fragment with matching boundary elements can replace it. In Figure 2.6, the fragment in the lower-left diagram is replaced by the fragment in the upper-left diagram. The result of the substitution is the model in the right diagram.

Because version 1 of CVL was used in one of the contributions of this thesis, its diagram notation will be briefly explained. Figure 2.7 shows part of a feature diagram in CVL 1 notation. In this diagram, the gray squares are the features. The text in the top part of each feature is the feature name. The lower part contains a list of the contained placement fragment (prefixed with a red square), replacement fragments (prefixed with a blue square with red fragments inside) and constraints (prefixed with a star symbol). For example, the constraint in the `Level3` feature is an implication. The red squares below some of the features are the associated fragment substitutions.

CVL 1 allows the modeling of optional features using a yellow oval with "0..1" indicating that zero or one of the following feature can be chosen. Features can be marked as alternative

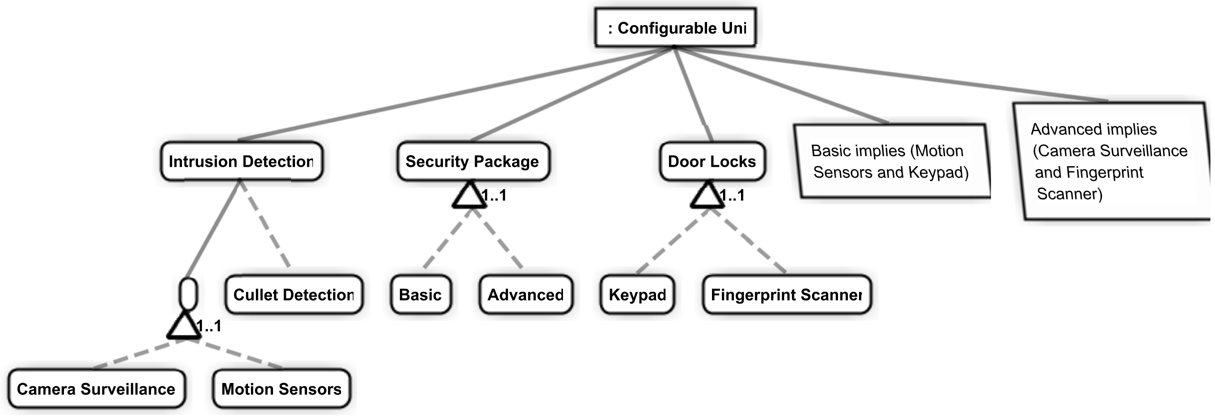


Figure 2.5: CVL VSpec Tree — Yields the Same Configurations as Figure 2.2 and 2.3

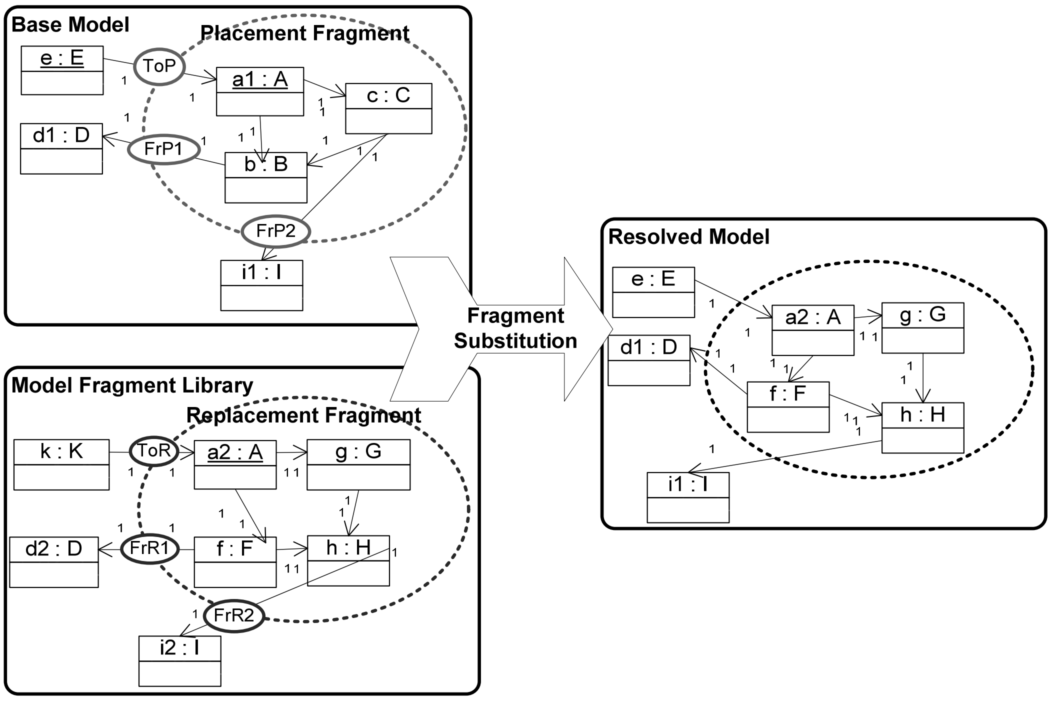


Figure 2.6: Fragment Substitution — From the CVL 1.2 User Guide [151]

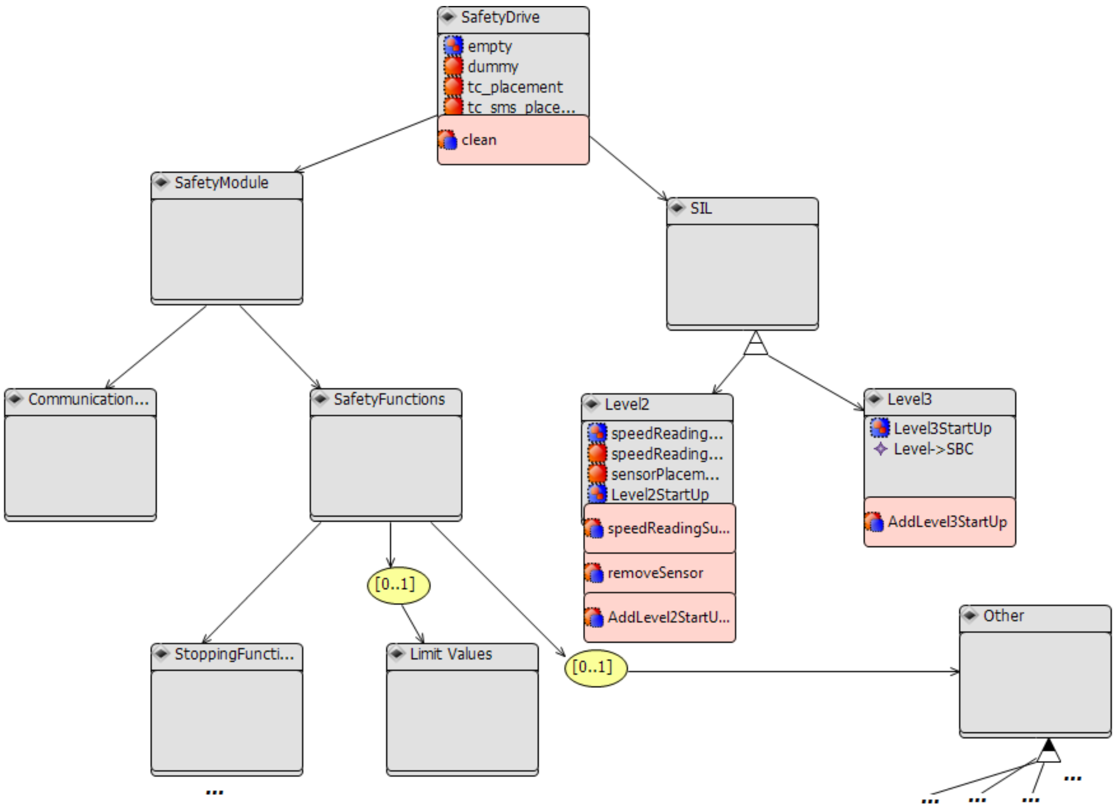


Figure 2.7: Part of Feature Diagram of the ABB Safety Module in CVL 1 Feature Diagram Notation

using an empty triangle below the parent feature. The partly filled triangle indicates that one or more features must be included. (The ellipsis are not a part of CVL; they indicate where the diagram has been cut.)

Delta-Oriented Programming

Delta-Oriented Programming (DOP) [134, 135] is a textual language for writing feature modules. It is also orthogonal to the implementation. Figure 2.8a shows an example core program of an SPL with two features `Print` and `Lit`.

They are implemented in Java within the core. Figure 2.8b shows two feature modules, `Eval` and `Add`. When executed against the core in Figure 2.8a, these deltas will add their features to the core. For example, executing `DEval` will add the feature `Eval` by adding the method `eval` to the interface `Exp` and an implementation to the class implementing it `Lit`.

In general, deltas can be constructed as shown in Figure 2.8c. It is possible to remove, add and modify classes and interfaces. Both methods and fields can be added, removed and renamed.

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
        int value;
        Lit(int n) { value=n; }
        void print() { System.out.print(value); }
   }
}
```

(a) Example Core Modules

```
delta DEval when Eval {
   modifies interface Exp { adds int eval(); }
   modifies class Lit {
      adds int eval() { return value; }
   }
}
```

```
delta DAdd when Add {
   adds class Add implements Exp {
      Exp expr1;  Exp expr2;
      Add(Exp a, Exp b) { expr1=a; expr2=b; }
   }
}
```

(b) Example Delta Modules

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

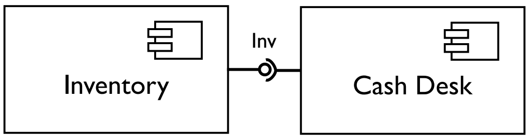
(c) General Structure of Deltas in DOP

Figure 2.8: Example of Delta-oriented Programming – From [134]

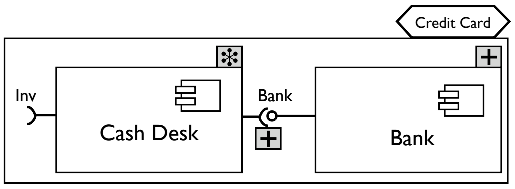
Delta Modeling

Delta Modeling [33,64,133,136] is a formal modeling approach for feature oriented programing (FOP). It is also orthogonal to the implementation.

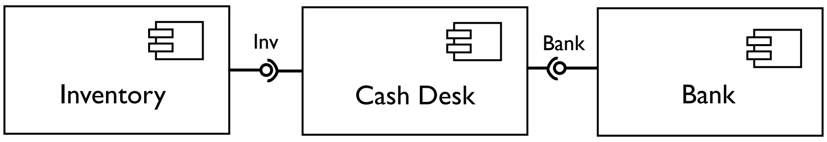
Figure 2.9a shows an example core model. Figure 2.9b shows an example Delta Model (Δ-model) that when applied to the core model in Figure 2.9a yields the model in Figure 2.9c.



(a) Example Core Model



(b) Example Delta Model (Δ-Model)



(c) Resulting Model From Applying the Delta Model in Figure 2.9b to the Core Model in Figure 2.9a

Figure 2.9: Example of Delta Modeling – From [133]

In the delta model in Figure 2.9b we can see that three model elements have been annotated with a symbol with a gray background. These show what these elements will do when run against a base model. The asterisk (*) means modification; the plus (+) means addition. This explains why in Figure 2.9c `Bank` got added, the `Cash Desk` component got modified by adding an arrow to `Bank`.

Delta modeling also supports removal of model elements.

One important difference between CVL and Delta Modeling is that CVL deals with model fragments by capturing the boundaries around fragments. Delta modeling, on the other hand, adds, removes or modifies individual elements.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [88] can be used for product line engineering [164]. Aspect-oriented programming was introduced based on the observation that certain concerns cross-cuts implementations. A typical example is logging. Logging will typically be spread around most of the source code instead of being isolated in a single component. It would be better if logging was separated out, so that it could be considered and worked with as an isolated unit instead of being scattered around in all the other units.

Aspects solve this problem as follows. By using pattern-matching techniques, logging calls could, for example, be added in the beginning of all method definitions. An aspect is a construct that, for example, requests all methods to do a call to a logging function at its beginning.

The compilation process for aspect-oriented programs involves a *weaving* step. This is, for example, to add the logging calls, or to weave them, into other functions.

Similarly, an aspect could be used to make features for an existing program. The new features could be programmed as an aspect to be woven into the core program on inclusion. The weaving will then result in the new feature becoming a part of the program. Aspects are, for this reason, also orthogonal to the implementation.

2.2 Background: Testing

The computer is fast, reliable and tireless in performing its tasks. Automatic testing is the idea that the computer should not only run programs to solve certain tasks quickly; it should also be used to run programs and check the answers given for correctness.

Automatic testing is in principle simple. We have a program P that can be invoked with input I_x to produce output O_x. An automatic test is simply a second program Q that invokes P with input I_x and verifies that the output O_x given by the program is correct either by a direct match, or that it fulfills some criteria.

2.2.1 The Test Process

Figure 2.10 shows the general process that is followed when using testing for quality assurance of a system. First, tests are created somehow. They are then executed against the system. The oracle verifies that the resulting behavior of the system is correct. If some of the tests fail, then the faults must be located and fixed. Further tests might be needed to help localizing the faults. Eventually, all tests will pass. It should then be asked whether there are sufficient tests to meet quality requirements. This can be expressed as a coverage criterion of some kind. If it is not met, further tests must be created to meet it. Otherwise, the testing process finishes.

The basic way to do testing is to write the tests manually, select input manually and select when the output is correct or not manually.

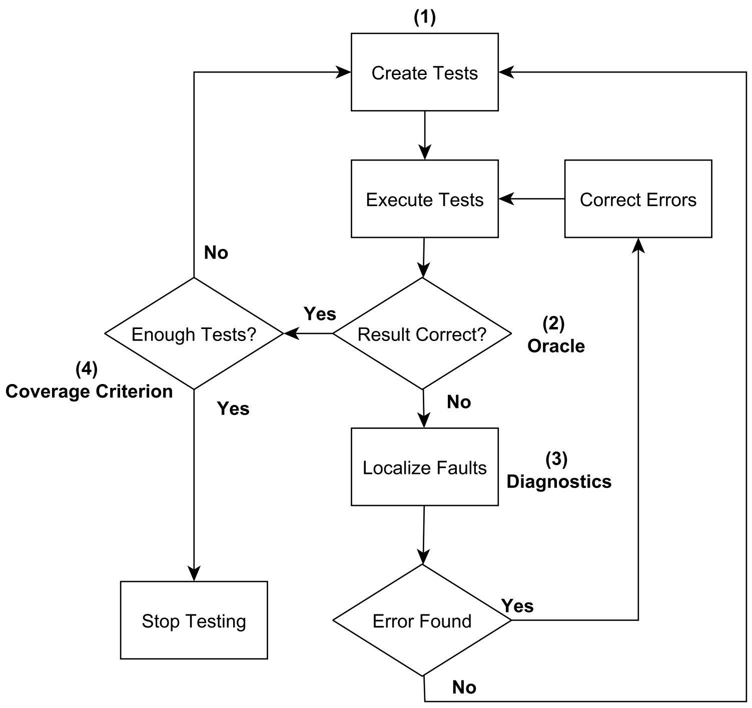


Figure 2.10: The Testing Process

2.2.2 V-Model

There are various testing activities that can be done during system development. These are usually modeled in the V-Model [83], Figure 2.11. Development chronologically proceeds

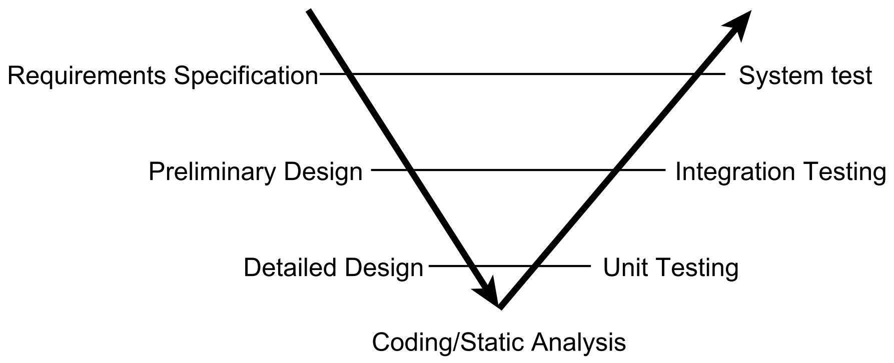


Figure 2.11: The V-Model

along the arrows. (Note that with agile development practices, the V-model is done as a whole in iterations.) First *requirements are specified* for some functionality. These requirements can be tested with *system tests*. Testing that the system behaves according to the requirements is the end goal. The requirements are used as the basis of *preliminary designs* that are then refined into *detailed designs*. Corresponding to these two stages are *integration testing* and *unit testing*. They are coarse grained and fine grained testing, respectively. Finally, the functionality is *coded*. During coding, *static analysis* should be used to check for errors during coding. After the functionality has been coded, the system can be unit tested, integration tested and, finally, tested using the system tests.

With test driven development (TDD) [12], tests are written before coding.

2.2.3 Unit Testing

One widely used form of testing is *unit testing*. Unit testing is to isolate units of a program, be it classes or methods, and then invoke them without any other unit being invoked.

Often, units have dependencies. These dependencies are then *mocked*. Mocking is to create a new class or method that behaves as one would expect them to in the situation set up in the unit test. Thus, when a failure occurs, it can be attributed to a fault in the unit, because nothing but the unit was invoked.

2.2.4 Model-based Testing

Model-based testing in general is testing a system by manually creating a test model and then generating tests from this model according to some criterion [159]. These tests are then executed against the actual system.

For example, say a software system has been implemented in a textual language such as C. Say the system is reactive; it receives and sends messages through an interface. Say it implements the controller software for a turnstile. A turnstile is a barrier typically used in public transportation systems: A customer is allowed past the barrier when he or she shows a ticket. In this example, however, the customer pays directly to the machine to pass.

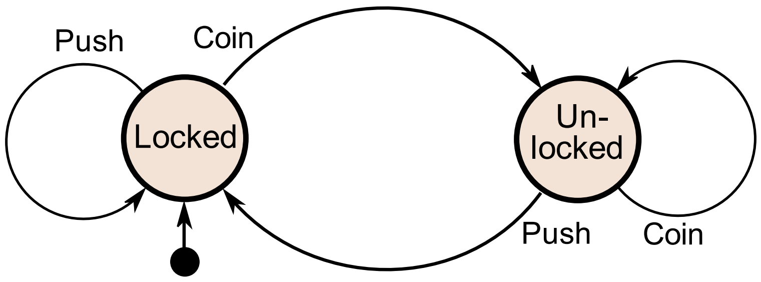


Figure 2.12: Test Model Example — State Machine of Turnstile Controller (Image from `https://en.wikipedia.org/wiki/File:Turnstile_state_machine_colored.svg`)

For such a system, a test model might be implemented as a state machine. Figure 2.12 shows a state machine for the high-level behavior of the turnstile. Initially, it is locked. When the customer puts a coin into it, it unlocks. The customer walks through by pushing the barrier. When the customer has passed, the barrier locks to be used by the next customer.

Various coverage criteria can be defined for a test model. In general, the effort required for exhaustive testing increases exponentially with the system complexity. Thus, exhaustive testing is infeasible in general. A coverage criterion allows generating fewer tests that somehow covers some aspect of the test model.

For example, for the test model in Figure 2.12, one of the simpler coverage criteria is the *all-states criterion*. It is fulfilled by visiting all states at least once. For example, it is fulfilled by putting a coin into the machine and pushing the barrier to verify that it is unlocked.

A more complex criterion is the *all-transitions criterion*. It is fulfilled by performing all transitions at least once: for example, by inserting a coin, then inserting another coin, pushing the barrier and, finally, pushing the barrier again. This sequence will visit all transitions of Figure 2.12.

Various other coverage criteria exists in the model-based testing literature [159].

2.2.5 Regression Testing

For testing in general, reuse is employed to minimize testing effort. For single system testing, when a system is changed, old test cases and test results can be reused in order to minimize testing of the new system. This is known as *Regression Testing* [108]. IEEE 1990 [74] defines regression testing as

> "Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or components still complies with its specified requirements."

The basic challenge of regression testing is how to reuse the old test cases and test results. Some tests can be reused directly, but some need to be modified. Determining which need to be modified is one challenge; automatically modifying them is another challenge.

2.2.6 Category Partition (CP)

Category Partition (CP) was introduced as a general-purpose, black-box test design technique [18] by Ostrand et al. 1988 [121]. It is a technique meant to be done manually; it is unsuitable for (full) automation because it involves considerations of the intended functionality of the system under test.

CP is for designing a test suite for a method m. Step 1 is to manually identify the functions $(f_1, ..., f_n)$ implemented by the method m. Step 2 is to manually identify all the input and output parameters $(i_1, ..., i_p; o_1, ..., o_q)$ of each function. For example, the input parameters of a method on an object typically include the object itself; thus, the object must be in the list of input parameters.

It is on these parameters the *categories* (from the name, *category* partition) are defined manually. This is Step 3. A category is a subset of the parameter values that cause a particular behavior in the output. The notion of categories is similar to the notion of *equivalence classes* [108].

It is these categories that are manually *partitioned* (again from the name, category *partition*) into choices of values. This is Step 4. Step 5 is to identify constraints between the choices. This will rule out some of the choices.

Step 6 is to enumerate all choice combinations. The cross-product will serve this purpose, but might yield a lot of combinations. Finally, Step 7 is to define the expected output values for each combination of input values either manually or using an oracle.

Having completed these steps results in a test suite that exercises the method m.

2.3 Related Work: Product Line Testing

The related work for product line testing is covered in three parts. Section 2.3.1 contains an overview of product line testing techniques classified by approach. The following six subsections contain descriptions of six approaches for product line testing: PLUTO, ScenTED, CIT, CADeT, MoSo-PoLiTe and Incremental Testing. Finally, Section 2.4 contains a detailed description of three algorithms for covering array generation (IPOG, MoSo-PoLiTe and CASA) with a detailed comparison with ICPL, an algorithm contributed in this thesis. This is followed by a short discussion of eight other algorithms for covering array generation (and then 13 further algorithms are listed but not discussed).

2.3.1 Overview

Product line testing approaches can roughly be divided into four categories: 1) approaches not utilizing product line assets, 2) model-based techniques, 3) reuse techniques based on regression testing and 4) subset heuristics.

Reusable component testing (RCT) is a simple and widely used technique for product line testing. It does not fit into the four categories and does not test feature interactions. Thus, reusable component testing and feature interactions are discussed before the product line testing techniques that utilize product line assets in order to test the product lines including feature interactions.

Contra-SPL-philosophies

Pohl et al. 2005 [125] discuss two techniques for testing product lines that do not utilize product line assets. These are the Brute Force Strategy (BFS) and the Pure Application Strategy (PAS).

The Brute Force Strategy (BFS) lives up to its name. The strategy is merely to produce all products of a product line and then test each of them. This strategy is infeasible for any product line but the very smallest. This is because the number of products generally grows exponentially with the number of features in a product line.

The Pure Application Strategy (PAS) is simply to not test anything before a certain product is requested. When a product is requested, this product is built and tested from scratch. Although this strategy is easy to use and feasible, it does not provide validation of products before they are requested. It would be good to somehow utilize the product line assets in order to ensure to some extent that the products work when they are requested and built. This is the basic concern of the product line testing technique discussed below.

Reusable Component Testing

Reusable Component Testing (RCT) can be used when a product line is built out of components that are composed into products. These reused components can be tested in isolation. If they function correctly, we have more confidence they will not fail because of an internal fault. The main drawback of this technique is, of course, that it does not test feature interactions.

Reusable component testing is actively used in industry [77]. It is a simple technique that scales well, and that can easily be applied today without any major training or special software. As it exercises commonalities in a product line it will find some of the errors early.

In our Johansen et al. [77], we noted that several companies reported having partly used reusable component testing in experience reports: Dialect Solutions reported having used it to test a product line of Internet payment gateway infrastructure products [144]. Testo AG reported having used it to test their product line of portable measurement devices for industry and emission business [53]. Philips Medical Systems reported having used it to test a product line of imaging equipment used to support medical diagnosis and intervention [140]. Ganesan et al. 2012 [54] 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.

Feature Interaction

As just mentioned, testing one feature in isolation, as is the case for, for example, reusable component testing, does not test whether the feature interacts correctly with other features. Two features are said to interact if their influence each other in some way [101].

Kästner et al. 2009 [87] addressed the *optional feature problem*, a problem related to feature interactions. They mention an example where two features are mutually optional, but where one is implemented differently depending on whether the other is included. One example they consider is a database system with the feature ACID and the feature STATISTIC. The latter feature will include the statistic "committed transactions per second" that only makes sense if the feature ACID is included because it facilitates transactions.

They note that feature interactions can cause unexpected behavior for certain combinations of features. An often mentioned example in telecommunications is the two features *call waiting* and *call forwarding*. When both are active, which is to take an incoming call?

Batory et al. 2011 [8] proposed that whenever two features f and g are included in a product, not only are they composed, but so is their interaction (denoted $f\#g$). In other words, including two features is not only $f \cdot g$ ($\cdot$ being the composition operator) but $(f\#g) \cdot g \cdot f$.

How frequent are feature interactions? n features may be combined in $O(2^n)$ ways; however, according to Liu et al. 2006 [101] experience suggests that interactions among features are sparse. That is, according to experience, most features do not interact. In other words, $(f\#g)$ is $\emptyset$ for most f and g. The experience is from, among other sources, the telecommunication system industry. Kästner et al. 2009 [87] agrees with Liu et al. in that there usually are more feature interactions than features.

There are techniques proposed in literature that may be able to test feature interactions effectively. Three of these are model-based techniques, reuse-based techniques and subset-heuristics. They are discussed in the following subsections.

They are combined with each other and with other techniques in attempts to efficiently test product lines. Some of these techniques are discussed in Section 2.3.2 onwards.

Model-based SPL Testing

For model-based product line testing, many different models have been proposed used.

Bertolino and Gnesi 2003 [15], with their PLUTO method for product line testing, propose modeling the use cases of the product line with extended type of Cockburn's use cases [34], a textual format for writing use cases. They suggest extending them with variability information. They call this new type of use case *Product Line Use Case*, or PLUC.

Several methods propose modeling product line behavior using UML activity diagrams annotated with variability information. Olimpiew 2008 [113] used them as a part of the CADeT method; Reuys et al. 2003 [130] used them as a part of the ScenTED method together with UML interaction diagrams and UML use case diagrams, variability was modeled with the Orthogonal Variability Model (OVM) [125]; and Hartmann et al. 2004 [67] proposed using UML activity diagrams as a part of their method. They annotate parts of the activity diagrams with product names, and not with features. Thus, their method is limited to situations where all the products are known up front.

State machines have been used for a long time in software engineering. Several propose using them for modeling product line behavior. Cichos et al. 2011 [32] proposed creating a single state machine containing all behaviors of all products, called a *150% model*. When instantiated according to a configuration, the 150% model yields a *100% model*, the behavior of one product.

Lochau et al. 2012 and Lity et al. 2012 [100, 102] also proposed modeling the behavior as a state machine, but instead of modeling the union of all product behaviors, they suggested capturing the difference between pairs of state machines using delta modeling. Oster et al. 2011b [120] used state machine models to build a 150% model as a part of their MoSo-PoLiTe method.

Svendsen et al. 2011 [148, 149] modeled train stations using the domain specific language

Table 2.2: Model-based Techniques .
[1] Product selection is done after test selection and not before.
[2] Products are selected using CIT before testing.
[3] Products are selected with any technique (or selected for delivery) before they are tested.
[4] *Model Coverage* means any coverage criterion relevant for the test model might be employed.
[5] As long as a basic feature model can be extracted for testing purposes

Technique	Test Model	Variability Mechanism	PL Test Generation
PLUTO	Cockburn's Use-Cases	Annotations	CP[1]
ScenTED	Activity Diagrams	OVM	Model Coverage[1,4]
CADeT	Activity Diagrams	PLUS	CIT[2] + Model Coverage[4]
Svendsen et al.	DSLs, Alloy	CVL	Any[3] + BMC and Manual
MoSo-PoLiTe	State Machines	pure::variants	CIT[2] + Model Coverage[4]
Lochau et al.	State Machines	Delta Modeling	Any[3] + Model Coverage[4]
This thesis	n/a	Any[5]	CIT[2] + Manual

Train Control Language (TCL) [150]. They used the Common Variability Language (CVL) [69] to model the difference between two or more train stations. They used Alloy [75] to model the semantics of TCL and CVL.

Based on the respective models, each technique has an associated test generation method. As a part of their PLUTO method, Bertolino and Gnesi 2004 [16] proposed using the Category Partition (CP) method [121] (explained in Section 2.3.2). For the CADeT method, Olimpiew 2008 [113] used combinatorial selection techniques (explained in Section 2.3.4) to select products to test. For the ScenTED method, Stricker et al. 2010 [147] proposed an enhanced, specialized test generation algorithms for ScenTED to further avoid redundancies in testing.

Table 2.2 shows six model-based testing techniques. The last row of the table shows the technique proposed in this thesis. Both PLUTO and ScenTED selects products to test after having designed the test cases. CADeT and MoSo-PoLiTe uses CIT. The work by Svendsen et al. and Lochau et al. on incremental testing supports any selection of products, either manually or by CIT or other methods. They then propose using incremental testing to minimize the effort on testing these selected products. Svendsen et al. use bounded model checking (BMC) with Alloy. Lochau et al. propose new algorithms based on Delta Modeling to minimize testing. They propose generating test cases using coverage criteria for state machines.

The technique proposed in this thesis is not a model-based technique. It can use any variability mechanism as long at a basic feature model can be extracted for testing purposes. From this basic feature model, the products are selected automatically using covering array generation from CIT. The test cases are proposed made manually.

A recent survey of model-based testing techniques for product lines is provided by Oster et al. 2010b [114].

Reuse

Regression testing techniques can be adapted for product lines. Because any two products p_a, p_b of a product line are similar, p_b can be seen as a development of p_a. If we then test p_a, we can optimize the testing of p_b using regression testing techniques.

This is product line testing because when delivering product p_n, the testing of p_1 to p_{n-1} will minimize the effort needed to testing p_n.

Engström et al. 2010 [50] surveyed the use of regression testing technique for product line testing.

Uzuncaova et al. 2008 and 2010 [160, 161] proposed using regression testing techniques to minimize test generation time for the next product.

Svendsen et al. 2011a [148] proposed using regression testing techniques to determine whether a test needs to be re-executed for the next product. Svendsen et al. 2011b [149] proposed using regression testing techniques to minimize the time to verify a property of the next product.

Lochau et al. 2012 and Lity et al. 2012 [100, 102] proposed using regression testing techniques to minimize the number of test cases needed to fulfill a model-based coverage criterion for the next product. Dukaczewski et al. 2013 [49] proposed using regression testing techniques on annotated textual requirements, as they are more common than test models.

Subset-Heuristics

A product line has surely been thoroughly tested if all products are executed for all possible inputs. The idea of subset-heuristics is to select a subset of the products or of the inputs such that the testing remains (almost) as good as testing all products with all inputs. Some of the challenges of subset heuristics are 1) being able to generate a small subset, 2) being able to generate the subset within a reasonable time and 3) knowing how good a subset selection technique is.

Combinatorial Interaction Testing (CIT) can be used to both generate a subset of products and generate a subset of inputs for testing. The technique has been known for a long time, and is discussed and developed extensively. Cohen et al. 1994 [35] is one of the earlier papers that introduced an algorithm for generating subsets. The related work will be discussed in depth later. CIT handles constraints between features in a product line and between input variables for system testing.

Kuhn et al. 2004 [94] investigated the bug reports for several large systems. They found that most bugs can be reproduced by setting a combination of a few parameters (and no more than six). Thus, they indicate that most bugs can be detected by exercising any combination of a few parameters.

Generating a subset that covers all simple combinations is classified as NP-hard. Algorithms finding non-optimal subsets still keep advancing. Some of these algorithms are discussed later. A contribution of this thesis is an argument for why the subset selection of products of a product line is quick in practice. Another contribution is an algorithm that is currently the fastest algorithm for selecting a subset of products for testing. Both of these contributions relate to product subset selection for testing, and only indirectly to program input subset selection.

There are subset heuristic techniques that are not based on CIT. Kim et al. 2011 [89] use static analysis to determine a reduced set of products that are relevant for a test. This reduction lessens the combinations of products that need to be tested given a certain test. Kolb 2003 [92] proposed using risk analysis to identify a subset of products to test. Scheidemann 2008 [137], proposed a technique that selects products such that requirements are covered.

Oster et al. 2009 [118] proposed a method where feature models (FMs) and classification trees (CT) are combined to a Feature Model for Testing (FMT). Classification trees are used in the CT-method [62] for software testing. The FMT model can be used with classification tree tools such as CTE [3] to generate a subset of system tests that also exercise the product line.

2.3.2 Product Line Use case Test Optimization (PLUTO)

The *Product Line Use case Test Optimization PLUTO* is "a test methodology for product families" [14]. The PLUTO method was first introduced by Bertolino and Gnesi 2003 [15] and later in Bertolino and Gnesi 2004 [16] and Bertolino et al. 2006 [14].

PLUTO, a model-based technique, uses Cockburn's use cases [34] extended with variability information. These enhanced use cases are called *Product Line Use Cases*, or PLUCs. Cockburn's use cases are represented in a textual notation with natural language. Variability information is added as textual annotations.

A modified version of the category partition method (CP) is applied to the product line use cases (PLUCs) in order to design a product line test suite. Before explaining how this is done, product line use cases are explained.

Product Line Use Cases (PLUCs)

Figure 2.13 shows a product line use case (PLUC). There are two main sections: before and after "PL Variability Features", both written in natural language. Above is a use case. This use case has been annotated with variability information that is defined below. This particular use case is for the call answer feature of a mobile phone, written on the first line. The goal in this use case is to answer an incoming call on a mobile phone. The goal is annotated with "[CA0]". Below, we can see that there are three alternatives for this annotation, Model 1–3.

When this and the other annotations are specified, we get a *Product Use Cases* (PUCs) [14]. This must be done manually. The "[CA0]" annotation can be directly replaced by one of the model names, making for example "Goal: answer an incoming call on a Model 1 mobile phone".

Step 2 of the main success scenario is less simple. If the selected phone model is Model 1, then a value for "[CA2]" must also be specified. If, say, "a" is chosen, then "[CA1]" becomes "Procedure B". Thus, Step 2 of the main success scenario becomes "2. The system establishes the connection by following Procedure B."

The PLUC in Figure 2.13 yields 5 different PUCs: for (CA0, CA1, CA2), we can have (0, A, a), (1, B, a), (1, C, b), (2, B, a) and (2, C, b). Not all of these necessarily need to be instantiated for testing. This depends on the application of category partition to PLUCs.

Using CP on PLUCs

PLUTO involves using CP on PLUCs to produce a product line test suite. CP is first applied normally to the use case. Then, after having selected categories and choices, the products corresponding to the choices made are selected. It might be the case that each test gets its unique product, or it might be the case that all tests are run on a single product. This depends on what is in the use case. One consequence of this is that no more products are needed than there are test cases. In PLUTO, product selection is a consequence of applying the CP method.

```
PLUC CallAnswer
Goal: answer an incoming call on a [CA0] mobile phone
Scope: the [CA0] mobile phone
Precondition:
    Signal is available.
    Mobile phone is switched on.
Trigger: incoming call
Primary actor: the user
Secondary actors: the {[CA0] mobile phone} (the system)
                  the mobile phone company

Main success scenario:
    1. The user accepts the call by pressing the button "Accept".
    2. The system establishes the connection by following the {[CA1]
       appropriate} procedure.

Extensions:
    1a. The call is not accepted:
        1a.1. The user presses the button "Reject".
        1a.2. Scenario terminates.

PL Variability Features:
    CA0: Alternative:
        0. Model 0
        1. Model 1 [CA2]
        2. Model 2 [CA2]

    CA1: Parametric:
        case CA0 of:
            0: Procedure A:
                2.1 Connect caller and callee.
            1 or 2:
                if CA2 = a then Procedure B:
                    2.1 Interrupt the game.
                    2.2 Connect caller and callee.
                if CA2 = b then Procedure C:
                    2.1 Save current game status.
                    2.2 Interrupt the current game.
                    2.3 Connect caller and callee.

    CA2: Alternative:
        a. Games are available; if interrupted, status is not saved.
        b. Games are available; if interrupted, status is saved.
```

Figure 2.13: Example of Product Line Use Case (PLUC) (taken from [14] and fixed)

For example, ten test cases of the use case in Figure 2.13 might only need three different products out of the five possibilities: e.g. (0, A, a), (1, B, a), and (2, C, b). This is likely, because all the options of all three alternatives are present in one of the three. If an eleventh test case runs through 1 and b (for the second and third option respectively) then, of course, a product with those is needed also, making it four.

2.3.3 Scenario-based Test Case Derivation (ScenTED)

Scen*ario based* **TE***st case* **D***erivation* (ScenTED) was first introduced in Kamsties et al. 2003b [84]. It is “a solution for applying product line concepts to the testing process by providing detailed guidelines on how to create generic test artifacts in domain engineering and how to reuse these generic artifacts in application engineering” [131].

The idea of ScenTED is to create test assets in UML which can be reused after deriving products, also modeled in UML with variability specified using an orthogonal variability model (OVM). In order to achieve this, they start by creating UML use case scenarios—descriptions of how to use the software—that includes variation points (Figure 2.14, Step A). These are UML activity diagrams with variability information as annotated branches. From these they extract test-case scenarios—a selection of paths through the activity diagrams just described (Figure 2.14, Step B). These are modeled as UML interaction diagrams with variability information as annotated messages. They suggest a branch coverage criterion, modified for variability, to select good test cases from the activity diagram. These test-case scenarios get their variability resolved to become tests for products (Figure 2.14, Step C).

Figure 2.15 shows the information model of ScenTED. Here we find the use case and test-case scenarios described above. In addition there are the architecture scenarios and executable test cases. The former are scenarios with interactions between components—and not just the user and the system as is the case for use case scenarios. The latter are the test-case scenarios including concrete data for the input and output values of the tests.

The arrows between the boxes in Figure 2.15 are traceability links. Traceability in this context means that you can find your way to, for example, the use case from a particular use case scenario. The system is, as can be seen from the figure, traceable all around.

A thorough presentation of ScenTED can be found in [131] which is also the primary ref-

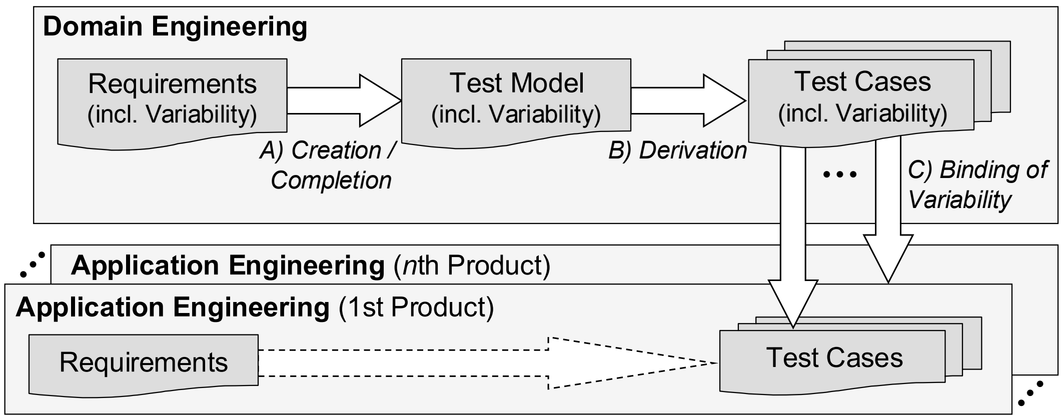

Domain Engineering
Requirements (incl. Variability)
Test Model (incl. Variability)
Test Cases (incl. Variability)
A) Creation / Completion
B) Derivation
C) Binding of Variability
Application Engineering (nth Product)
Application Engineering (1st Product)
Requirements
Test Cases

Figure 2.14: Main Activities and Artifacts of ScenTED – taken from [107]

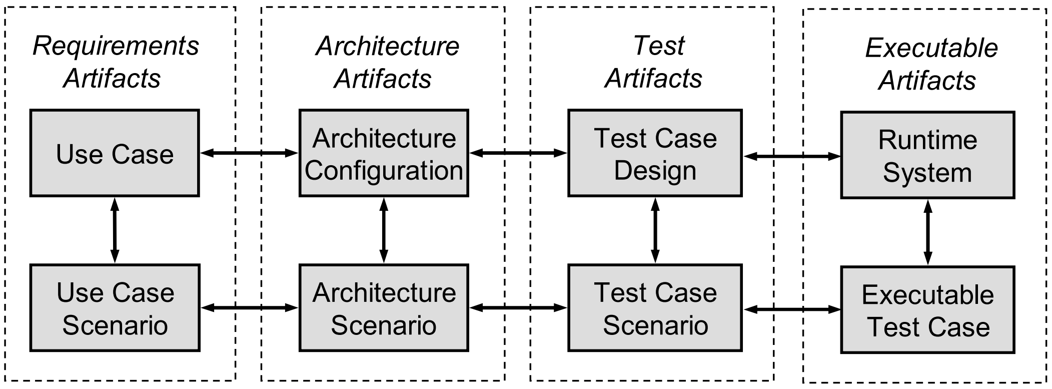


Figure 2.15: ScenTED Information Model – taken from [131]

erence of this presentation of ScenTED. The ScenTED method was, prior to the primary reference, presented in [107], [129], [85] and [84]. The activity diagrams extended with variability were presented earlier in [130].

Pohl and Metzger 2006 [126] is a short paper that discusses three challenges with SPL testing and presents six principles which can be used to solve these challenges. The solutions presented are compared against the ScenTED method where most of the principles are reported to be addressed by ScenTED.

Reis et al. 2006 [128] presents an extension of ScenTED for performance testing called ScenTED-PT. They explain how test assets for performance testing are derived and reused in their extended method.

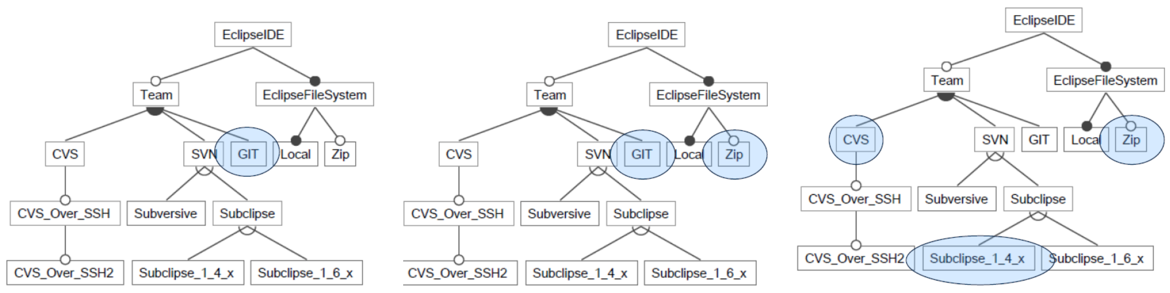
Stricker et al. 2010 [147] presents another extension of ScenTED called ScenTED-DF for avoiding redundant testing.

2.3.4 Combinatorial Interaction Testing (CIT)

Combinatorial interaction testing [42] is an approach for performing interaction testing between the features in a product line. The approach deals directly with a basic 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 (see for example [18] or [83].)

The idea is to select a small set of products where the interaction faults are most likely to occur. An important motivation for CIT is a paper by Kuhn et al. 2004 [94]. They investigated a number of highly configurable industrial systems, and found that most of the bugs reported can be attributed to a specific assignment of a few parameters. This means that by adjusting just a few parameters, most bugs can be found. These empirical investigations apply to configuration options also.

Figure 2.16a shows a feature model. If there is a fault in `GIT` that can be detected in any product that contains this one feature, the fault is called a *1-wise fault*. In Figure 2.16b, however, there is a fault in the interaction between `GIT` and `Zip` that can be detected in any product that include these two features. This *2-wise fault* is not a fault in either feature, but a fault in the interaction between them. They do not interact correctly. Surely, if they never interact either directly or indirectly, there can be no such fault, but in this case they can: The `Zip` feature allows browsing a zip-file and editing its internals. The `GIT` feature is a source control system.



(a) 1-wise fault (b) 2-wise fault (c) 3-wise fault

Figure 2.16: Examples of Feature Combinations Required to Reproduce a Fault

When the internals of a zip-file is changed, the `GIT` feature must mark it as changed and as a candidate for a commit. Not doing this is an example of a 2-wise interaction fault between these features. Figure 2.16c shows an example of combination of features triggering a *3-wise fault*. Similar arguments presented for 1-wise and 2-wise faults apply to 3-wise and higher types of *t-wise faults*.

CIT for product lines starts by selecting a few products where all valid combinations of a few features are present at least once. 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 2.3 shows the 22 products that must be tested to ensure that every *pair-wise* interaction between the features in a sub-set of the Eclipse IDE, Figure 2.1, functions correctly. Each row represents one feature, and every column represents 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 because they are invalid according to the feature model.

Testing every pair is called *2-wise testing*, or pair-wise testing. This is a special case of *t-wise testing* where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in one product, 3-wise coverage means that every combination of three features are present, etc. For the Eclipse IDE example, 5, 22, 64 and 150 products are sufficient to achieve 1-wise, 2-wise, 3-wise and 4-wise coverage, respectively.

Kuhn et al. 2004 [94] 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. Thus, 3-wise testing is an important milestone for testing non-critical product lines. Figure 2.17 shows these numbers are plotted in Kuhn et al. 2004 [94]. The *FTFI Number* is the number of test-parameters that had to be given a specific value to reproduce a bug. The y-axis shows the percentage of bugs that could be attributed to such an FTFI-number.

There is even more empirical support for CIT. Garvin and Cohen 2011 [57] did an exploratory study on two open source product lines. They extracted 28 faults that could be analyzed, and which were 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.

Table 2.3: Pair-wise coverage of the feature model in Figure 2.1 the test suites numbered

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
EclipseSPL	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WindowingSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
GTK	-	X	-	-	-	-	X	-	-	X	-	-	-	X	-	-	X	-	-	-	-	-
Motif	-	-	-	-	X	-	-	X	-	-	-	X	-	-	-	-	-	-	X	-	-	-
Carbon	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	X	-	X	-	-	-	-
Cocoa	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	X	-
OS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OS_Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
Linux	-	X	-	-	X	-	X	X	-	X	-	X	-	X	-	-	X	-	X	-	-	-
MacOSX	X	-	-	X	-	-	-	-	-	-	X	-	X	-	-	X	-	X	-	X	X	-
Hardware	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
x86	X	-	-	X	X	-	-	X	-	X	X	X	-	-	X	X	-	X	X	-	-	-
x86_64	-	X	X	-	-	X	X	-	X	-	-	-	X	X	-	-	X	-	-	X	X	X
Team	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X
CVS_Over_SSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
CVS_Over_SSH2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X
SVN	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X	X
Subversive	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	-	X
Subclipse	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	X	-	X	X	X	X	-
Subclipse_1_4_x	-	-	-	-	-	-	-	X	X	X	-	-	-	-	-	X	-	-	-	-	X	-
Subclipse_1_6_x	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	X	X	X	-	-
GIT	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	X	X	-	X	X	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	-	-	X	X	-	-	-	X	-	-	-	-	X	-	-	-	-	-	X	-	X

These observations strengthen the case for combinatorial interaction testing.

Steffens et al. 2012 [146] 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.

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 subset of products must be generated from the feature model for some coverage strength. Such a subset is called a t-wise covering array for a coverage strength t. Lastly, a single system testing technique must be selected and applied to each product in this covering array.

Covering Arrays

The set of products that must be tested in combinatorial interaction testing is called a covering array, but what precisely are covering arrays? A formal description should clarify.

An assignment a is a pair $(f, included)$, where f is a feature of a feature model FM, and $included$ is a Boolean specifying whether f is included or not. A configuration, C, is a set of assignments where all features in the feature model, FM, have been given an assignment.

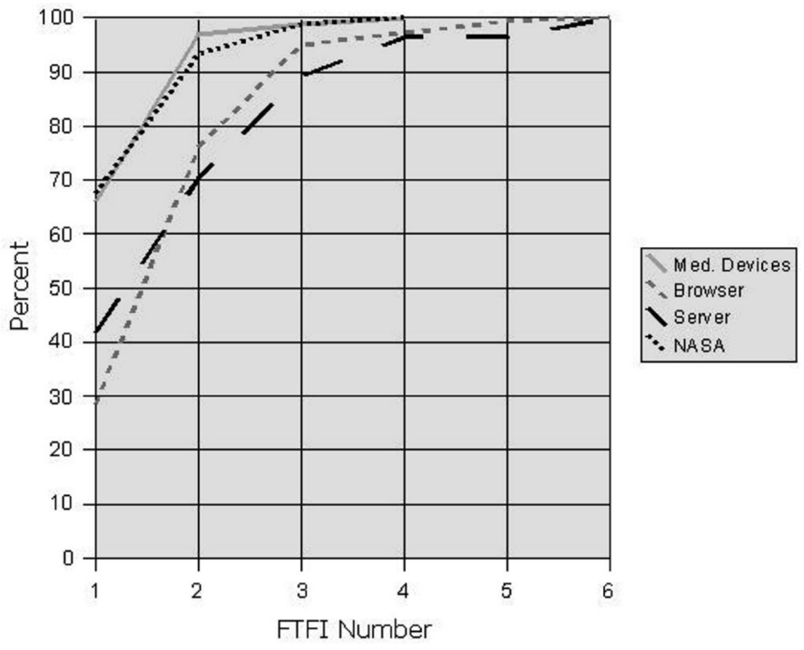


Figure 2.17: Faults Attributed to Interactions - taken from [94]

A *t-set* is a set of t assignments, $\{a_1, a_2, ...\}$. The set of all t-sets of a feature model, FM, is denoted T_t, e.g. $T_1, T_2, T_3, ...$ The set of invalid t-sets is denoted I_t, e.g. $I_1, I_2, I_3, ...$ The set of valid t-sets is thus $U_t = T_t \setminus I_t$. A covering array of strength t is a set of configurations, C_t, in which all valid t-sets, all t-sets in U_t, are a subset of (or the same set as) at least one of the configurations.

The generation of covering arrays is a topic of many publications. Nie and Leung 2011 [109] is a recent survey of combinatorial testing. They state that their survey is the first complete and systematic survey on this topic. They found 50 papers on the generation of covering arrays.

A detailed comparison between three leading algorithms and the algorithm contributed in this thesis (ICPL) is provided in Section 2.4.

2.3.5 Customizable Activity Diagrams, Decision Tables, and Test specifications (CADeT)

The **C***ustomizable* **A***ctivity Diagrams,* **De***cision* **T***ables, and Test Specifications* (CADeT) method was introduced by Olimpiew in [113] and is a "test design method that can be used to create functional test specifications for SPLs".

There are four phases of CADeT: Phase I is to create activity diagrams from use cases. The use cases have already been made as PLUS requirement models, introduced by Gomaa et al 2005 [60]. Phase II is to create decision tables (as described by Binder 1999 [18]) and test specifications from these activity diagrams. Phase III is to define and apply a feature-based coverage criteria; CADeT proposes to use a combinatorial selection technique as used in combinatorial interaction testing. Finally, Phase IV is to derive tests based on the decision table and the test specifications.

2.3.6 Model-driven Software Product Line Testing (MoSo-PoLiTe)

Mo*del-driven* **So***ftware* **Pro***duct* **Li***ne* **Te***sting* (MoSo-PoLiTe) was introduced by Oster et al. 2009 [118] and further enhanced and developed later, including Oster et al. 2010 [117], Oster et al. 2010b [114], Perrouin et al. 2011 [123], Oster et al. 2011 [119] and Oster et al. 2011b [120].

Oster 2012 [115] describes the state of the art of MoSo-PoLiTe as of 2012. An overview of it can be seen in Figure 2.18. MoSo-PoLiTe is a model-based testing technique. It uses combinatorial techniques to subset the products of a product line. These are then tested with tests generated from a test-model. The method starts with a feature model and a test model. The feature model is modeled using pure::variants (developed by pure::systems), and is seen center-top in Figure 2.18. The test model is a union of behavior of all products, known as a 150% model. This behavior is modeled using Rational Rhapsody (developed by IBM) shown in the top-right.

The variability modeled in the feature model is bound to the 150% model using pure::variants

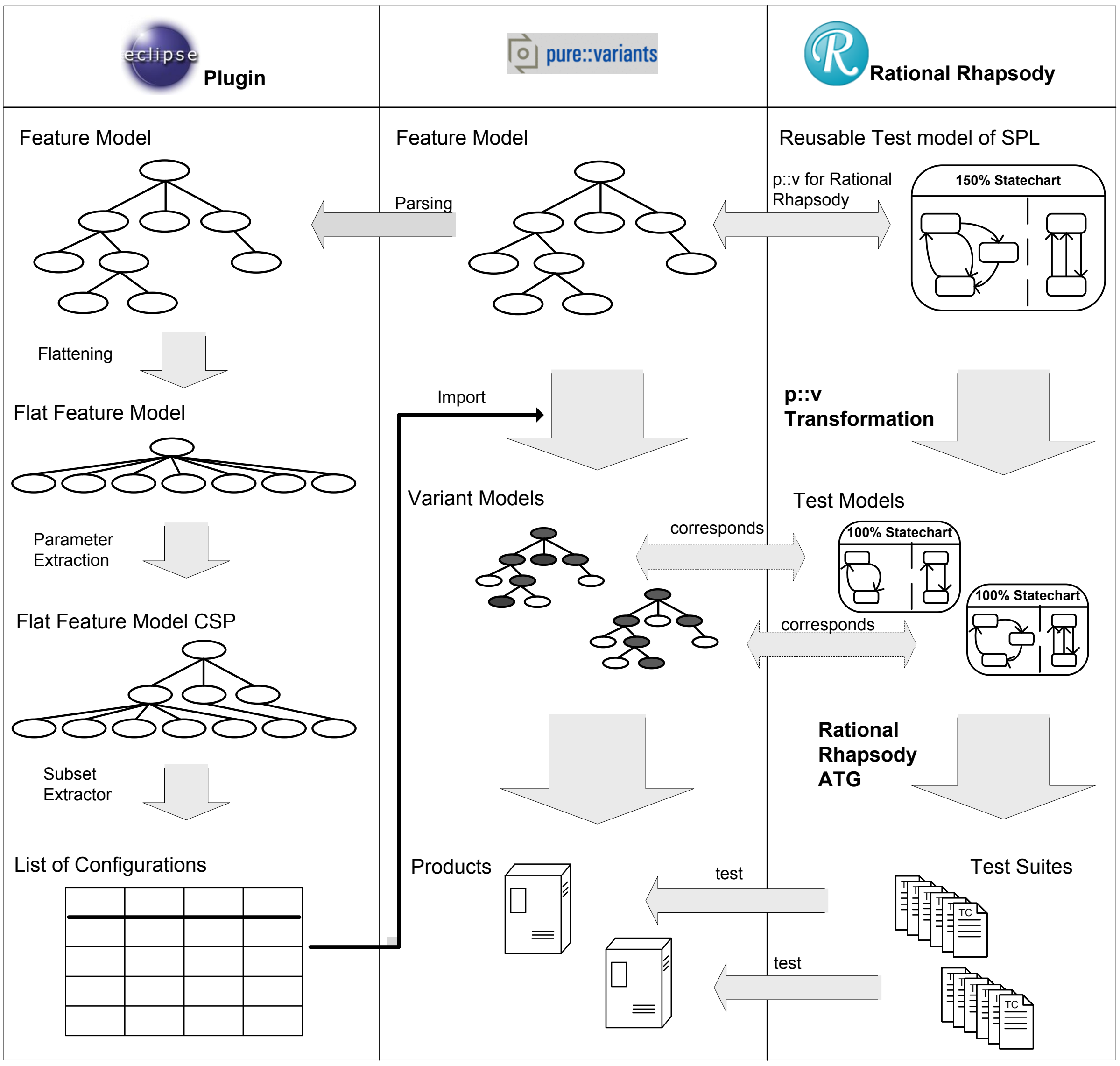


Figure 2.18: MoSo-PoLiTe Overview – taken from [115]

integration with Rational Rhapsody. This is shown with the two-way arrow between the feature model and the 150% model.

First, a subset of products for testing is generated. This is shown as the left side. First, the pure::variants feature model is parsed into an Eclipse Plug-in tool. Then, the MoSo-PoLiTe covering array generation algorithm is applied. This algorithm is covered and discussed in Section 2.4.2. This results in a list of configurations, shown in the lower-left.

These configurations are then imported back into pure::variants as feature model configurations. This is shown as a black arrow going from the configurations and up into the center. This results in the variant models. They can now be used to instantiate product test models, or 100% model. This is shown on the right side. There are bindings between these product models and the configurations.

Then the tests need to be generated. They are generated using the Rational Rhapsody Automatic Test Generation (ATG). These test suites can then be executed on the products built from the configurations, shown in the lower-center.

2.3.7 Incremental Testing

Say an organization provides products of a product line. When a new product is requested, this product is thoroughly tested from scratch. Because this new product, being a product of the product line, is similar to the previously tested products, it should be possible to significantly reduce the effort of testing this new product. *Incremental testing for product lines* proposes to reduce the effort significantly.

Incremental testing for product lines was first presented in Uzuncaova et al. 2008 [161] based on Alloy [75] and AHEAD [9]. Later, Svendsen et al. 2011a [148] proposed basing incremental testing on CVL [69], and Lochau et al. 2012 and Lity et al. 2012 [100, 102] proposed basing incremental testing on delta modeling [33].

Figure 2.19, from the original paper by Uzuncaova et al. 2008 [161], depicts the basic idea of incremental testing for product lines.

A company that organizes and offers products using a product line strategy wants to ensure that all the utilized products are of high quality. The way this is proposed done with incremental testing is as follows. s_0 is the formal specification of a product p_0. This product, which can be any product of the product line, is referred to as the base product or the core product.

Even though any product can in principle be chosen as the base product, some have noted possible benefits of strategically selecting it. For example, Svendsen et al. 2010 [152] sug-

$$\begin{tikzcd}
\boxed{s_0} \arrow[rr, "\Delta_s"] \arrow[d, dotted, "\tau"'] & {} \arrow[d, dashed, "\tau'"] & \boxed{s_3} \arrow[d, "\tau"] \\
\boxed{t_0} \arrow[rr, dotted, "\Delta_t"'] & {} & \boxed{t_3}
\end{tikzcd}$$

Figure 2.19: Incremental Testing as Described by [161]

gests selecting the base product such that the difference between it and any other product is minimized.

A test case t_0 can be generated using test generator τ from the specification s_0 of the product p_0. Another product, say p_3, has a different specification s_3. From s_3 the test generator τ generates the test t_3.

Now, we can see p_3 as a development from p_0. Thus, we have a relationship between product line testing and *regression testing* [2]. Regression testing aims at minimizing the effort needed to retest a product that has changed.

We can capture the change between the specification s_0 and s_3 as Δ_s and the change between test t_0 and t_3 as Δ_t.

Uzuncaova et al. 2008 and 2010 [160,161] proposed an algorithm τ' that allows following a different path. Instead of generating the test t_3 from s_3 using the test generator τ, they propose generating Δ_t from Δ_s using the algorithm τ'. Then, by applying Δ_t to t_0 we get t_3. They notice that this path takes less computational time than generating the test t_3 from s_3 using τ.

Uzuncaova et al. 2008 and 2010 [160, 161] used Alloy [75] to model the specifications s_i and AHEAD [9] for modeling the differences between the specifications, Δ_s, and to produce the new specification based on the difference.

Svendsen et al. 2011a [148] addressed a different aspect of incremental testing. Their case study was a product line of train stations. Train stations have rails, of course, and a lighting system automatically organized by a software system. ABB, being the company in charge of developing this software, tests each train station thoroughly from scratch.

Svendsen et al. proposed using incremental testing to lessen the effort of testing a second, third, etc. train station by inferring which test cases needed to be rerun given that a previous, similar train station had already been thoroughly tested. They had a DSL called the Train Control Language (TCL) [150] in which the train stations were modeled. Instead of using AHEAD as Uzuncaova et al., Svendsen et al. used the Common Variability Language (CVL) [69] to capture the difference between two or more train stations.

As Uzuncaova et al., Svendsen et al. also used Alloy to model the semantics; in their case, the semantics of TCL and CVL. They showed for their case study (to a limited extent) which functionality was affected by a change from one station to another. Thus, if the first train station had been thoroughly tested, they could tell (to a limited extent) which tests did and did not need to be re-executed. The change of functionality was termed the *semantic ripple effects* of a change of a product, in this case, a train station.

Svendsen et al. 2011b [149] addressed yet another different aspect of incremental testing. If a property p of a train station ts_1 has been established to always hold for ts_1, based on the change from ts_1 to another product, say ts_2, modeled in CVL, they showed (to a limited extent) how to determine if that property also holds for the second product, ts_2.

Lochau et al. 2012 and Lity et al. 2012 [100,102] introduced incremental testing of product lines based on delta modeling [33]. They address a different aspect of incremental testing than Uzuncaova et al. and Svendsen et al. did.

Say an organization maintains a product line. They have one base product p that they have tested with a test suite ts containing test cases $\{tc_1, tc_2, ..., tc_n\}$. The test suite is generated from the test model tm of the product p based on a coverage criterion C. The coverage criterion C is captured as a set of test goals tg.

Because the base product was tested with test suite ts, the test plan tp for the core product p simply is ts. This might not be the case for a second, third, etc. product of the product line.

When a second product needs testing, say product p', the difference between the test assets of p and p', called ta and ta' respectively, is captured as a tuple of deltas as follows:

$$\delta_{ta,ta'} = \delta_{tm,tm'}, \delta_{tg,tg'}, \delta_{ts,ts'}, \delta_{tp,tp'}$$

When constructing the test plan tp' for the second product p', the fact that the first product was tested with tp can be taken into account to minimize tp'.

Lochau et al. 2012 and Lity et al. 2012 [100, 102] proposed modeling the test model as a state machine and the test cases as paths in this state machine. Thus, the test goals are elements of the state machine and the test suite is the set of all test cases satisfying these goals.

The deltas are proposed created using delta modeling [33, 135]. The test assets are all constructed out of elements of a state machine; thus, the deltas are additions and removals of these elements.

Based on the test models and the deltas, they did a case study where test cases were not transferred from tp to tp' if they had automatically determined that the test case was obsolete or if the test case tested something that had not been impacted by the change in the test model. Of course, test cases that is relevant for p' that were irrelevant for p is added to tp'.

The case study compared the size of the test plans to an application of a combinatorial testing technique. For the combinatorial testing technique, the relevant tests were run for each product. The case study showed that the average number of test cases for combinatorial testing was 64, while that number was reduced to 19 using their incremental technique, 10 being new and 9 being reused.

Dukaczewski et al. 2013 [49] observed that, in industry, formal test models rarely exist; thus, the incremental technique proposed by Lochau et al. and Lity et al. is not straight-forward to apply in industry. Dukaczewski et al. propose to apply the techniques on the requirements instead. First, the requirements and the test cases, written in natural language, are manually modeled as formal constructs and related to each other. Then, delta modeling can be applied to these formal models in order to reduce the testing efforts.

2.4 Related Work: Algorithms for Covering Array Generation

Three state of the art algorithms for covering array generation are IPOG, CASA and MoSoPoLiTe. In this section, these three algorithms will be described in detail and compared with each other and to the contribution of this thesis.

These three algorithms were chosen because they fulfill criteria that are needed to perform the extensive, automated, reproducible comparison done in Contribution 2 of this thesis:

1. it must be freely available
2. it must allow programmatic invocation
3. it must at least support 2 and 3-wise covering array generation
4. it was the latest development of that algorithm

5. it must have some support for constraints

Other algorithms that fulfilled these criteria but that were not included in our detailed comparison were ATGT [23], Microsoft PICT [46] and Grieskamp's algorithm [61].

Comparisons between these algorithms and CASA exist in the literature. Thus, it is possible to do an implicit, indirect comparison. ATGT is compared to PICT and an earlier version of CASA in [26]. A comparison of Grieskamp's algorithm with PICT and an earlier version of CASA are included in [61].

2.4.1 IPOG

In Contribution 2 of this thesis, the tool *NIST ACTS version 2 beta, revision 1.5* (acquired 2012-01-04) was compared to *SPLCATool v0.3 (SPLC 2012)* with respect to covering array generation.

The version of NIST ACTS we had was closed source, so it is unknown exactly how the algorithm is implemented in the tool. We ran the tool by selecting the option named "ipog", which, according to the documentation of the tool [5], was the only option with support for constraints. It implements some version of the IPOG algorithm. The *In Parametric Order General* (IPOG) algorithm [96, 97] is a development of the *In Parametric Order* (IPO) [98] algorithm. In addition to selecting "ipog", we set the *optimization level* to 5 and the *fast mode* to "on".

Note that there are several algorithms implemented in NIST ACTS and discussed in e.g. [97]. The tool in particular implements "ipog", "ipog_d", "bush", "paintball", "ipof" and "ipof2" [5]. Again, only the option "ipog" supports constraints.

Both Lei et al. 2008 [97] and Lei et al. 2007 [96] describe a version of the IPOG algorithm that do not handle constraints. Yu et al. 2013 [168] introduces the algorithm IPOG-C, which introduces constraint handling into the IPOG algorithm in Lei et al. 2007 [96]. This paper was published in March 2013 and cites Paper 2 of this thesis.

Thus, the latest paper on the IPOG algorithm was published by Lei et al. 2008 [97]. The algorithm described in that paper in Section 3 will be discussed here. It is, however, not what is implemented in NIST ACTS version 2 beta, revision 1.5 under the option "ipog", because it actually handles constraints.

The Algorithm

Algorithm 1 shows the IPOG algorithm that is described in Lei et al. 2008 [97]. The algorithm is explained in the same level of details in Lei et al. 2008 [97]. It does not support constraints, but is the latest published algorithm before the acquisition of NIST ACTS version 2 beta, revision 1.5, which was used in our comparison.

It takes two parameters, an integer t and a set of parameters ps. The set of parameters is assumed sorted with the parameters with the largest domain first. ps contain k parameters. Each parameter is known as P_i for $1 \leq i \leq k$.

The algorithm starts by adding every combination of the first t parameters to ts. Because each combination is different, they must have one test set for each of them. The algorithm then

Algorithm 1 Generate a Covering Array with IPOG (no support for constraints): $IPOG(t, ps) : ts$

```
1: $ts \leftarrow$ a test for each combination of values of the first $t$ parameters
2: for $i = t + 1; i \leq k; i++$ do
3:     $\pi \leftarrow$ the set of all t-way combinations of values involving parameter $P_i$ and any group of $(t-1)$ parameters
       among the first $i - 1$ parameters
4:     for each $\tau = (v_1, v_2, ..., v_{i-1})$ in $ts$ do
5:         choose a value $v_i$ of $P_i$ and replace $\tau$ with $\tau' = (v_1, v_2, ..., v_{i-1}, v_i)$ so that $\tau'$ covers the most number
           of combinations of values in $\pi$.
6:         remove from $\pi$ the combinations of values covered by $\tau'$
7:     end for
8:     for each combination $\sigma$ in $\pi$ do
9:         if there exists a test $\tau$ in test set $ts$ such that it can be changed to cover $\sigma$ then
10:            change $\tau$ to cover $\sigma$
11:        else
12:            add a new test to cover $\sigma$
13:        end if
14:    end for
15: end for
```

proceeds to extend the test set one parameter at a time at Line 2, hence the name "In Parametric Order".

Within the outer loop from Line 2–15, there are two inner loops. The first inner loop is the extension to a new parameter, which is called horizontal growth. The second inner loop is the extension of new test cases, which is called vertical growth.

Before these two loops are run, a set π is initialized to all combinations of t values of the preceding $i - 1$ parameters including the values of the current working-parameter P_i.

The horizontal growth loop (Line 4–7) iterates through all test sets. Each test set get a value of the current working parameter P_i such that it covers the most number of combinations in π. On the next line, these newly covered combinations are removed from π.

The vertical growth loop (Line 8–14) iterates through all combinations σ in π. This loop checks if a test can be changed to include the combination, Line 9, and, if it can, that test is modified accordingly. It is possible to change a test without covering fewer combinations. The reason is the presence of *wild-cards*. Changing a test means specifying the wild-card in order to cover σ. If it cannot be changed, then a new test is added that covers σ, Line 12.

Comparison

These are the main similarities and differences between ICPL and IPOG as presented in Lei et al. 2008 [97] and Algorithm 1.

- ICPL handles constraints, and IPOG does not.
- Both ICPL and IPOG approach the SCP problem (Set Cover Problem), of which covering array generation is an instance, with a greedy polynomial time approximation algorithm similar to that described by Chvátal 1979 [31].
- ICPL fills up an entire test greedily with as many uncovered combinations as possible before it moves on to the next test. It then continues until all combinations are covered. IPOG, in contrast, starts with $i = t$ parameters and then covers all combinations of t

parameters of the first i parameters before it moves on to the $(i+1)$th parameter. It adds tests when a combination does not fit any test. In other words, ICPL grows only vertically.

- Both IPOG and ICPL support parameters with multiple values. They support them, however, in different ways. ICPL supports them by a constraint saying that only one of a set of values can be set to true. i.e. ICPL supports multiple values through the alternative construct: $v_1, \ldots, v_k$ are the k values of P, and they are constrained as follows: $((v_1 \vee \cdots \vee v_k) \Leftrightarrow P) \wedge \bigwedge_{i<j} \neg(v_i \wedge v_j)$. IPOG supports multiple values by having unique assignments to a parameter from a domain. The alternative construct used in ICPL might look more complicated than just having a parameter with a wide domain, but the construct is dealt with easily by SAT solvers: Either one value is set to true, and the whole construct passes, or else it does not.
- ICPL stores uncovered combinations in a Java `HashSet`. This set shrinks as more and more t-sets are covered, making the algorithm run quicker as more t-sets are covered. IPOG, on the other hand, stores the combinations in a special structure: Each parameter combination has its own pointer in an array. This pointer points to a bit-map structure that tells whether each value combination of the parameters is covered.

2.4.2 MoSo-PoLiTe

In Contribution 2 of this thesis, the tool *MoSo-PoLiTe v1.0.5 plug-in* (acquired 2012-01-25) to *pure::variants 3.0.21.369* (acquired 2012-01-02) was compared to *SPLCATool v0.3 (SPLC 2012)* with respect to covering array generation.

The MoSo-PoLiTe plug-in was not openly available but was provided to us under a research license. The source code was provided but under a non-disclosure agreement.

In Perrouin et al. 2011 [123], it was reported that MoSo-PoLiTe generated a 3-wise covering array from the feature model with 287 features. That covering array was of size 841. In our experiments, we observed significantly longer execution times for MoSo-PoLiTe than what was reported in Perrouin et al. 2011 [123], but it produced similar covering array sizes.

In personal correspondence with the authors, we determined that the version of MoSo-PoLiTe used in Perrouin et al. 2011 [123] is most likely different from the version provided to us. The version used Perrouin et al. 2011 [123] was not provided to us.

The version of MoSo-PoLiTe provided to us does not have a full support for constraints. It does support *require* and *exclude* constraints, but it has no support for converting constraints that can be rewritten as a conjunction of require and exclude constraints.

Within the provided system, several algorithms were implemented. We used the algorithm implemented in the classes `Pairwise_NoRP` and `Threewise` after recommendation by the creators. `Pairwise_NoRP` uses Hash sets and is not deterministic.

The Algorithm

Because the provided source of MoSo-PoLiTe is under a non-disclosure agreement, we have chosen to describe the algorithm as published in the latest paper prior to us acquiring the implementation, Oster et al. 2010 [117]. The paper does not give any pseudo-code of the algorithm. The algorithm is only briefly described in text. Thus, its details are unavailable.

The MoSo-PoLiTe algorithm consists of two parts: 1) flattening and 2) sub-set extraction. The input of the algorithm is a basic feature model as describe in Table 2.1 except the propositional formula must be a conjunction of require and exclude constraints and not a general propositional formula. i.e. if a require b and c excludes d, then the constraint P is $(a \to b) \wedge (c \to \neg d)$. The algorithm as described in the document only supports 2-wise covering arrays, thus a value of t is not given.

The output of the algorithm is a set of configurations, i.e. a set of set of tuple(f, b) where f is a feature name and b is a Boolean specifying whether the f is included or not.

Sub-Step 1: Flattening The first sub-step of MoSo-PoLiTe is to flatten the input to an intermediate type of model that is then used as input to the second sub-step, the subset extraction.

The intermediate model is as follows. It is a tuple (P, C) where P is a set of parameters and C is a constraint. Each parameter is a tuple $(names, values)$ where $names$ is a set of names that comprise the parameter, and $values$ is the set of values of the parameter. Only one value can legally be assigned to a parameter. e.g. for Boolean parameters p, the two values are $\neg p$ and p. e.g. if a parameter is mandatory, only one value is possible.

The constraint C is a conjunction of require and exclude constraints. The primitives of this constraint are the values of the parameters.

In order to construct the intermediate model (P, C), a set of rules are applied to the tree bottom-up. It is called 'flattening' because for each application of a rule, the tree becomes flatter. Eventually, the tree becomes entirely flat, at which point it fits into the intermediate model.

Oster et al. 2010 [117] refers to an associated website [116] for the 16 rules that comprise the flattening algorithm. These 16 rules include all possible situations that can occur in a valid model for MoSo-PoLiTe. Thus, an application of these rules bottom-up guarantees a flat model eventually.

For each rule, there is a grandparent node, one or two parent nodes and one or two child nodes.

The first four rules deal with situations involving one mandatory child node. In all four situations, the parent node is merged with the child node. The parent node thus becomes a set of two or more node names.

The next tree rules deal with the situations involving one single optional child node. In all three situations, the child node is added as a child to the grandparent node, and a require constraint is added saying that the child requires the old parent node.

The final nine rules deal with the remaining situations utilizing similar substitutions.

Sub-Step 2: Subset Extraction The second sub-step is only briefly sketched. Oster et al. 2010 [117] states that this sub-step is done in a similar manner as IPO. IPO is similar to what was described above as a part of the IPOG algorithm.

MoSo-PoLiTe reduces the set of combinations by looking at the constraint and the legal parameter assignments. Its implementation of IPOG thus reduces the set of combinations that are considered in the sub-steps of IPOG to those that are valid.

In addition, MoSo-PoLiTe applies a technique called 'forward checking'. They reference Haralick and Elliott 1980 [65] for what this technique consists of. Forward checking means that when a pair is considered added to a configuration, it is first confirmed that it will not inevitably lead to an invalid configuration. The reason forward checking is required is that two pairs that are valid might lead to an invalid configuration when added to the same configuration. Thus, when covering a legal pair, it must be checked that it does not cause the configuration to become invalid as a total.

Comparison

MoSo-PoLiTe as described in Oster et al. 2010 [117] has the following similarities and differences with IPOG as described in Lei et al. 2007 [96] and with ICPL.

- ICPL handles all propositional constraints; MoSo-PoLiTe supports binary constraints; IPOG does not support constraints.
- ICPL handles propositional constraints using a SAT solver; MoSo-PoLiTe support constraints by excluding invalid pairs based on the binary constraints and using forward checking.
- ICPL generates a propositional constraint from the basic feature model given to it; MoSo-PoLiTe flattens its input into its own intermediate format.
- MoSo-PoLiTe can only generate 2-wise covering arrays. IPOG and ICPL can generate t-wise covering arrays
- MoSo-PoLiTe approaches the SCP problem as both ICPL and IPOG, with a greedy polynomial time approximation algorithm similar to that described by Chvátal 1979 [31].
- MoSo-PoLiTe covers one parameter at a time, similarly to IPOG; ICPL generates one entire configuration at a time.
- For its intermediate format, MoSo-PoLiTe supports parameters with multiple values using the alternative construct in the same way as ICPL. For the covering array generation, however, the alternative construct is treated as a set of possible value assignments instead, just as IPOG.

2.4.3 CASA

In Contribution 2 of this thesis, the tool *Covering Arrays by Simulated Annealing (CASA) v1.1* was compared to SPLCATool v0.3 (SPLC 2012) with respect to covering array generation. This tool is free and open-source[2], and it implements an algorithms described in Garvin et al. 2011 [59], a development of earlier results presented in Garvin et al. 2009 [58].

Before the work by Garvin et al., Simulated Annealing (SA) had been studied for some time for the generation of covering arrays. The more general class of algorithms containing simulated annealing is the meta-heuristic algorithms.

Nurmela and Östergård 1993 [110] used simulated annealing to construct covering designs, which are similar to covering arrays. Stardom 2001 [145] investigated several meta-heuristic algorithms for covering array generation including, simulated annealing, Tabu Search (TS) and Genetic Algorithms (GA). Cohen et al. 2003 [43] and Cohen et al. 2003b [39] integrated the previous work on simulated annealing with an algebraic approach for better covering array generation. Most of the early work on simulated annealing did not consider constraints. Cohen et

[2]`http://cse.unl.edu/~citportal/tools/casa/`, accessed 2013-05-29

al. 2007 [41] extended simulated annealing with a SAT solver in order to deal with constraints. This forms the basis of which Garvin et al. 2009 [58] developed the algorithms for the CASA tool.

The Algorithm

Garvin et al. 2011 [59] shows the pseudo code of a basic version of the simulated annealing algorithm for covering array generation. They then proceeded to describe two main improvements and six refinements to improve this algorithm. Except one, these improvements are described textually. The improvements are, however, specific to the simulated annealing approach. Thus, because the purpose here is to compare CASA with ICPL, MoSo-PoLiTe and IPOG, the basic version of CASA will be described and compared with these three other algorithms.

Algorithm 2 shows the highest level of the basic CASA algorithm. It takes a value t for the strength of the covering array. k is the number of parameters, and v is the number of values for each parameter. C is the complete propositional constraint. $lower$ and $upper$ are the size bounds within which a covering array will be searched for. For example, if $lower$ is 1 and $upper$ is 10, a covering of size 5 might be found, being between 1 and 10.

Algorithm 2 Generate a Covering Array with CASA:
$CASA(t, k, v, C, lower, upper) : A$

```
 1: $A \leftarrow \emptyset$
 2: $N \leftarrow \lfloor (lower + 2upper)/3 \rfloor$
 3: while $upper \geq lower$ do
 4:     $A' \leftarrow anneal(t, k, v, C, N)$
 5:     if $countNoncoverage(A') = 0$ then
 6:         $A \leftarrow A'$
 7:         $upper \leftarrow N - 1$
 8:     else
 9:         $lower \leftarrow N + 1$
10:     end if
11: end while
```

Algorithm 2 implements a binary search within the bounds. First, it makes a suggestion, Line 2. For example, if $lower$ is 1 and $upper$ is 10, N is first set to $(1 + 2{\cdot}10)/3 = 7$. Then, the loop at Line 3-11 is run until the space has been delimited. It is delimited in two ways: Line 4 finds a valid array of configurations using the $anneal$ method described later. If it is a complete covering array, i.e. if there is no non-coverage, that solution is stored and the upper bound is lowered. If, however, the coverage is incomplete, the lower bound is raised.

Algorithm 2 calls Algorithm 3. Algorithm 3 takes the values t, k, v and C directly from Algorithm 2. N is the current intermediate between $lower$ and $upper$. It returns an array of products that may or may not be a covering array.

Algorithm 3 calls various methods that are given as pseudo-code: $initialState$, $stabilized$, $countNoncoverage$, SAT and $cool$. The method starts by calling one of these functions, $initialState$, in order to get an array of size N. Then, $temperature$ is initialized to some initial value. The main loop starting at Line 3 is continued until the $stabilized$ method has determined that the number of coverage has stabilized. The loop starts by selecting a random row and column, and then it copies the array and modifies one of its entries by selecting a new

Algorithm 3 Find an Array:
$anneal(t, k, v, C, N) : A$

```
1: $A \leftarrow initialState(t, k, v, C, N)$
2: $temperature \leftarrow initialTemperature$
3: while $\neg stabilized(countNoncoverage(A))$ do
4:     $(row, column) \leftarrow$ random value from $(1..N, 1..k)$
5:     $A \leftarrow A'$
6:     $A'_{row,column} \leftarrow$ random value from $v_{column}$
7:     if $SAT(C, A'_{row,1..k})$ then
8:         $\Delta fitness \leftarrow countNoncoverage(A') - countNoncoverage(A)$
9:         $p \leftarrow$ true with probability of $e^{-\Delta fitness/temperature}$
10:        if $(\Delta fitness \leq 0) \vee p$ then
11:            $A \leftarrow A'$
12:        end if
13:        $temperature \leftarrow cool(temperature)$
14:    end if
15: end while
```

random value. It is then checked, Line 7, whether that new, modified row is valid according to the constraints. If it is, a change in fitness value is calculated, Line 8, and the variable p is set to true with a certain probability depending on the fitness change value and the temperature. If the fitness value is an improvement, the change is negative or the random value is true, then the temporary change is made permanent. Then, the temperature is cooled making it less likely that p will become true (because $e^{-\Delta fitness/temperature} \rightarrow 0$ as $temperature \rightarrow 0$).

Algorithm 3 will stabilize $countNoncoverage(A)$ if it successfully finds a Covering Array of size N.

Comparison

In our comparison, CASA was run with the default settings. This makes it estimate a good lower and upper bound for the sizes.

- CASA is not based on a greedy approach (as ICPL, IPOG and MoSo-PoLiTe) but uses *simulated annealing*. Thus, instead of being a greedy-algorithm, it starts with a valid array, and then tries to improve it. The initial array might of course have been made by a greedy algorithm.
- CASA supports general propositional constraints in the same way as ICPL, by using a SAT-solver. As mentioned, MoSo-PoLiTe and IPOG does not support general constraints.
- CASA deals with parameters with multiple values using the same technique as MoSo-PoLiTe and ICPL: Each value is either selected or not but only one can be selected. This makes it possible to use an off-the-shelf SAT solver, just like in ICPL.
- CASA requires its input to be in the form of a list of parameter, value and one constraint expressed in CNF form. ICPL supports generating this from a basic feature model first. IPOG input is without constraints, and MoSo-PoLiTe supports *require* and *exclude* constraints only, in addition standard feature groups.
- CASA supports generating t-wise covering arrays just as ICPL and IPOG, unlike MoSo-PoLiTe.

2.4.4 Other Algorithms

There are many papers on the problem of covering array generation. The website `pairwise.org` lists 37 tools for covering array generation. Nie and Leung 2011 [109] had collected more than 50 papers about the problem of covering array generation. Here are some of the other algorithms for covering array generation.

According to our investigations, none of these algorithms perform better than ICPL when it comes to generating t-wise covering arrays from large feature models with general propositional constraints. This was recently seconded by Liebig et al. 2013 [99].

AETG The *Automatic Efficient Test Case Generator* (AETG) was an early algorithm for covering array generation presented in Cohen et al. 1994 [35], Cohen et al. 1996 [37] and Cohen et al. 1997 [36]. It is currently available as a commercial product accessible via a subscription based web interface[3].

AETG is a greedy algorithm. It has some support for constraints by listing the illegal combinations explicitly. It also has some support for generating the list of invalid combinations from propositional formulas. Further details of the algorithm are, however, unavailable.

Cohen et al. 2006 [40] explained how combinatorial interaction testing though AETG could be used to test product lines.

AllPairs The *AllPairs* tool [6] implements an algorithm for covering array generation. The tool is freely available online[4]. It uses a greedy approach, and builds up the answer row by row.

ATGT The *ASM Test Generation Tool* (ATGT) was presented in Calvagna and Gargantini 2008 [23] and Calvagna and Gargantini 2009 [24]. The algorithm takes Abstract State Machines (ASM) with constraints as input and produces a set of test cases for the ASMs. Constraints are dealt with by invoking SAL, a bounded and symbolic model checker tool [47].

The tool is available online, freely[5].

The *Laboratory for Combinatorial Interaction Testing* (CITLAB)[6] was introduced by Calvagna et al. 2012 [56], Calvagna et al. 2013a [28] and Calvagna et al. 2013b [27]. Just as SPLCATool (the tool implementing the contributions of this thesis), it supports importing feature models. However, instead of converting it to a propositional constraint, it converts it to input to the multi-valued parameter formats used by AETG and IPOG. They used it as a framework for comparing the performance of various algorithms for covering array generation algorithms, including AETG, DDA, PairTest, CASA and IPOG.

DDA The *Deterministic Density Algorithm* (DDA) was introduced in Colbourn et al. 2004 [44] and in Bryce and Colbourn 2006 [21]. It is also a greedy algorithm with a new heuristic

[3]`http://aetgweb.argreenhouse.com/`, accessed 2013-05-28

[4]`http://www.mcdowella.demon.co.uk/allPairs.html`, accessed 2013-05-28

[5]`http://cs.unibg.it/gargantini/software/atgt/atgt_smt_ccit.html`, accessed 2013-05-28

[6]Available at `https://code.google.com/a/eclipselabs.org/p/citlab/`, accessed 2013-05-28

that makes it faster and more often accurate than four other algorithms they it was compared to in an experiment: AETG, TCG, IPO and TConfig.

GTWay Zamli et al. 2011 [169] introduced an algorithm for covering array generation called *GTWay*. Their algorithm supports concurrent execution, and performs better than the compared algorithms for high coverage strength, $t > 6$. They compared GTWay with IPOG, Whitch, Jenny, TConfig and TVG II. GTWay was a development of *G2Way* introduced in Klaib et al. 2008 [91]. It is a backtracking algorithm that tries to combine uncovered interactions in the best possible way. It sets up the possible pairs in a similar structure as IPOG (discussed above).

IPOG-C Yu et al. 2013 [168] introduces the IPOG-C algorithm. It introduces constraint handling into the IPOG algorithm in Lei et al. 2007 [96]. IPOG-C utilizes some optimization techniques similar to those in ICPL in addition to some new techniques. (The paper cites Paper 2 of this thesis.)

mAETG mAETG was introduced in Cohen et al. 2003 [43]. It was extended with constrain handling using a SAT solver in Cohen et al. 2007 [41] and Cohen et al. 2008 [42]. mAETG, like AETG, is a greedy algorithm.

PACOGEN PACOGEN is an algorithm and a tool[7] by Hervieu et al. 2011 [73]. It can generate covering arrays from constrained feature models. It uses constraint programming instead of a SAT-solver to validate constraints. PACOGEN makes a constraint-solving problem out of the problem of generating a 2-wise covering array from a specific feature model. It then runs a branch-and-bound method to find a covering array.

Perrouin's Algorithm Perrouin et al. 2010 [124] introduced an algorithm for covering array generation that converted the problem of generating a covering array into a problem for Alloy [75]. They presented several optimizations to make this quicker. In Perrouin et al. 2011 [123] this algorithm was compared with MoSo-PoLiTe. They found that MoSo-PoLiTe was more stable, was much faster and produced smaller covering arrays.

PICT Microsoft's tool PICT implements an AETG-like algorithm that is optimized for speed. It was introduced in Czerwonka 2006 [46].

Other There are even more algorithms: Calvagna's Algorithm [29], Jenny [76], AETGm [38], a Generic Algorithm (GA) and an Ant Colony Algorithm (ACA) was introduced in Shiba et al. 2004 [143], PairTest [153], Grieskamp's Algorithm [61], IPO-s [25], IBM *Intelligent Test Case Generator* (Whitch) CTS and TOFU [66], TConfig [167], TestCover [142] and *Test Case Generator* (TGC) [158].

[7]PACOGEN is available at `http://hervieu.info/pacogen/` and `http://www.irisa.fr/lande/gotlieb/resources/Pacogen/`, accessed 2013-05-28

Chapter 3

Contributions

3.1 Overview and Relation to the State of the Art

Figure 3.1 shows a test process for product lines and where in this process this thesis contributes (marked 1–5.) Three assets are needed before the test process can be applied: A feature diagram (Asset A), an implementation (Asset B) and test cases (Asset C).

None of the contributions address the generation of tests (Asset C), in contrast to PLUTO, ScenTED, CADeT, MoSo-PoLiTe and incremental testing. They address this issue in various ways and also integrate this selection with product selection.

Contribution 1, 2 and 4 address the sub-setting of products for product line testing. Products are proposed selected by combinatorial selection techniques as in CIT, MoSo-PoLiTe, CADeT and some of the incremental testing approaches. Contribution 1 and 2 investigate and improve the efficiency of algorithms for test product selection. The ICPL algorithm, described in Contribution 2, enables the (2-wise) sub-setting of products for product lines with as much as 7,000 features, one of the largest documented product lines. Contribution 4 enables market-focused selection and prioritization for product selection with combinatorial techniques.

Contribution 3 is a technique for using information in the implementation (Asset B) to improve the testing of the selected products: Products that differ only in the implementations of the homogeneous abstraction layers can be tested using the same test suites and their results can be compared to form a voting oracle.

Finally, Contribution 5 addresses the automation of the entire testing process shown in Figure 3.1. Assuming the existence of the Asset A, B and C, the proposed algorithms and their implementations automatically generate Asset D, E and F. The automated test process is demonstrated by a fully automated application to the part of the Eclipse IDE maintained by the Eclipse Project. Such a large-scale, fully reproducible and fully documented application of a product line testing technique is not found in the existing literature on product line testing.

The idea of Contribution 5 is not necessarily to represent a finalized and all-encompassing testing process, but rather to serve as a base-line for comparison and advancement of scalable product line testing techniques.

The five contributions are covered in four sections. The generation of covering arrays from large feature models (Contribution 1 and 2) is covered in Section 3.2, and then three advancements of CIT (Contributions 3–5) are presented in sections 3.3–3.5 respectively.

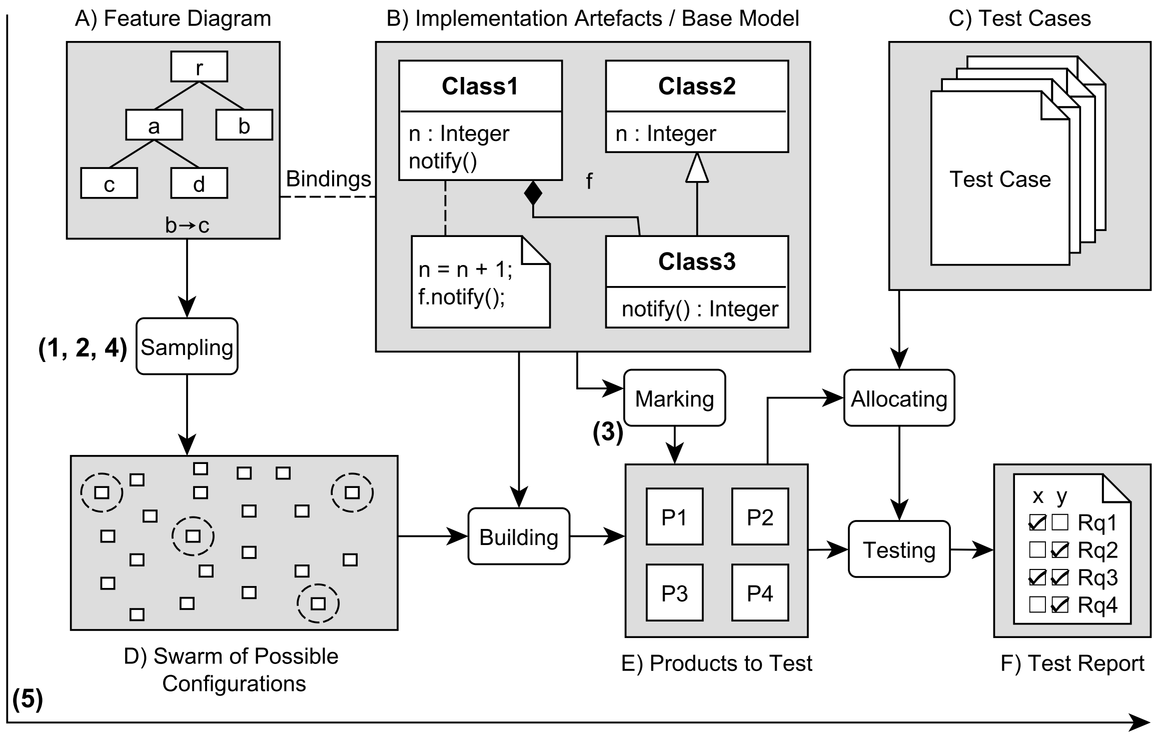


Figure 3.1: Contributions 1, 2, 3, 4 and 5 as Parts of a Whole Test Process

3.2 Generation of Covering Arrays from Large Feature Models

3.2.1 Difficulty of Covering Array Generation

The generation of t-wise covering arrays is generally regarded to be an intractable problem [109]. This is arrived at through *complexity analysis*, with which it is classified as NP-hard.

The argument is: In order to get a set of valid products, one valid product must be found. A basic feature model is expressively equivalent to a Boolean formula; thus, finding a single valid product is equivalent to the Boolean satisfiability problem (SAT). This problem is the classic NP-hard problem. NP-hard means the problem is at least as hard as the hardest problem in NP [55].

In complexity analysis, an algorithm's complexity is usually arrived at by considering the worst of the valid inputs (worst-case analysis). This is of course useful when any input can occur. However, such analysis can be too pessimistic when the context of a problem limits which inputs will ever be presented to an algorithm.

SPLE-SAT is Easy

Such is the case for feature models in the context of software product line engineering (SPLE). This was the main theme of our paper titled "Properties of Realistic Feature Models make Combinatorial Testing of Product Lines Feasible" published at MODELS 2011 [78], Paper 1.

It is not unique for the SPLE context. In a series of publications starting with [30] and ending in [17], it was established that the satisfiability problem is easy for Boolean formulas originating from realistic VLSI circuits (Very-Large-Scale Integration circuits). This sub-problem was termed *ATPG-SAT* (Automatic Test Pattern Generation-SAT). These quotes from [30] should make their reasoning clear:

> It has been observed that SAT formulae derived from ATPG problems are efficiently solvable in practice. This seems counter-intuitive since SAT is known to be NP-Complete. This work seeks to explain this paradox. We identify a certain property of circuits which facilitates efficient solution of ATPG-SAT instances arising from them. In addition, we provide both theoretical proofs and empirical evidence to argue that a large fraction of practical VLSI circuits could be expected to have the said property.

and

> Note that we are not generating arbitrary SAT formulae during ATPG; there is some structural quality inherent in ATPG problems which makes them easy to solve.

Following the same style as ATPG-SAT, the sub-problem of SAT originating from realistic feature models can be called *SPLE-SAT*.

The role of feature models in SPLE is for potential customers to be able to configure a product according to their needs. Had this been difficult, the customers could not do this. Indeed, in the worst case, not a single product from the product line could be delivered. This is absurd. In SPLE, feature models are easily configurable in order to serve their purpose. Figure 3.2 shows the situation in a Venn diagram: Even though both SPLE-SAT the hard SAT problems are both sub-sets of SAT, they do not necessarily intersect. Given that this situation is true, SAT solvers can be used to configure feature models without risking intractability.

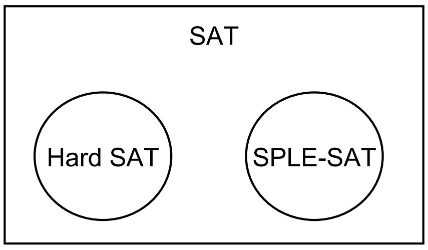


Figure 3.2: Although there are hard SAT problems, SPLE-SAT need not overlap the region.

Paper 1 establishes that SPLE-SAT is easy in two parts: the argument just put forth and an empirical investigation. 19 realistic feature models were investigated. It was found that even for the largest feature models publicly available, the Linux Kernel feature model, satisfiability is quick, taking 125 ms with SAT4J. Indeed, the Linux kernel is meant to be configured by hand (assisted by a SAT solver), something that is regularly done by advanced Linux users.

Earlier than our work, Mendonca et al. 2009 [106] had noticed that satisfiability for SPLE-SAT for some reason was quick in practice. They did not provide an explanation for this, however.

Covering Array Generation is Feasible

Now, given that SPLE-SAT is easy, generation of covering arrays then, by complexity analysis, reduces to the set-cover problem (SCP). SCP is not NP-hard; it is actually NP-complete and has a polynomial time approximation algorithm with an established and acceptable upper bound [31]. This algorithm is a greedy approximation of SCP. Approximation, in this context, means that there will be more products than optimally possible, but it also means that the number of products in the worst-case still is limited by some upper bound. The greedy algorithm is quite simple: Select the configuration that covers the most uncovered interactions until all interactions are covered. This algorithm needs to be modified for covering array generation from feature models because the configuration space generally is exponential with respect to the number of features. This additional stage is a contribution of Paper 1: This is solved by greedily packing uncovered interactions into a configuration. This voids Chvátal's upper bound guarantee; however, we did investigate the impact of this as follows:

A tool was implemented, and a corpus of 19 feature models was collected. The experiment is documented in Paper 1. In it, all 19 feature models were given to the tool, and 1–4-wise covering arrays were attempted generated. It could be understood from the sizes of the feature models in the corpus when a covering array would take too long to generate.

As just mentioned, the algorithm was based on a greedy approximation for the set-cover problem (SCP). This basic algorithm was shown to have an acceptable upper bound [31]; but, for feature models, due to their huge configuration space, this upper bound cannot be achieved. Still, it was shown that for the realistic feature models, the covering arrays did stay within an acceptable size. Due to the central role of a satisfiability solver within the algorithms, we suggested that the algorithm was a good basis for a scalable algorithm for t-wise covering array generation.

The algorithm's performance is not compared with the other available algorithms for t-wise covering array generation. The performance of this algorithm is, however, on par with the other state of the art algorithm, as would be later documented in Paper 2, which contributions are discussed next.

3.2.2 Scalable t-wise Covering Array Generation

Having established the tractability of SPLE-SAT and having a basic algorithm for covering array generation with potentials for improvement, the basis was set for the development of a scalable algorithm for t-wise covering array generation.

Such an algorithm was the contribution of our paper titled "An Algorithm for Generating t-wise Covering Arrays from Large Feature Models" published at SPLC 2012 [79], Paper 2.

The algorithm named ICPL was developed and implemented by profiling the basic algorithm and speeding up its bottlenecks by an analysis of the situation creating the bottleneck. This resulted in a range of optimizations presented in Section 3 and 4 of Paper 2. (The interested reader is advised to read those sections at this point.)

To establish its performance, an evaluation and comparison was set up. The results of this evaluation and comparison are reported in Section 5 of Paper 2. The performance of the algorithm was compared to the basic algorithm from Paper 1, called "Algorithm 1", and three of the leading algorithms for covering array generation: IPOG [96], CASA [59] and MoSo-PoLiTe [117]. Thus, these five algorithms were compared: ICPL, Algorithm 1, IPOG, CASA and MoSo-PoLiTe. This comparison is presented as the space permits in Paper 2. The remaining details and the additional diagrams are added as Appendix C. A detailed description of IPOG, CASA and MoSo-PoLiTe is found in Section 2.4 including a qualitative comparison with ICPL.

The evaluation of ICPL shows that only it can generate the 2-wise covering array from the largest feature model available, the Linux Kernel, within a time limit of 12 hours. This feature model is gigantic with 6,888 features and 187,193 clauses in its complete CNF-constraint.

The speed of each algorithm was measured by giving each algorithm three whole days to generate 100 covering arrays for each feature model and for each strength: 1, 2 and 3. All algorithms managed to do this for the same 16, 14 and 12 feature models for 1, 2 and 3-wise covering arrays, respectively. A statistical analysis of these runs showed that ICPL's time is $O(f^{0.76})$ with respect to the number of feature in the feature mode, f, while the second fastest, MoSo-Polite's, time is $O(f^{1.75})$. For 3-wise covering arrays, ICPL's time is $O(f^{1.14})$ against $O(f^{2.62})$ for the second fastest, MoSo-PoLiTe.

For a feature model with 1,000 features, ICPL is almost 1,000 times faster than the second best for 2-wise covering array generation, and ICPL is almost 30,000 times faster than the second best for 3-wise covering array generation.

3.2.3 Consolidations

For this section of contributions, some consolidations can be noted.

- The existence and performance of ICPL are further arguments in favor of the claim of Paper 1 that the generation of t-wise covering arrays from realistic feature models is feasible in practice.
- Henard et al. 2012 [72] acknowledges that the tool developed for our MODELS 2011-paper (SPLCATool v0.2) was a state of the art tool for covering array analysis and generation. They used it as the main comparison to their own contributed algorithms and implementations.
- Paper 2 suggested that there were more optimizations possible for t-wise covering array generation. Haslinger et al. 2013 [68] suggest additional improvements to speed up the algorithm.
- Kowal et al. 2013 [93] further improves ICPL by adding a filtering step before running ICPL. This filter filters out feature combinations that are known not to interact. They show that this both decreases the time it takes to run ICPL and decreases the number of products generated by ICPL.
- Liebig et al. 2013 [99] propose a variability aware analysis technique for validating product lines. They evaluate their proposed technique by comparing it with three sampling techniques, one being pair-wise sampling. For pair-wise sampling they chose the ICPL algorithm proposed in Paper 2 of this thesis as implemented in SPLCATool. They report that not only is this the fastest algorithm they could find for pair-wise covering array generation, it was also the only algorithms that was scalable enough to be used in their experiments.

3.3 Optimized Testing in the Presence of Homogeneous Abstraction Layers

Cabral et al. 2010 [22] writes that "Alternative features [...] present a more difficult challenge for testability. We argue that these are the true deterrents to testability since only one feature can be present in an SPL at a time."

One type of common software construct causes alternative features in the product line's feature model: *homogeneous abstraction layers*. Customers usually have a wide range of different deployed software that provides essentially the same functionality: operating systems—with their file handling, networking, virtual memory, threading and synchronization—databases, with their obvious functionality; windowing systems, interfaces, hardware architecture, etc. These all come in different variants (e.g. Windows, Linux, MacOS, etc.; Oracle, Microsoft SQL, MySQL, PostgreSQL, etc; Windows, GTK, Motif, Photon, etc; Intel x86, Intel x86-64, PowerPC, ARM, etc.), but provide essentially the same functionality.

A technique used to make a system run in different environments is to make an abstraction layer that is implemented concretely for each particular system but that provides a uniform way of interacting with them to the product line code. Thus, in order to make a product of a product line function in a specific environment, we need to set up the right implementation of the abstraction layer. The selection of such an implementation must be modeled as an alternative feature.

Thus, when a product line is designed to be able to run in differing environments, its feature model will have many alternative features. This again, deters the testability of the product line with CIT.

We found a way to turn this problem into a benefit. Whenever alternative features due to homogeneous abstraction layers cause a deterioration of the performance of combinatorial interaction testing, it can be turned around and used to help testing instead. This result was presented in the article titled "Bow Tie Testing: A Testing Pattern for Product Lines" published in "Pattern Languages of Programs", the proceedings of the 16th European Conference, 2012[1] [80], Paper 3.

3.3.1 Description of the Technique

Because implementations of homogeneous abstraction layers are supposed to provide the same functionality to the product line assets, we know that whatever concrete implementation is chosen, the product will behave essentially the same. Thus, if a test case is run on two products that only differ in their implementations of their homogeneous abstraction layers, the test will result in essentially the same execution trace. Thus, we can compare the executions; if they differ, we know that something is wrong. It is of course not limited to two products. If there are three or more, we can construct a *voting oracle* [18]; if we know one of the executions are correct, we can use it as a *gold standard oracle* [18].

We did implement tool support for the detection of such groups of products. Annotate a feature model to inform the tool which alternative feature groups are due to a homogeneous

[1]This paper was published at a conference on patterns. The patterns community have several special requirements in the presentation of a pattern. In this part of the thesis, the pattern-style of presentation will not be used.

abstraction layer. Then, the tool will note groups of products that only differ beneath this layer.

3.3.2 Application to Finale's Systems

Finale's systems are systems for reporting various things about a company's finances to the government. This is required by the government to ensure that all the taxes are collected as required.

Finale's systems run on all versions of Windows that is supported by Microsoft. When we worked with this case study, three different versions were supported on the client-side: Windows XP SP3, Windows Vista SP2 and Windows 7. All come in 32-bit and 64-bit variants. Finale's systems had to run correctly on all systems with a graphical interface zoom level of 100%, 125% and 150%. On the servicer-side, four versions of windows were supported: Windows 2000, Windows 2003, Windows 2008 and Windows 2008 SR2. The major abstraction layer was, however, towards the accounting systems. They supported 18 types of accounting systems, all interacted with through a homogeneous abstraction layer.

If we want to exercise all pairs of interactions, all products containing a version of client-side Windows paired with a version of an accounting system, then we need at least $6 * 18 = 108$ products. Indeed, the pair-wise covering array contains 116 products. If we group together those products that only differ below a homogeneous abstraction layer, we reduce the number to 60.

The Finale case was not documented in detail in Paper 3 because of confidentiality concerns. Instead, we used the more open Eclipse IDE case study.

3.3.3 Application to the Eclipse IDEs

The Eclipse IDEs are used as a case study throughout this thesis. One aspect of the Eclipse IDEs is given focus only in this part. Although Eclipse is programmed mostly in Java, it is not completely so. The reason is that the programmers found it to be too slow. Thus, they replaced the implementations of abstraction layers in Java with their own implementations. For version 3.7.0, they supported six operating systems: Windows, Linux, MacOS X, Solaris, AIX and hpux. These operating systems provided various windowing systems and ran on various hardware architectures. In total, they support 16 combinations of hardware, windowing systems and operating systems.

This caused the pair-wise covering array of the sub-set of the Eclipse IDEs used in this part to grow to 41 products. Out of these, only 20 groups of products need to be tested with a unique test suite, the rest differ only beneath the homogeneous abstraction layers. Covering the system with 1-wise testing gives 10 products, but only 5 test suites are required. For 3-wise testing, with 119 configurations, only 45 test suites are required.

It should be noted that if the other constructs in the feature model are the main cause the covering array's large size, then no benefits can be derived from the technique described here, but, then, the alternative features are not the cause of the deteriorations either.

For more details on this and on the Eclipse IDE case, see Appendix E.

3.4 Market-Focused Testing based on Weighted Sub-Product line Models

The idea behind combinatorial interaction testing is to exercise *all* simple interactions. In some cases, however, this implies some wasted effort. The product line's market can limit which products will occur. Then, there is no need to exercise all simple interactions.

In order to focus on the market-relevant interactions, the market situation must be captured or modeled somehow. Such a model is a contribution of our paper titled "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines" published at MODELS 2012 [82], Paper 4. Relevant algorithms based on this new type of model were introduced along with two reports on applications: to TOMRA's RVMs and to the Eclipse IDEs. These things are covered in turn in the next subsections. Note that because of space constraints in Paper 4, we did not include details of these algorithms. Such details are provided in Appendix B.

3.4.1 Weighted Sub-Product Line Models

During our work with the TOMRA industrial case study, the domain experts noted that although a customer could order any valid configuration of their feature model, it would never happen in practice. The reason was that there were certain clusters of features that were usually combined to serve the need of certain market segments.

To model this, we developed a new kind of model called a *Weighted Sub-Product Line Model*. This model is created in a spreadsheet editor such as Microsoft Excel or Open Office. It contains a set of sub-product line models. They are called sub-product lines because they are partial configurations of a feature model. Because a feature model defines a set of products, a sub-product line defines a sub-set of the feature model's products, thereby the term *sub-product line*. They are weighted because each sub-product line is annotated with a positive integer value that serves as a measure of that sub-product line's importance with respect to testing.

A sub-product line model is made by specifying, for each feature, whether that feature is included, excluded or unassigned. This last option retains some of the configurability. Had all features been specified as included or excluded, we would have a configuration. The symbol chosen for this last option is a question-mark ('?'); the symbols for included and excluded are a cross ('X') and a dash ('–'), respectively.

A sub-product line can be easily classified as valid or invalid: Simply set all the assignments specified in the sub-product line in the propositional formula, and ask a SAT-solver whether a valid configuration exist. If it does, the sub-product line is valid; if it does not, it is invalid.

A property of any valid sub-product line is that the number of products subsumed by it is equal to or smaller than the number of products subsumed by the product line itself. Another property is that these products are the same as or a subset of the products of the product line.

It can be argued that these models model the market situation relevant for testing purposes. Say that the sub-product lines are market segments, and that the weights are the size of that segment, they model the market situation in the following ways: (1) They might exclude a wide range of configurations that will never occur because they are not a valid configuration of any of the sub-product lines. (2) They capture the fact that certain products are more prevalent than other products, and should, in case of prioritization, be the higher priority. Predictions or expectations of the future can also be embedded in the weights. Then, testing is forward-looking in that it prioritizes those interactions that are more likely to occur in the future.

3.4.2 Algorithms Based on These Models

Weighted t-sets

Because we are doing combinatorial interaction testing, we want to get knowledge about t-sets. A t-set is given a weight in the following way: If a particular assignment of t features is in a sub-product line, then the sub-product line's weight is given to the t-set. If the t-set is captured within a sub-product line with n unassigned features, then it gets a $\frac{1}{2^n}$-part of the weight. This is because one question mark gives two possibilities, and thus n question marks give 2^n possibilities. This algorithm is fully described in Section B.2.

Weighted Covering Array Generation

During ordinary covering array generation, t-sets are selected from a set. Because sets are unordered, picking a t-set from a set means picking it at random. When each t-set has a weight, we can order them. One data structure for doing this is the *priority queue*. The t-set with the highest weight will always be on top of the priority queue. Instead, then, of selecting a t-set at random, the highest weighted t-set will be selected first.

All the weighted t-sets can be put into a priority queue from which t-sets will be picked during covering array generation. This means that the first products of a covering array will contain many feature interactions that are popular in the market. Thus, we can more meaningfully talk about a *partial covering array*. In ordinary covering array generation, the first product is selected so that they cover as many interactions as possible. These interactions might not be the most relevant to test, which is not the point of ordinary combinatorial interaction testing anyway because its purpose is to test all simple interactions. If we have to select a limited number of products, then a partial weighted covering array is a meaningful answer. An algorithm for generating such arrays is detailed in Section B.3.

Weight Coverage

In ordinary covering array generation, we want to cover simple interactions. Thus, we can define the t-wise coverage of a set of products as the number of t-sets they cover divided by the number of valid t-sets. With weights defined, however, we can introduce the notion of *weight coverage*. This is defined as the amount of weight covered by a set of products divided by the sum of all weights for all t-sets.

Incremental Evolution

The notions introduced thus far allow us to cater for another need. Given that we have an elaborate test setup, how can we gradually and incrementally adapt it? Such a gradual evolution is good because large changes might be costly or time consuming. Small gradual changes fit into a process of gradual evolvement of a product line, which is the usual process.

Let us say a new market segment is introduced, or the size of a market segment changes. How should the test lab be changed to gradually adapt to these changes? Given a set of products already set up and already being tested, we constructed an algorithm that suggest one, two or three changes to the lab such that the weight coverage is maximized because of the changes. These changes consists of changing a feature from included to excluded, or *vice versa*. Such an algorithm is detailed in Section B.4.

3.4.3 Application to TOMRA's RVMs

The main case study of Paper 4 is an application to the TOMRA case study. TOMRA Verilab is responsible for testing TOMRA's RVMs. They did not have a feature model of their product line. The first stage was to model the feature model (Fig. 2 of Paper 4) and to specify the configurations of the products in their test lab (Table 3 of Paper 4). After we had created the notion of a weighted sub-product line model, we modeled their market situation (Table 4 of Paper 4). This allowed us to do several experiments to evaluate the usefulness of this new kind of model.

1. The domain experts compared the products generated from ordinary covering array generation to those generated from weighted covering array generation. They found that the latter ones were more familiar to them and were closer to products found in the market.
2. A second experiment was to calculate the two kinds of coverages for their existing lab. We found that it had higher weight coverage than interaction coverage. This is consistent with the domain experts' claim that their lab is manually configured to be relevant for the market situation yet exercise a variety of different interactions.
3. A third experiment was to generate a completely new test lab as a partial weighted covering array. We found that significantly fewer products were needed than they currently had to achieve the same coverage. Another take on this is that they can achieve higher coverage with the same amounts of products in their lab.
4. A fourth experiment was to generate some simple changes to their existing lab to increase its coverage. These changes were evaluated by the domain experts. They were initially skeptical of the suggested changes, but, after some deliberation, they agreed that the suggestions were good. This shows how the automatic tooling takes into account more factors quicker.

Unfortunately, we did not get to the point of evaluating the error-detection capabilities of the approach due to time and resource limitations. Also and because the actual system was unavailable to us, the further application of the technique was outside our control.

3.4.4 Application to the Eclipse IDEs

We also got to document a small application to the Eclipse IDEs. This was primarily included to indicate the generality of the contributions. We could gain some knowledge of the market situation of the Eclipse IDEs because the Eclipse project publish the number of downloads for each of their 12 offered products. The downloads are an indication of the amount of that product out there. A small experiment showed that only four products were needed to be tested to exercise 95% of the weight of the interactions in the market.

For more details on the Eclipse IDE case, see Appendix E.

3.5 Agile and Automatic Software Product Line Testing

Up to this point, a lot has been said and claimed about product line testing, but something that is lacking is a complete, end-to-end, large-scale application of a testing technique. Having such a thing documented would be of use to researchers and industry, and it would give an insight into were new contributions are needed the most.

This was the motivation for the results in a paper titled "A Technique for Agile and Automatic Interaction Testing for Product Lines" published at ICTSS 2012 [81], Paper 5.

3.5.1 Challenges in Need of an Answer

There were a number of challenges that needed to be answered in order to get such a large-scale application in place.

We already had an open source case study that we could work with, the Eclipse IDEs. We also had a scalable way to select a small set of products that we could test, namely the ICPL algorithm. The challenges remaining and our answer to them are:

- **How to build the products automatically?** As for the Eclipse IDEs, we found that the *Equinox p2 Director* of the Eclipse Project was bundled with the Eclipse Platform. It allowed features to be installed programmatically. Thus, by providing a list of features, we could automatically construct an Eclipse IDE product from scratch. This is an application of the *components, frameworks and plug-ins* approach to software product line engineering, discussed in Section 2.1.2.

  In addition, the CVL Tool v1.x already had the capability to build product models automatically. Thus, we decided to include a CVL-based product line as a case study. This is an application of the *CVL* approach to software product line engineering, discussed in Section 2.1.2.

- **How to get test-assets?** In the Eclipse project, each feature is developed by a separate team. As a part of each feature project, a test suite is developed to test that feature as a feature. This was a source of test cases that we could try to utilize for product line testing.

  During our work with the ABB case study, we had set up some test cases that exercised various features. These would serve the same purpose as the tests gathered for the Eclipse IDE case.

- **How to run the test?** The answer to this question depends on the availability of test assets. What we have for the Eclipse IDE are test cases that test the features (or a combination of them). Because making test cases is time-consuming and requires domain expertise, it was infeasible to do it within the time frame of the thesis. We established that we could in fact learn some interesting things by exercising a feature (or a combination of features) when that feature (or that combination of features) is a part of some product (see the next subsection for an explanation). Thus, it was decided that the last challenge that needed an answer could be answered by running the existing test suites for the features (or a combination of features) on the products that contain them.

3.5.2 Testing Interactions using Tests for Features

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 3.1a.

Table 3.1: 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 3.1b. 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 3.1b and to higher orders of combinations.

3.5.3 Evaluation of Previous Techniques

Before we could construct a testing technique that could be applied to large-scale software product line, we needed to know that whatever we learn from the application of the new technique, we did not already know with another existing technique. The two other similarly, directly applicable techniques are reusable component testing (RCT) and combinatorial interaction testing (CIT). For a discussion of other techniques and the reason why they are not as easily applicable, see Section 2.3.

Reusable Component Testing – Pros and Cons

- **Pro: Tests commonalities** – The technique tests the common features by testing components used as building blocks for products in the product line.
- **Pro: Usable** – The technique is a fully usable software and hardware product line testing technique: It scales, and free, open source algorithms and software exists for doing the automatic parts of the approach.
- **Pro: Agile** – The technique is agile in that changes to the feature model will not break the feature test suites. This allows the product line to evolve dynamically while still being able to test the commonalities independently of the feature model. In addition,

the technique is agile in that changes to a component might only cause its tests suite to stop functioning. Thus, when the internals of a feature changes, only its test suite might need modification. This allows the maintainers of a feature to evolve their feature without breaking the other parts of the product line test suite.

- **Con: Does not test interactions** – The major con of this approach is that it does not test interactions between features. It does not test whether the features will work in the presence of other features, whether it will cooperate correctly, or whether it works when other features are absent.

Combinatorial Interaction Testing – Pros and Cons

- **Pro: Usable** – The technique is a fully usable software and hardware product line testing technique It scales, and free, open source software exists for doing all the automatic parts of the approach.
- **Pro: Tests interactions** – The technique can find faults that can be attributed to interactions between features or that are caused by missing features. This is based on empirics by Kuhn et al. 2004 [94], and supported further by Garvin and Cohen 2011 [59] and Steffens et al. 2012 [146].
- **Con: Sensitive to changes** – The covering arrays generated may completely change if the feature model is changed. Because this is the first step in the application of the approach and because the other stages depends on it, the technique makes the product line rigid in that a change to a feature model causes the test suites of the product line to break.
- **Con: Manual** – The technique has automatic steps: Both the covering array and the products in it can be automatically generated, but then tests must be set up manually for each product in the covering array.

3.5.4 The Automatic CIT Technique

Recall that the goal of this part of the achievements is to set up a complete testing of a large-scale software product line. In order to achieve this within the time and resource constraints, the following simple technique was devised:

1. Generate a t-wise covering array (preferably 2 or 3-wise).
2. Automatically build each product.
3. For each product,
   (a) run the test suites that are related to its features.
   (b) run static analysis on the source code, if available.
4. Note the results in a table for analysis by domain experts.

In many cases, it was found that it is possible to determine the erroneous interactions by looking at when a test fails and when it succeeds. The difference between the products can give a definitive answer to this. In most cases, however, faults may overshadow each other. In those cases, a more detailed look at the error reports is needed.

The two applications of the technique are empirical evidence that not only is the technique applicable, but new bugs were also found without creating any new tests than the existing tests for the systems. How can a test that was not designed to be an interaction test cause interaction failures? Simply because if a feature (or a feature combination) succeeds in most cases but fails in the presence or absence of a certain other combinations of features then the failure can be attributed to an interaction between the feature (or features) being tested and that other combination.

3.5.5 Application to the Eclipse IDEs

As was the motivation, we did a complete application of the technique on the Eclipse IDEs v3.7.0 (Indigo), the 22 features supported by the Eclipse Project plus (3) additional features commonly used.

To speed things up, we created a local mirror of the Eclipse v3.7.0 (Indigo) repository totaling 3.6GiB that would serve as a cache. We downloaded 37 existing test suites with a total of 40,744 unit tests from four feature projects.

We scripted the entire technique to be applied in total at the click of a button. These scripts are documented in Appendix E and Section D.3.5[2].

When run, the technique produced 417,293 test results among which 513 were some kind of failures. All test suites succeeded for at least one product. This indicated that there was some interaction problem occurring at some stage in the testing process. The running of the technique a single time on one machine took about 23 hours. With 13 nodes, this would have been reduced to about 2 hours. The maximum disk space spent was 11 GB.

This case study was not completely documented in Paper 5 due to space constraints. Those details are available in this thesis. Appendix E contains the long version of the application outlined here.

3.5.6 Application to ABB's Safety Modules

We also did an application of the technique on a simulated version of the ABB safety module. Two bugs were identified by the creator of the simulated version of this safety module. Unfortunately, due to time limitations, we did not get to apply the technique to the real safety module. It was not available to us at SINTEF. According to ABB, the safety module did have a large collection of test cases. The Eclipse IDE case can indicate how an application would look like.

The ABB Safety Module is a physical component that is used in, among other things, cranes and assembly lines, to ensure safe reaction to problematic events, 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 thesis—for experimenting with testing techniques. It is, unfortunately, only this version of the ABB Safety Module which testing is reported in this thesis.

Figure 3.3 shows the feature model of the ABB Safety Module. The author of this thesis did not participate in the creation of the simulated version of the ABB Safety module. The feature model in the paper is in the FeatureIDE syntax and is slightly different. The reason is that the model used in Paper 5's experiment was an earlier version.

[2]They are also available on the Paper 5 resource page

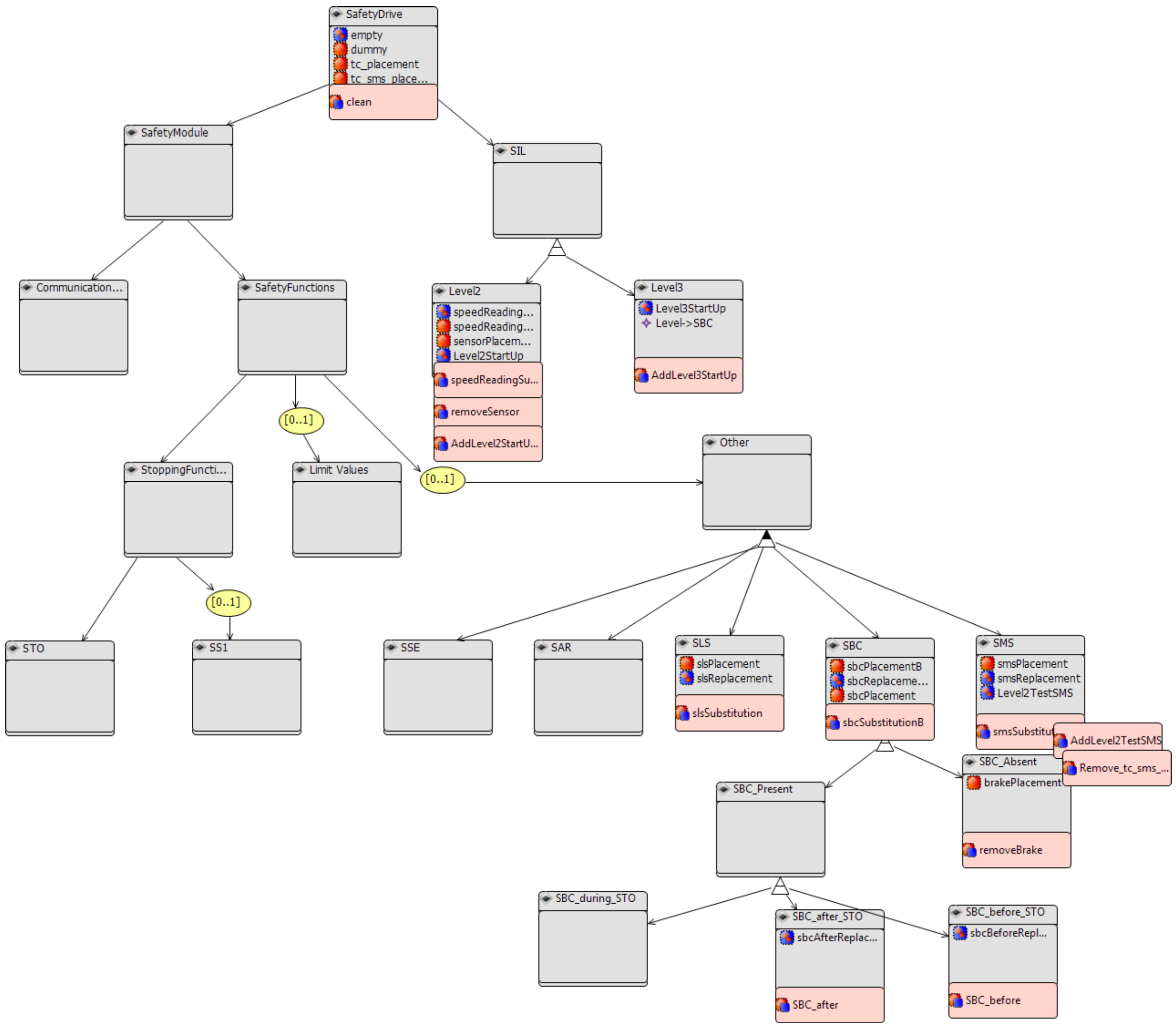


Figure 3.3: Feature Diagram of the ABB Safety Module in CVL 1 Feature Diagram Notation

Table 3.2: Test Cases for the Simulated ABB Safety Module

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

Table 3.3: Two Variants of the ABB Safety Module Named by the Context of its Use

Feature\Product	Conveyor Belt	Hoisting Machine
SafetyDrive	X	X
SafetyModule	X	X
CommunicationBus	X	X
SafetyFunctions	X	X
StoppingFunctions	X	X
STO	X	X
SS1	X	X
Limit_Values	-	-
Other	X	X
SSE	X	X
SAR	-	-
SLS	-	-
SBC	-	X
SBC_Present	-	X
SBC_during_STO	-	-
SBC_after_STO	-	-
SBC_before_STO	-	X
SBC_Absent	-	-
SMS	X	-
SIL	X	X
Level2	X	-
Level3	-	X

The product line is developed using the CVL 1 tools. The Safety Module simulation is implemented in UML with action code written in Java. The interface towards UML in Java was made possible with JavaFrame [70], a target framework for code generation from UML.

Five test cases were specified using sequence diagrams. These test cases are listed in Table 3.2 along with the feature they primarily exercise.

There are in total 512 possible configurations of the feature model. Two of these are specified in Table 3.3, named after the context these specific configurations of the safety module are used in. These products are, of course, valid configurations of the feature model of the ABB Safety Module, Figure 3.3.

The ABB Safety Module was used as a case study in Paper 5, in which most details are included[3]. Table 3.4a shows the 11 products of the 2-wise covering array of the ABB-case. Table 3.4b shows the result of running the relevant test cases on these 11 products. The four gray columns are products that did not build or start. All these four were due to a fault in the CVL-model. The gray cell of Product 9 indicates a failed test case. This was due to a dependency that was missing in the product line specification.

[3]A resource page was set up that describes how to reproduce the experiments described in the paper.

Table 3.4: 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	

3.6 Tool Support and Integration of Achievements

All of the above achievements are backed by open source tool support. The user manual for the tools runs through an application to the Eclipse IDEs utilizing most of the contributions discussed. The user manual with the integrated example is available as Appendix D. It is followed by a detailed discussion of the application of Automatic CIT to the Eclipse IDEs in Appendix E.

Chapter 4

Discussion

This chapter first discusses threats to validity for the contributions in this thesis (Section 4.1) and then discusses potentials and ideas for future work (Section 4.2).

4.1 Limitations and Threats to Validity

The following four subsections include threats to validity for the contributions. It also covers threats to validity from the perspective of product line testing as a whole.

Generation of Covering Arrays from Large Feature Models

The ICPL algorithm was a major result of Paper 1 and 2. Several threats to validity can be noted for these two papers: ICPL was evaluated by a comparison with three other algorithms for covering array generation. The authors of ICPL are also those who performed the comparison with the other algorithms. Below we discuss things that might have influenced the result of the comparison as presented in Paper 2.

- **Corpus Selection** – The feature model corpus consists of feature models not made by those who wrote or contributed to the contributions of this thesis. They are all used as originals except for some minor changes: e.g. slight changes to the IDs due to differences in what characters an ID can consist of.

  ICPL was tuned to perform well on the corpus, while the other algorithms were presumably developed without knowledge of these particular models. It would have been better to compare all algorithms of a set of feature models none of the developers knew about in advance.
- **Corpus Completeness** – No feature models were excluded from the corpus except for their lack of realism. All feature models the authors were aware of and could verify the origin of were included in the corpus within the time available for conducting the experiment.

  It is a possibility that the measurements are different given a more extensive corpus. A more extensive corpus might be required to be representative of realistic feature models in general.

- **Adaptors** – Some of the tools came with limited documentation. The adaptors that convert from the three formats of feature models in our corpus to input for each of the three other algorithms were written for the experiment in Paper 2 and are available as open source. The results produced by each algorithm were confirmed to be complete and valid by the same, separate algorithms.

  It is, however, possible that the adaptors are unfair to the other algorithms, and that the algorithms would have performed better with more suited converters. Had they been available from the implementations, they would have been used instead.
- **Other Algorithms** – There are other algorithms that were not included in the comparison. This issue was discussed in Section 2.4.4.
- **Other Values of t** – Empirics were not collected for t-wise testing of $t >= 4$. One of the reasons for this is that the sizes of the covering arrays, using any algorithm, then becomes significantly larger than the number of features in the feature model from which it is generated. A second reason is that empirics indicate that 95% of bugs can be attributed to the interaction of 3 features [94]. If the quality requirements are above this level, it is not yet established that combinatorial interaction testing is the approach that should be taken in the first place.

Optimized Testing in the Presence of Homogeneous Abstraction Layers

What are the liabilities in applying our techniques for reapplying test suites for products of a covering array that differ only in the implementation of homogeneous abstraction layers?

- The solution does not make sense if the abstraction layer contains the union of functionality of the implementations. The intersection has to be considerable in order for the application of this technique to pay off.
- The abstraction layer cannot be too general. For example, the team provider functionality in Eclipse has multiple implementations: CVS, SVN, git, etc, but the functionality the implementations offer is different. Thus, one have to supply one test suite per provider, and that is also what is done in the Eclipse project.
- If one decides to make available implementation-specific behavior in the abstraction layer, then the benefits of the pattern no longer applies, and one will have to significantly change the test suites.
- Even though the test cases might be the same for all implementations, the initializations of the test suites might be different.

Market-Focused Testing based on Weighted Sub-Product Line Models

A discussion of the limitations and threats to validity were not included in Paper 4 due to space constraints.

- **Partial Coverage** – The theory of combinatorial interaction testing deals with the benefit of 100% coverage. If a certain 95% 3-wise coverage is a 100% 2-wise coverage, does it then have the bug-detection capabilities of 3-wise testing, or just that of 2-wise testing? TOMRA wanted an answer to this question during our cooperation, but, unfortunately, we were unable to offer any answer to this. This is a question for future work.
- **Better than Manual Configuration by Domain Experts** – One of our findings in the experiments done at TOMRA was that the manually configured products were quite good. If this is the case, then why do we need theory and automation for doing what they already accomplish manually? Indeed, maintaining the feature model and the weighted sub-product line models are not free. Thus, even though the algorithms are fast once these models are up-to-date, the task of maintaining them might be slower than just manually updating the test lab by hand.
- **Inflexibility** – Configuring products by hand has a lot of flexibility. The domain experts can choose to focus on whatever factors they want. They can react to new requirements quickly. A fixed algorithm such as those related to the weighted sub-product line models might not cater for any need they might have. This inflexibility might be a sign the theory is not comprehensive enough yet and thus not ready for production.
- **Difficulty** – Understanding the theory of combinatorial interaction testing is not difficult; however, it does require at least some days of training. This must be recognized by anyone who will work with or think about it. This is in contrast with the apparent simplicity of the covering arrays. They are seemingly just a handful of products. The computational power required to generate them, however; is testament to the complexity that is packed into those simple results.

Agile and Automatic Product Line Testing

Automatic CIT was never meant to be a final solution for product line testing but rather a complete technique that could form the basis of further improvements or the basis of comparison. These are some possible issues with the technique as proposed by us:

- **Feature Interactions** – Feature interactions are behavior that is caused by the combination of two or more features. Our technique does not specifically test that a feature interaction between t features (for t-wise testing) interacts correctly with even more 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; however, Automatic CIT is not fully automatic when the products must be set up manually.
- **Quality of the Automatic Tests** – The quality of the results of the technique is dependent on the quality of the automatic tests that are run for the features of the products.
- **Feature Interaction Testing** – A problem within the field of feature interaction testing is how to best create tests as to identify interaction faults that occur between two or more concrete features, *the feature interaction problem* [20]. 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.

4.2 Future Work

4.2.1 Empirical Investigations

Empirical investigations were important for the contributions of this thesis. The work by Kuhn et al. 2004 [94] form the basis for the arguments in favor of combinatorial interaction testing. Their investigations are, however, not primarily targeted towards product line testing. Further work on such empirical investigations has been carried out by Garvin and Cohen 2011 [59] and Steffens et al. 2012 [146].

The three largest feature models in our collection were presented in and provided by She et al. 2011 [141]. Their extraction of these feature models provided the basis for the development of ICPL.

Tartler et al. 2011 [154] and related publications have done interesting experiments on static analysis and testing of the Linux Kernel that have resulted in identifications of serious bugs latent in some configurations.

Empirical studies of open source products lines by Berger et al. 2012 [13] show that many open source product lines are developed using Kconfig. They note that "Kconfig models are distributed over multiple files, organized according to the source code hierarchy." This aspect is not usually incorporated into variability models from research.

Our industrial partners were surprised to be confronted with the claim that because they have a product line, they must also have a problem with faults popping up when a new product is configured. They did not experience this as a serious problem; they were not convinced that a lot of effort was needed.

Now, the reason for this might be that they were unaware of the relation between a fault and product line engineering—that a fault was due to the new configuration triggering some latent fault. They might have thought that the fault is an ordinary fault and not a member of some new category of faults caused by the new factors introduced with product line engineering. Indeed, neither of the companies or the open-source case study describe their development practice as product line engineering although that certainly is what they are doing.

Further technical advancements in the theory of product line testing depends on what is found in the empirical investigations, as there is still significantly more to be learned from them.

For example, an empirical investigation of latent faults in product lines would be valuable. Such an investigation could either tell us that such faults are actually not a problem, or they would give us a solid argument as to why product line testing is worthy of industrial attention. Our industrial experiences gave us a clue that the actuality *might* be closer towards the former than currently thought.

4.2.2 Complex Product Lines

Simple product lines are configured by a single feature model, its products are distributed evenly around its configuration space, and the building of the products is done automatically by a click of a button. A complex product line, in this context, means a product line with additional challenges beyond such simple product lines. These complexities are such that techniques developed for the simple product line might not address the complexities in a suitable manner.

Mixed Hardware and Software Product Lines

TOMRA's product line of RVMs is a good example of a product line with both hardware and software variability. In our work with TOMRA, we focused on the hardware variability. The products of hardware product lines are naturally more difficult to build automatically: The assembly might require heavy-duty tools and expertise, or expensive robots. However, the software variability, although linked with the hardware variability, is easy to automatically build using software tools.

Whereas hardware often requires slow manual tests to be performed, software can be quickly tested.

The basic question here is how to balance the hardware and software testing to maximize the fault-detection of the testing effort?

In addition, a single, fixed hardware configuration can still have its software configuration automatically reconfigured. This brings up interesting questions in regards to covering array generation optimized to respect this possibility.

Mixed-Variability Systems

Another kind of concern is a product line of which only a few products configured and sold, but that still leaves a wide range of configurability for the customer or user. This lessens the product line developers' ability to check or know about any particular configuration. It actually becomes more important to exercise the user-variability, while at the same it might seem less important because only a few products are configured and sold by the developer.

In these cases, variability must be classified as more or less prone to being configured, something that is not possible to specify using today's feature models.

High Commonality, Few Products

Some product lines have a certain characteristic: They have a variability model, the products have a high degree of commonality and there is a huge configuration space; yet, due to the market situation and the nature of the product line, only a few products are configured and available for purchase. The company derives benefits from using product line engineering, but, when they have tested their (say six) products, they have tested every product they will sell for the foreseeable future.

In these situations, combinatorial interaction testing does not make sense. The technique derives it benefit from exercising every simple interaction. This is a waste if most of them will never be used. Still, because of the high degree of commonality, it is wasteful to test every system independently from scratch.

There should be a term to differentiate these kinds of product lines. We suggest *highly, moderately and sparsely realized product lines*, for lower degrees of realization, respectively.

Thus, we can ask the question: How should sparsely realized product lines be tested? Is it essentially different from testing highly realized product lines? Are there special techniques for each degree of realization?

Optimized Distribution of Test Cases

In our work on automatically testing the Eclipse IDEs or ABB's Safety Modules, we assumed that all test cases should be executed when they exercise relevant functionality in a product. This might, however, be wasteful. This is especially a concern when manual testing is done.

The basic question is: Given that a test case exercises a set of functionality, on which products of a covering array should the test be performed, or, maybe more importantly, where should they not be performed?

4.2.3 Product Line Engineering

Some opportunities for future work relate to product line engineering. These are opportunities that are linked to novelties and technologies in product line engineering as such that might benefit testing.

Distributed Variability Specifications

The Eclipse Project does not model their variability in a centralized way with a single feature model; they primarily model their variability with distributed fragments throughout the system. They do, however, for the purpose of automatic building, extract what is essentially a feature model from the distributed fragments.

This is also observed in the Linux kernel project and in eCos. In both cases, the variability of a component or a package is specified locally to that component or in that package.

Is testing systems with a distributed variability specification essentially different from testing systems with a centralized variability specification? Are there special properties that can be exploited in either?

One particular thing that is made easy using distributed variability specifications is dependencies on components within a range of versions. This is difficult in basic feature models because it would quickly bloat the model making it difficult to make out what it visualizes. This is commonly solved in, for example, the Eclipse variability specifications by having the variability specifications of each version separately specified.

Utilizing Implementation Information

A basic feature model is a simple model, and that has its benefit. It is essentialized, captures variability information and is easy to work with using SAT-solvers and analyzers.

One way to keep feature models simple, but still enhance their powers, is to extract useful information from the implementation of the product line. We explored one aspect of this in our work on utilizing homogeneous abstraction layers. There we realized that some alternative features are alternative because of the presence of a homogeneous abstraction layer. This was then further used in generating covering arrays and testing the products in it.

We suspect, however, that there are more such things that can be exploited.

Variability models such as CVL, delta-oriented programming and delta modeling can specify variability on a very fine level of granularity. Variability realization using preprocessors are popular for the Linux kernel and other open source systems. Can these fine-grained, or other coarse-grained variability realizations, give us additional power when testing?

If such things are not automatically found, maybe they can be annotated on the feature models directly by the designers of the product line? In our work, we annotated basic feature models with a "Homogeneous Abstraction Layer"-annotation. This annotation allows tool-support for marking products that are the same except for the implementations of these layers. Are there other such useful annotations that can give us further tool support for testing?

Design Principles and SPL Testing

Before it makes sense to do quality assurance, a product line must be well designed. A good design is a way to avoid problems early in development. When this is in place, however, testing can be used to validate the system further.

There are a wide range of design principles and patterns in software engineering literature. Do some of these make the product line easier to test and validate? For example, the Wrapper-Facade design pattern [138] encapsulates platform specific, non-object-oriented functionality in a class. This is obviously useful to promote variability of a product line over a variety of platforms. As a second example, the Release-Reuse Equivalency Principle [104] states that the granule of reuse should be the granule of release. How does this apply in the context of product line engineering? How does this level of granularity affect testing?

Questions such as these can be made for many of the relevant principles and patterns.

Exploiting Version Information

Dependencies in the Eclipse IDEs are not just to another feature but also to a range of feature versions. Such versioned dependencies are difficult to model in a basic feature model and difficult to graphically visualize.

More importantly however, how do the versions give us information about how to test the system? Maintaining old versions are important for several reasons. A range of versions might be installed in offline systems. In addition, older versions might need maintenance because of customers who do not need or want to upgrade. As with homogeneous abstraction layers, maybe they can be turned into an advantage?

4.2.4 Algorithms

The opportunities in this subsection are related to algorithmic challenges.

Further Improvements of ICPL

The ICPL algorithm was a significant improvement over previous algorithms for t-wise covering array generation. We did not, however, rule out further improvements. Can the algorithms be parallelized further? Complete parallelization would enable large clusters to efficiently run it without an upper bound. In addition, can the memory requirements due to the storage of all t-sets be reduced? Is it possible not to store all t-sets? There are many possibilities for further improvements of ICPL.

Upper bound for ICPL

Chvátal's algorithm is a greedy approximation algorithm for a problem classified as NP-complete by complexity analysis. A good upper bound was established in Chvátal 1979 [31]. This upper bound does not apply for ICPL because not all possible configurations are browsed. Instead, another greedy step is added to construct a single configuration. It would be interesting to know what is the upper bound guarantee of ICPL (or a similar algorithm), or, alternatively, whether ICPL can be modified to give it the upper bound guarantee of Chvátal's algorithm.

Extracting Basic Feature Models

This thesis deals with basic feature models. There are other kinds of feature models with more complex mechanisms. The work presented in this thesis still applies to more complex feature modeling mechanisms. Many types of feature models are more complex than basic feature models. Some promising techniques for testing product lines require a basic feature model; thus, these techniques cannot be used if a product line is not modeled with a basic feature model.

Many, more complex types of feature models have been proposed since the introduction of the notion of a feature model. Many of the powerful mechanisms have been included in the proposed standard variability language by the OMG, CVL [71]. This standard also introduces additional complex mechanisms to model variability: non-Boolean Variability, non-Boolean constraints, cardinality-based feature models [45], recursive variability, contextual constraints, OCL Constraints, etc.

Enabling the extraction of a basic feature model from a more complex type of feature model for the purpose of testing would be an interesting topic. This could be both manual and automatic methods to be done by a test engineer. A solution would enable the application of some promising techniques for testing product lines when a product line is modeled using a more complex type of feature model. The test engineer may, however, in some situations, have to conclude that extracting a basic feature model for the purpose of testing is simply not purposeful.

Evolving Covering Arrays

When a software product line evolves, the products of an old covering array might no longer cover the simple interactions of the product line. Thus, a new test suite must be generated. Currently, no algorithm exists for minimizing the difference between the old and new test products, causing the new test suite to be unnecessarily different from the old test suite. This causes unnecessary effort in co-evolving a test suite with an evolving product line.

It would be good to have an efficient algorithm for incremental evolution of test products with an evolving product line. This would reduce effort in maintaining a test suite when a product line evolves. The efficiency could be demonstrated with an experiment on realistic feature models that were realistically evolved. Well-documented evolutions exist for, for example, the Linux kernel and the Eclipse IDEs.

4.2.5 Engineering

These opportunities are not so much related to novel research, but might spawn some interesting research challenges that need solving. At the face of it, these challenges are currently demanding engineering tasks.

Using Virtual Machines

While the testing of a system running on a single platform can be carried out on that platform, testing systems for various platforms in a uniform and automatic way cannot be done easily on a single platform. Today, good virtual machine technology exists. This enables the execution and control of systems running in other operating systems and hardware architectures.

To fully test, for example, the Eclipse IDEs or the Linux kernel, a variety of virtual machines could be automatically set up with the right configuration. Eclipse and Linux could then be automatically installed on these virtual machines and tested.

Full Testing of the Eclipse IDEs

Setting up a full-scale production test of the Eclipse IDEs should be possible given the experiments carried out in Paper 5 of this thesis. This requires some engineering work, some domain experts, hardware or virtual machines.

We think that such an application would be an interesting case study from which we could learn something about large-scale product line testing in practice.

It could also be a demonstration worth looking at for other developers of industrial product lines for whether they should adapt such techniques for their product line. In the end, a demonstration that something actually works on a large-scale is an argument you cannot ignore.

Chapter 5

Conclusion

This chapter concludes Part I of the thesis by summarizing the results and their main evaluations.

5.1 Results

In this thesis, we contributed to the testing of product lines of industrial size by advancing combinatorial interaction testing (CIT).

Existing applied techniques such as reusable component testing, does not exercise feature interactions. There are many suggestions of how to test feature interactions. Based on the background and related work, and taking into account the industrial case studies, combinatorial interaction testing (CIT) emerged as the most promising approach. The primary reasons were that 1) the technique does address the issue of interaction testing and 2) the technique can be applied within the current work-flows established; it does not require a significant change in development practices. This enables us to, for example, apply the technique of Contribution 5 to the entire part of the Eclipse IDE product line supported by the Eclipse project with a small workforce.

Contribution 1 and 2 culminated in the ICPL algorithm. ICPL is a covering array generation algorithm that can generate covering arrays for the larger product lines. In particular, it was the first algorithm to generate a pair-wise covering array for the Linux Kernel feature model, one of the largest product lines whose feature model is available.

Having a scalable algorithm for covering array generation encourages further advancements of combinatorial interaction testing for product lines. We contributed three advancements: Utilization of homogeneous abstraction layers for oracles, market-focused CIT and automatic CIT.

Contribution 3: Many product lines utilize homogeneous abstraction layers to get a uniform interface to a multitude of similar systems. We showed how such layers could be exploited to improve product line testing. In a covering array generated from a product line with such abstraction layers, test suites can be reused for products that differ only in the implementation of one or more homogeneous abstraction layers. Further, these repeated executions of the test suites allows for a voting oracle or taking one of the executions to be a gold standard oracle.

Contribution 4: Sub-product line models were contributed for modeling the market situation of a product line. Associated algorithms allow covering arrays to be adapted to, optimized for and evolved with the market situation. The market situation can model the customer priority or criticality. They can model the expectations and make testing of what will be the situation in the future.

Contribution 5: The automatic CIT technique filled in what was missing to make CIT automatic. Test for parts of a product line can be reused to test interaction among the features. Rerunning existing tests on each applicable product of the covering array of a product line does give us some insight into the simple interaction among the features. Such tests are often developed for product lines and are thus readily available even for existing product lines. This allowed us to, for example, apply the technique in full to the part of the Eclipse IDE product line supported by the Eclipse project.

Tool support: Tools were implemented with the contributed algorithms and techniques. These tools are available freely as open source under the EPL v1.0 license.

5.2 Evaluation

Contribution 1 and 2: The ICPL algorithm was evaluated by comparing it to three other algorithms from the research community and one of our previous algorithms. The five algorithms were executed on 19 existing, realistic feature models. The following was concluded from this evaluation:

ICPL generates covering arrays of an acceptable size, comparable to other state of the art algorithms. Even though it is non-deterministic, there is little to none variance between executions. It is faster than the other algorithms; for 2-wise testing, it was estimated to be 1,000 times faster than the second fastest algorithm for an industrially sized feature model. In addition, although MoSo-PoLiTe was the second fastest, its covering array sizes were larger than the other algorithms. For 3-wise testing, ICPL was found to be almost 30,000 times faster than the second fastest for a feature model of industrial size. Note that these factors grow as the feature model gains more features; however, the example here is one of the largest known feature models.

The three advancements of CIT were evaluated by the application of them to four industrial case studies; the open source system allowed us to document our results and provide reproducible results.

Contribution 3: Two of our case studies had homogeneous abstraction layers with which we could experiment with our related contribution. In the Finale case study, if we want to exercise all pairs of interactions, all products containing a version of client-side Windows paired with a version of an accounting system, then we need at least $6 * 18 = 108$ products. Indeed, the pair-wise covering array contains 116 products. If we group together those products that only differ below a homogeneous abstraction layer, we reduce the number to 60. Unfortunately, details of this experiment are unavailable for documentation.

To provide a documented experiment, we did apply the technique to the Eclipse IDEs. In total, they support 16 combinations of hardware, windowing systems and operating systems. The pair-wise covering array of a certain sub-set of the Eclipse IDEs was 41 products. Out of these, only 20 groups of products need to have unique test suites. Internally in the groups, the products differ only beneath the homogeneous abstraction layers. Covering the system with 1-wise testing gives 10 products, but only 5 test suites are required. For 3-wise testing, with 119 configurations, only 45 test suites are required.

Contribution 4: The main experiment with weighted sub-product line models and associated algorithms was the TOMRA case study. This means that the experiments are not reproducible. The feature model of the TOMRA case has 68 features (which yield 435,808 possible configurations) TOMRA's existing lab had 12 products manually configured by domain experts. The experiments centered on these products. We found that: (1) The domain experts found that covering arrays generated from the weighted models were more familiar to them and were closer to products found in the market. (2) Their existing lab had higher weight coverage than t-set coverage. (3) Significantly fewer products were needed than they currently had to achieve the same weight coverage. (4) Finally, the domain experts found the suggestions given by the incremental evolution algorithm to be good and relevant.

We supplemented this commercial case with an application to the Eclipse IDEs. It was simpler, but it is at least documented openly and reproducibly. Based on download statistics of the 12 Eclipse IDE products offered for download, we found that 4 products are needed to achieve 95% weight coverage.

Contribution 5: The full Automatic CIT technique was applied to two industrial product lines: First of all, it was applied to a simulated version of the ABB safety module. The feature model had 22 features and yielded 640 possible products in the original experiment (512 in a corrected version). Of these configurations, the pair-wise covering array consisted of 11 products that when tested with Automatic CIT produced five test failures. These failures revealed two bugs as identified by the creators of the simulated version of the safety module.

Automatic CIT was also applied to the part of the Eclipse IDEs supported by the Eclipse Project. This is one of the largest documented applications of a product line testing technique. We created a feature model for the part of the Eclipse IDEs supported by the Eclipse Project. The feature model consisted of 22 features. We implemented Automatic CIT for the Eclipse Platform plug-in system, and we created a feature mapping for 36 existing test suites. This experiment consisted of 40,744 tests and resulted in 417,293 test results. Out of these, 513 failures were produced that are probably caused by interaction faults.

Bibliography

[1] ABB. ABB industrial drives - Safety functions module for extending application safety. `http://www05.abb.com/global/scot/scot201.nsf/veritydisplay/cff4a15410027401c1257aef003b7bc7/$file/16645_FSO-11_flyer_EN_3AUA0000116739_RevB_lowres.pdf`, 2013.

[2] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[3] Sergej Alekseev, Rebecca Tiede, and Peter Tollkühn. Systematic approach for using the classification tree method for testing complex software-systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, SE'07, pages 261–266, Anaheim, CA, USA, 2007. ACTA Press.

[4] Finale Systemer AS. Produktkatalog. `http://www.finale.no/produktkatalog.4828511-165509.html`, 2013.

[5] Automated Combinatorial Testing for Software group at NIST, Arlington, Texas, US. *User Guide for ACTS*, beta 2 - revision 1.5 edition, December 2011.

[6] J. Bach. Allpairs test case generation tool (version 1.2.1), May 2013.

[7] Don Batory. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1:355–398, 1992.

[8] Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. *SIGPLAN Not.*, 47(3):13–22, October 2011.

[9] Don Batory, Jacob N. Sarvela, Axel Rauschmayer, Student Member, and Student Member. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30, 2003.

[10] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(6):2004, 2004.

[11] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[12] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the systems software domain. Technical Report GS-DLAB–TR 2012–07–06, Generative Software Development Laboratory, University of Waterloo, 2012. journal submission under review.

[14] Antonia Bertolino, Alessandro Fantechi, Stefania Gnesi, and Giuseppe Lami. Product line use cases: Scenario-based specification and testing of requirements. In Käkölä and Dueñas [90], pages 425–445.

[15] Antonia Bertolino and Stefania Gnesi. Use case-based testing of product lines. *SIGSOFT Softw. Eng. Notes*, 28(5):355–358, 2003.

[16] Antonia Bertolino and Stefania Gnesi. Pluto: A test methodology for product families. In van der Linden [162], pages 181–197.

[17] Armin Biere and Wolfgang Kunz. Sat and atpg: Boolean engines for formal hardware verification. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 782–785, New York, NY, USA, 2002. ACM.

[18] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] Jan Bosch and Jaejoon Lee, editors. *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*. Springer, 2010.

[20] T.F. Bowen, FS Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.J. Lin. The feature interaction problem in telecommunications systems. In *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*, pages 59–62. IET, 1989.

[21] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[22] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In Bosch and Lee [19], pages 241–255.

[23] Andrea Calvagna and Angelo Gargantini. A logic-based approach to combinatorial testing with constraints. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-79124-9_6.

[24] Andrea Calvagna and Angelo Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In Catherine Dubois, editor, *Tests and Proofs*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer Berlin / Heidelberg, 2009.

[25] Andrea Calvagna and Angelo Gargantini. Ipo-s: Incremental generation of combinatorial interaction test data based on symmetries of covering arrays. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:10–18, 2009.

[26] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45:331–358, 2010. 10.1007/s10817-010-9171-4.

[27] Andrea Calvagna, Angelo Gargantini, and Paolo Vavassori. Combinatorial interaction testing with citlab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, Proceedings of*, 2013.

[28] Andrea Calvagna, Angelo Gargantini, and Paolo Vavassori. Combinatorial testing for feature models using citlab. In *The 2nd International Workshop on Combinatorial Testing (IWCT 2013), Proceedings of*, 2013.

[29] Andrea Calvagna, Giuseppe Pappalardo, and Emiliano Tramontana. A novel approach to effective parallel computing of t-wise covering arrays. In *Proceedings of the 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '12, pages 149–153, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Philip Chong and Mukul Prasad. Satisfiability for atpg: Is it easy?, 1998.

[31] Václav Chvátal. A greedy heuristic for the Set-Covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[32] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based coverage-driven test suite generation for software product lines. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 425–439, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. *SIGPLAN Not.*, 46(2):13–22, October 2010.

[34] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, January 2000.

[35] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient tests generator. In *Fifth IEEE International Symposium on Software Reliability Engineering, Proceedings of*, page 303–309, 1994.

[36] David Cohen, Ieee Computer Society, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[37] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Softw.*, 13(5):83–88, September 1996.

[38] Myra B. Cohen. *Designing test suites for software interaction testing*. PhD thesis, The University of Auckland, 2004.

[39] Myra B. Cohen, Charles J. Colbourn, and Alan C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.

[40] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan 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*, pages 53–63, New York, NY, USA, 2006. ACM.

[41] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan 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.

[42] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 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.

[43] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *ICSE*, pages 38–48. IEEE Computer Society, 2003.

[44] Charles J. Colbourn and Myra B. Cohen. A deterministic density algorithm for pairwise interaction coverage. In *Proc. of the IASTED Intl. Conference on Software Engineering*, pages 242–252, 2004.

[45] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[46] Jacek Czerwonka. Pairwise testing in real world. In *Proceedings of 24th Pacific Northwest Software Quality Conference, 2006*, 2006.

[47] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.

[48] DIMACS. Satisfiability Suggested Format. Technical report, Center for Discrete Mathematics and Theoretical Computer Science, 1993.

[49] Michael Dukaczewski, Ina Schaefer, Remo Lachmann, and Malte Lochau. Requirements-based delta-oriented spl testing. In *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, 2013.

[50] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Inf. Softw. Technol.*, 52(1):14–30, January 2010.

[51] Erich Gamma. How (7 years of) eclipse changed my views on software development. `http://qconlondon.com/dl/qcon-london-2008/slides/ErichGamma_qcon2008.pdf`, 2008.

[52] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[53] Dharmalingam Ganesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In *Proceedings of the 11th International Software Product Line Conference*, pages 74–83, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Dharmalingam Ganesan, Mikael Lindvall, David McComas, Maureen Bartholomew, Steve Slegel, Barbara Medina, Rene Krikhaar, and Chris Verhoef. An analysis of unit tests of a flight software product line. *Science of Computer Programming*, 2012.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[56] Angelo Gargantini and Paolo Vavassori. Citlab: A laboratory for combinatorial interaction testing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 559–568, Washington, DC, USA, 2012. IEEE Computer Society.

[57] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE 2011)*, November 2011.

[58] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. *Search Based Software Engineering, International Symposium on*, 0:13–22, 2009.

[59] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Softw. Engg.*, 16:61–102, February 2011.

[60] Hassan Gomaa. *Designing software product lines with UML - from use cases to pattern-based software architectures*. ACM, 2005.

[61] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra Cohen. Interaction coverage meets path coverage by smt constraint solving. In Manuel Núñez, Paul Baker, and Mercedes Merayo, editors, *Testing of Software and Communication Systems*, volume 5826 of *Lecture Notes in Computer Science*, pages 97–112. Springer Berlin / Heidelberg, 2009.

[62] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[63] O. Gruber, BJ Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Systems Journal*, 44(2):289–299, 2005.

[64] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta modeling for software architectures. In *In MBEES*, 2011.

[65] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[66] Alan Hartman. Software and hardware testing using combinatorial covering suites. In MartinCharles Golumbic and IrithBen-Arroyo Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.

[67] Jean Hartmann, Marlon Vieira, and Axel Ruder. A UML-based approach for validating product lines. In Birgit Geppert, Charles Krueger, and Jenny Li, editors, *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 58–65, Boston, MA, August 2004.

[68] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using feature model knowledge to speed up the generation of covering arrays. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 16:1–16:6, New York, NY, USA, 2013. ACM.

[69] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[70] Øystein Haugen and Birger Møller-Pedersen. Javaframe: Framework for java-enabled modelling. In *Proceedings of Ericsson Conference on Software Engineering*, 2000.

[71] Øystein Haugen et al. Common Variability Language (CVL)—OMG Revised Submission, August 2012. `http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf`.

[72] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. *CoRR*, abs/1211.5451, 2012.

[73] A. Hervieu, B. Baudry, and A. Gotlieb. Pacogen: Automatic generation of pairwise test configurations from feature models. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 120–129. IEEE, 2011.

[74] IEEE. *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[75] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[76] Bob Jenkins. jenny: a pairwise testing tool, May 2013.

[77] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. A Survey of Empirics of Strategies for Software Product Line Testing. In Lisa O'Conner, editor, *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, pages 266–269, Washington, DC, USA, 2011. IEEE Computer Society.

[78] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In Jon Whittle, Tony Clark, and Thomas Kuehne, editors, *Proceedings of Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*, pages 638–652, Wellington, New Zealand, October 2011. Springer, Heidelberg.

[79] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In Vander Alves and Andre Santos, editors, *Proceedings of the 16th International Software Product Line Conference (SPLC 2012)*. ACM, 2012.

[80] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Bow Tie Testing - A Testing Pattern for Product Lines. In Paris Avgeriou, editor, *Proceedings of the 16th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2011)*. ACM, 2012.

[81] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A Technique for Agile and Automatic Interaction Testing for Product Lines. In Brian Nielsen and Carsten Weise, editors, *Proceedings of The 23rd IFIP International Conference on Testing Software and Systems (ICTSS'12)*, pages 39–54. Springer, 2012.

[82] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In Robert France, Juergen Kazmeier, Colin Atkinson, and Ruth Breu, editors, *Proceedings of Model Driven Engineering Languages and Systems, 15th International Conference, MODELS 2012*, Innsbruck, Austria, September 2012. Springer.

[83] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition*. AUERBACH, 3 edition, 2008.

[84] Erik Kamsties, Klaus Pohl, Sacha Reis, and Andreas Reuys. Testing variabilities in use case models. In van der Linden [162], pages 6–18.

[85] Erik Kamsties, Klaus Pohl, and Andreas Reuys. Supporting test case derivation in domain engineering. In *Proceedings of the Seventh World Conference on Integrated Design and Process Technology (IDPT-2003)*. Society for Design and Process Science (SDPS), 2003.

[86] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[87] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 181–190, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[88] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[89] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[90] Timo Käkölä and Juan C. Dueñas, editors. *Software Product Lines - Research Issues in Engineering and Management*. Springer, Berlin, Heidelberg, 2006.

[91] Mohammad F. J. Klaib, Kamal Zuhairi Zamli, Nor Ashidi Mat Isa, Mohammed I. Younis, and Rusli Abdullah. G2way a backtracking strategy for pairwise test data generation. In *APSEC*, pages 463–470. IEEE, 2008.

[92] Ronny Kolb. A risk-driven approach for efficiently testing software product lines. In *Net.ObjectDays 2003. Workshops. Industriebeiträge - Tagungsband : Offizielle Nachfolge-Veranstaltung der JavaDays, STJA, JIT, DJEK*, pages 409—-414, 2003.

[93] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards efficient spl testing by variant reduction. In *Proceedings of the 4th international workshop on Variability & composition*, VariComp '13, pages 1–6, New York, NY, USA, 2013. ACM.

[94] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[95] Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, ECE Department, University of Waterloo, Canada, 2006.

[96] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 549–556. IEEE, 2007.

[97] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18(3):125–148, September 2008.

[98] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[99] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, New York, NY, 8 2013. to appear; accepted 30 May 2013.

[100] S. Lity, M. Lochau, I. Schaefer, and U. Goltz. Delta-oriented model-based spl regression testing. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pages 53–56, 2012.

[101] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM.

[102] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In Achim D. Brucker and Jacques Julliand, editors, *TAP*, volume 7305 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2012.

[103] Roberto E. Lopez-herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Proc. 2001 Conf. Generative and Component-Based Software Eng*, pages 10–24. Springer, 2001.

[104] R.C. Martin. Engineering notebook: Granularity. *C++ Report*, 8:57–57, 1996.

[105] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, School of Computer Science, University of Waterloo, Jan 2009.

[106] M. Mendonca, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009.

[107] A. Metzger, K. Pohl, S. Reis, and A. Reuys. Model-based testing of software product lines. In *In Proceedings of the 7th Intl. Conference on Software Testing (ICSTEST)*, 2006.

[108] Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2 edition, 2004.

[109] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

[110] Kari J Nurmela and Patric RJ Östergård. *Constructing covering designs by simulated annealing*. Helsinki University, Digital Systems Laboratory, 1993.

[111] Object Management Group (OMG). OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. Technical report, Object Management Group (OMG), August 2011.

[112] Object Management Group (OMG). OMG MOF 2 XMI mapping specification. Technical report, Object Management Group (OMG), August 2011.

[113] Erika Mir Olimpiew. *Model-based testing for software product lines*. PhD thesis, George Mason University, Fairfax, VA, USA, 2008. AAI3310145.

[114] S. Oster, A. Wübbeke, G. Engels, and A. Schürr. Model-Based Software Product Lines Testing Survey. In J. Zander, I. Schieferdecker, and P. Mosterman, editors, *Model-based Testing for Embedded Systems*. CRC Press/Taylor & Francis, 2010. to appear, accepted for application.

[115] Sebastian Oster. *Feature Model-based Software Product Line Testing*. PhD thesis, Technische Universität Darmstadt, 2012.

[116] Sebastian Oster. flattening algorithm, May 2013.

[117] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In Bosch and Lee [19], pages 196–210.

[118] Sebastian Oster, Florian Markert, Andy Schürr, and Werner Müller. Integrated modeling of software product lines with feature models and classification trees. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009). MAPLE 2009 Workshop Proceedings. Springer, Heidelberg*, 2009.

[119] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise feature-interaction testing for spls: potentials and limitations. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 6:1–6:8, New York, NY, USA, 2011. ACM.

[120] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: tool support for pairwise and model-based software product line testing. In Patrick Heymans, Krzysztof Czarnecki, and Ulrich W. Eisenecker, editors, *VaMoS*, ACM International Conference Proceedings Series, pages 79–82. ACM, 2011.

[121] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.

[122] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context awareness for dynamic service-oriented product lines. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 131–140, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[123] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 2011.

[124] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 459–468, Washington, DC, USA, 2010. IEEE Computer Society.

[125] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[126] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[127] The Linux Kernel Project. KBuild Documentation. `http://www.kernel.org/doc/Documentation/kbuild/`, 2013. [Online; accessed 30-10-2013].

[128] Sacha Reis, Andreas Metzger, and Klaus Pohl. A reuse technique for performance testing of software product lines. In *Proceedings of the SPLC International Workshop on Software Product Line Testing*, page 5–10, 2006.

[129] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. *Advanced Information Systems Engineering*, pages 519–534, 2005.

[130] Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. Derivation of domain test scenarios from activity diagrams. In Klaus Schmid and Birgit Geppert, editors, *Proceedings of the PLEES'03 International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, Erfurt, 2003.

[131] Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. The scented method for testing software product lines. In Käkölä and Dueñas [90], pages 479–520.

[132] J. Rivieres and W. Beaton. Eclipse Platform Technical Overview, 2006.

[133] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, January 2010.

[134] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Bosch and Lee [19], pages 77–91.

[135] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 43–56, New York, NY, USA, 2011. ACM.

[136] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. A model-based framework for automated product derivation. In *1st International Workshop on Model-Driven Approaches in Software Product Line Engineering Goetz Botterweck, Iris Groher, Andreas Polzer*, page 14, 2009.

[137] Kathrin Danielle Scheidemann. *Verifying families of system configurations*. Shaker, 2008.

[138] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[139] P. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering, 14th IEEE International Conference*, pages 139–148, 2006.

[140] Gerard Schouten. Philips medical systems. In *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, pages 233–248. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[141] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 461–470. ACM, 2011.

[142] George B. Sherwood, Sosina S. Martirosyan, and Charles J. Colbourn. Covering arrays of higher strength from permutation vectors. *Journal of Combinatorial Designs*, 14(3):202–213, 2006.

[143] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 72–77 vol.1, 2004.

[144] Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 176–183, Washington, DC, USA, 2004. IEEE Computer Society.

[145] J Stardom. *Metaheuristics and the Search for Covering and Packing Arrays*. PhD thesis, Simon Fraser University, 2001.

[146] Michaela Steffens, Sebastian Oster, Malte Lochau, and Thomas Fogdal. 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*, 1 2012.

[147] Vanessa Stricker, Andreas Metzger, and Klaus Pohl. Avoiding redundant testing in application engineering. In Bosch and Lee [19], pages 226–240.

[148] Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Analyzing variability: capturing semantic ripple effects. In *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, pages 253–269, Berlin, Heidelberg, 2011. Springer-Verlag.

[149] Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Using variability models to reduce verification effort of train station models. In *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC 2011)*, Ho Chi Minh City, Vietnam, 2011.

[150] Andreas Svendsen, Gøran K. Olsen, Jan Endresen, Thomas Moen, Erik Carlson, Kjell-Joar Alme, and Øystein Haugen. The future of train signaling. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2008.

[151] Andreas Svendsen, Øystein Haugen, and Xiaorui Zhang. Cvl 1.2 user guide. `http://www.omgwiki.org/variability/lib/exe/fetch.php?media=cvl1.2_user_guide.pdf`, May 2011.

[152] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and Gøran K. Olsen. Developing a software product line for train control: A case study of cvl. In Bosch and Lee [19], pages 106–120.

[153] Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1):109–111, 2002.

[154] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.

[155] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, , and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[156] TOMRA ASA. Helping the world recycle. `http://www.tomra.com/files/corporate_brochure_web2.pdf`, 2013.

[157] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 191–200, New York, NY, USA, 2006. ACM.

[158] Yu-Wen Tung and W.S. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431–437 vol.1, 2000.

[159] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[160] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309 –322, may-june 2010.

[161] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 249–258, Washington, DC, USA, 2008. IEEE Computer Society.

[162] Frank van der Linden, editor. *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*. Springer, 2004.

[163] Markus Voelter. Using domain specific languages for product line engineering. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 329–329, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[164] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 233–242. IEEE, 2007.

[165] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 211–220, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[166] Jules White, Brian Dougherty, Doulas C. Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 11–20, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[167] Alan W Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6*, volume 1, pages 59–74, 2000.

[168] Linbin Yu, Mehra Nourozborazjany, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *Proceedings of Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013.

[169] Kamal Z. Zamli, Mohammad F. J. Klaib, Mohammed I. Younis, Nor A. Isa, and Rusli Abdullah. Design and implementation of a T-Way test data generation strategy with automated execution tool support. *Information Sciences*, January 2011.

Part II

Research Papers

Chapter 6

Overview of Research Papers

Paper 1: Properties of Realistic Feature Models make Combinatorial Testing of Product Lines Feasible

Authors Martin Fagereng Johansen, Øystein Haugen and Franck Fleurey.

Publication Status Published in *Model Driven Engineering Languages and Systems - 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings* by Springer, 2011, pp. 638–652.

Contribution Martin Fagereng Johansen is the main contributor of this paper and has contribute to all parts of it (e.g. ideas, paper writing, tool implementation, all topics of the paper) responsible for 90% of the work.

Main Topics This paper presents an analysis of covering array generation. Covering array generation is usually classified as NP-complete on two levels using the techniques of the field of complexity analysis. The paper studies these two levels and finds that the second level, equivalent to the Boolean satisfiability problem, classified as an NP-complete problem, is easily solvable for the cases occurring in practice based on the nature of product line engineering. The first level is equivalent to the set cover problem (SCP), for which an approximation algorithm is known. Taken together, the paper concludes that the problem of covering array generation is tractable in practice for product lines.

To back up these claims, the theoretical consideration are demonstrated as applicable on a corpus of 19 feature models of realistic product lines. Their sizes are from only 9 features up to a gigantic 6,888 features and 187,193 clauses in the CNF constraint, one of the largest feature model known that is also in active use by both computers and human beings for configuring the Linux kernel. The satisfiability time of the largest feature model is only 125 ms.

Chvátal's algorithm for the set cover problem (SCP) is adapted for covering array generation from realistic feature models, and the speed of generating and the sizes of the resulting covering arrays for all 19 feature models, strengths 1–4, are presented, where the algorithm succeeds within a given time on a given computer. This clarifies where improvements can be made for an algorithm that handles covering array generation for large feature models.

Paper 2: An Algorithm for Generating t-wise Covering Arrays from Large Feature Models

Authors Martin Fagereng Johansen, Øystein Haugen and Franck Fleurey.

Publication Status Published in *SPLC '12: Proceedings of the 16th International Software Product Line Conference - Volume 1* by ACM, pp. 46–56.

Contribution Martin Fagereng Johansen is the main contributor of this paper and has contribute to all parts of it (e.g. ideas, paper writing, tool implementation, all topics of the paper) responsible for 90% of the work.

Main Topics The paper builds heavily on the MODELS 2011-paper. Based on the corpus of the same 19 realistic feature models and the adaption of Chvátal's algorithm for covering array generation for feature models, a new and significantly faster algorithm is proposed. The new algorithm is called ICPL (ICPL is a recursive acronym for "ICPL Covering array generation for Product Lines"). In the algorithm, several things are done to avoid redundant work, to learn from intermediate results and to do some things in parallel.

The algorithm is then compared to three of the major algorithms for generating covering arrays from feature models and the basic adaption of Chvátal's algorithm from the MODELS 2011-paper. All of the five compared algorithms were run 100 times for each of the 19 feature models, for each strength 1–3. The results show that ICPL produces small covering arrays that are acceptable in size but that it produce them orders of magnitude faster than the other algorithms. ICPL manages, as the only algorithm in the experiment, to produce 2-wise covering arrays, not only from the feature model with 287 features, as the second best does, but also from the feature models with 1,244, 1,396 and 6,888 features. For 3-wise covering arrays, the algorithm produces covering array for feature models with, not only 101 features, as the second best can, but also from the feature model with 287. In addition, 3-wise covering arrays, which cover only included features, are produced the feature models of 1,244 and 1,396 features.

Paper 3: Bow Tie Testing: A Testing Pattern for Product Lines

Authors Martin Fagereng Johansen, Øystein Haugen and Franck Fleurey.

Publication Status Published in *Pattern Languages of Programs - 16th European Conference, Proceedings* by ACM, pp. 9:1–9:13.

Contribution Martin Fagereng Johansen is the main contributor of this paper and has contribute to all parts of it (e.g. ideas, paper writing, tool implementation, all topics of the paper) responsible for 90% of the work.

Main Topics The paper presents a way to utilize knowledge of the structure of a product line in applying combinatorial interaction testing for testing product lines. If a product line has one or more homogeneous abstraction layers which is also modeled in the feature model as mutually exclusive alternatives, then the number of test suites required for a covering array generated for the product line can be reduced, and the same test suites can applied to several of the products with different implementations of the homogeneous abstraction layer. This allows not only a reduced effort in creating test suites but also a repeated application of the same test suite to form a voting oracle. The technique is demonstrated on a subset of the Eclipse IDE product line where, out of 41 products in the 2-wise covering array, only 20 test suites must be created, some of which can be applied to nine products, giving a considerable voting oracle.

Paper 4: Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines

Authors Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard and Torbjørn Syversen.

Publication Status Published in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings* by Springer, pp. 269–284.

Contribution Martin Fagereng Johansen is the main contributor of this paper and has contribute to all parts of it (e.g. ideas, paper writing, tool implementation, all topics of the paper) responsible for 80% of the work. Øystein Haugen contributed to the work on the feature model and Anne Grete Eldegard and Torbjørn Syversen contributed to work on the case study.

Main Topics This paper presents an application of combinatorial interaction testing to the TOMRA Reverse Vending Machine product line, consisting of hardware and software; the applications was only based on the hardware variability. In order to maximize testing efficiency at TOMRA, we developed a new kind of model called a weighted sub-product line model that models partially configured product lines with weights associated with them. These models were easy to make by domain experts and were close to their way of thinking about the current market situation. The new models allowed us to do several new and relevant things: First of all, it allowed us to create partial covering arrays that cover most of the weight, which means, most relevant according to the current market situation. Secondly, it allowed us to evaluate the current testing lab at TOMRA to establish that it is made with respect to the market situation but that it is not made as well as it could have been if made from scratch with the new approach. Lastly, an algorithm for suggesting small changes to a current test lab enabled incremental adaption of the test lab according to the changing market situation, modeled in the weighted sub-product line models.

Paper 5: A Technique for Agile and Automatic Interaction Testing for Product Lines

Authors Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen and Tormod Wien.

Publication Status Published in *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings* by Springer, pp. 39–54.

Contribution Martin Fagereng Johansen is the main contributor of this paper and has contribute to all parts of it (e.g. ideas, paper writing, tool implementation, all topics of the paper) responsible for 80% of the work. Erik Carlson, Jan Endresen and Tormod Wien contributed to the design of the simulation of the safety module and to the specifications of the test cases.

Main Topics The paper presents a new technique for testing product lines and applications of it on two industrial product lines. The technique is simple and easy to apply to existing product lines as opposed to the theories behind why it works and the explanation for the good results it produces. The technique consists of first automatically generating the products defined by a covering array of some strength and then applying to each product the test suites related to one or more of the included features. Executing these tests tests the features (or a combination of features) in the presence or absence of combinations of the other features.

The technique was applied to two industrial cases. In the first case, two actual bugs were identified. In the second case, many potential bugs were identified. This provides empirical evidence that not only is the technique applicable, but they also may find bugs *without* creating any new test than the existing tests for the systems. The second case was a complete application to the Eclipse IDE product line: 40,744 existing unit tests created 417,293 test results that, although passing for one distributed version of the Eclipse IDE, failed 513 times when applied to the 13 products of the 2-wise covering array of the Eclipse IDE.

Pseudo code for applying the technique to both CVL-based and Eclipse-based product lines is shown; they are pseudo-code of the implementations actually used in the case studies of the paper.

Chapter 7

Paper 1: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible

Errata

1. The formula in Section 5.1, $F^t = M/(t * x)$, is inaccurate. It should have been $\binom{f}{t}2^t = M/(t * x)$. Consequently, instead of "$= 20,000$ features" on p. 649, it should have been "$\approx 14,143$ features".

2. The algorithm on p. 644 does not say that the invalid tuples should only be removed once, and that is on the given condition. The algorithm still works but is slower. The condition should have been included in the pseudo-code and was included in the implementation all along.

3. We wrote on p. 643: "Therefore, for the class of feature models intended to be configured by humans assisted by computers, which we think at least is a very large part of the realistic feature models, quick satisfiability is also a property." This is an under-statement given the argumentation preceding it. It should have been: "Therefore, for realistic feature models, quick satisfiability is a property."

Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible

Martin Fagereng Johansen[1,2], Øystein Haugen[1], and Franck Fleurey[1]

[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

Abstract. Feature models and associated feature diagrams allow modeling and visualizing the constraints leading to the valid products of a product line. In terms of their expressiveness, feature diagrams are equivalent to propositional formulas which makes them theoretically expensive to process and analyze. For example, satisfying propositional formulas, which translates into finding a valid product for a given feature model, is an NP-hard problem, which has no fast, optimal solution. This theoretical complexity could prevent the use of powerful analysis techniques to assist in the development and testing of product lines. However, we have found that satisfying realistic feature models is quick. Thus, we show that combinatorial interaction testing of product lines is feasible in practice. Based on this, we investigate covering array generation time and results for realistic feature models and find where the algorithms can be improved.

Keywords: Software Product Lines, Testing, Feature Models, Practical, Realistic, Combinatorial Interaction Testing.

1 Introduction

A software product line is a collection of systems with a considerable amount of code in common. The commonality and differences between the systems are commonly modeled as a feature model. Testing of software product lines is a challenge since testing all possible products is intractable. Yet, one has to ensure that any valid product will function correctly. There is no consensus on how to efficiently test software product lines, but there are a number of suggested approaches. Each of the approaches still suffers from problems of scalability (Section 2).

Combinatorial interaction testing [4] is a promising approach for performing interaction testing between the features in a product line. Most of the difficulties of combinatorial interaction testing have been sorted out, but there is one part of it that is still considered intractable, namely finding a single valid configuration, an NP-hard problem. This is thus the bottleneck of the approach. In this paper we resolve this bottleneck such that combinatorial interaction testing should not be considered intractable any more (Section 3). We then investigate how a basic covering array generation algorithm performs on realistic feature models (Section 4), and suggest, based on the resolution of the bottleneck and on the empirics, how the algorithm can be improved (Section 5).

2 Background

2.1 Software Product Lines

A software product line (SPL) [19] is a collection of systems with a considerable amount of code 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 code. It is common for customers to have conflicting requirements. In that case, it is not even possible to ship one system for all customers.

The Eclipse products [22] can be seen as a software product line. Today, Eclipse lists 12 products on their download page[1]. 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 [10]. 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, one has to choose one windowing system which one wants Eclipse to run under. This is modeled as an empty semi-circle on the outgoing edges. When choosing a team functionality provider, one or all can be chosen. This is modeled as a filled semi circle. The team functionality itself is marked with an empty circle. This means that that feature is optional. A filled circle means that the feature is mandatory. One has to configure the feature model from the root, and one can only include a feature when the preceding feature is selected. For example, supporting CVS over SSH requires that one has CVS.

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. Such a configuration consists of specifying whether each feature is included or not.

[1] `http://eclipse.org/downloads/`

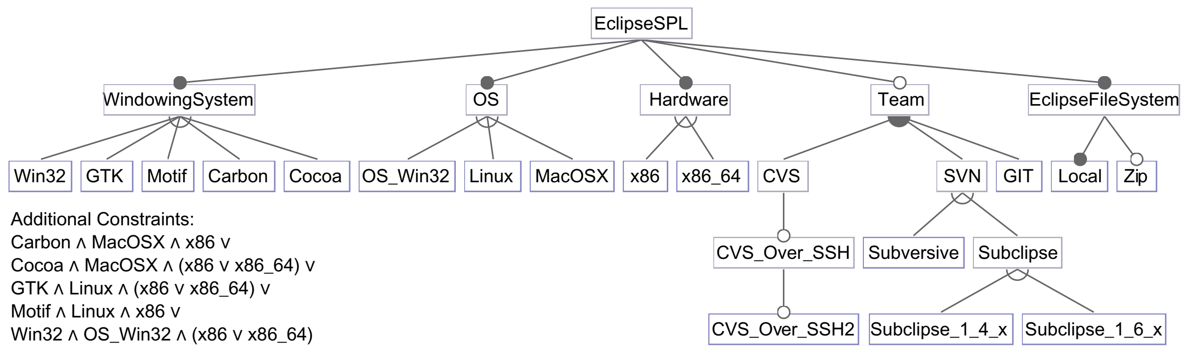


Fig. 1. Feature model for a subset of Eclipse

2.2 Software Product Line Testing

Testing a software 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 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, the number of possible configurations is 512, and this is a relatively simple product line.

There is no single recommended approach available today for testing product lines efficiently [5], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [4], discussed below; reusable component testing, 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 [21]; 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 [25].

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. 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 1 shows the 22 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 mandatory, and some pairs are not covered since they are invalid according to the feature model.

Table 1. Pair-wise coverage of the feature model in Figure 1 the test suites numbered

Feature\ Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
EclipseSPL	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WindowingSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
GTK	-	X	-	-	-	-	X	-	-	X	-	-	-	X	-	-	X	-	-	-	-	-
Motif	-	-	-	-	X	-	-	X	-	-	-	X	-	-	-	-	-	-	X	-	-	-
Carbon	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	X	-	X	-	-	-	-
Cocoa	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	X	-
OS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OS_Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
Linux	-	X	-	-	X	-	X	X	-	X	-	X	-	X	-	-	X	-	X	-	-	-
MacOSX	X	-	-	X	-	-	-	-	-	-	X	-	X	-	-	X	-	X	-	X	X	-
Hardware	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
x86	X	-	-	X	X	-	-	X	-	X	X	X	-	-	X	X	-	X	X	-	-	-
x86_64	-	X	X	-	-	X	X	-	X	-	-	-	X	X	-	-	X	-	-	X	X	X
Team	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X
CVS_Over_SSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
CVS_Over_SSH2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X
SVN	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X	X
Subversive	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	-	X
Subclipse	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	X	-	X	X	X	X	-
Subclipse_1_4_x	-	-	-	-	-	-	-	X	X	X	-	-	-	-	-	X	-	-	-	-	X	-
Subclipse_1_6_x	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	X	X	X	-	-
GIT	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	X	X	-	X	X	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	-	-	X	X	-	-	-	X	-	-	-	-	X	-	-	-	-	-	X	-	X

Testing every pair is called 2-wise testing, or pair-wise testing. This is a special case of t-wise testing where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in one product, 3-wise coverage means that every combination of three features are present, etc. For our running example, 5, 64 and 150 products is sufficient to achieve 1-wise, 3-wise and 4-wise coverage, respectively.

An important motivation for combinatorial interaction testing is a paper by Kuhn et al. 2004 [11]. 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.

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 subset of products must be generated from the feature model for some coverage strength. Such a subset is called a t-wise covering array for a coverage strength t. Last, a single system testing technique must be selected and applied to each product in this covering array. The first and last of these stages are well understood. The second stage, however, is widely regarded as intractable, thereby rendering the approach useless for industrial size software product lines.

3 The Case for Tractable t-wise Covering Array Generation

3.1 Complexity Analysis of Covering Array Generation

The generation of t-wise covering arrays is equivalent to the minimum set cover problem, an NP-complete problem. Given a set of elements, for example $U = \{1, 2, 3, 4, 5\}$; we have a set of sets of elements from U, for example $S = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$. The set cover problem is to identify the minimum number of sets, C, from S such that the union contains all elements from U, which is for the example $C = \{\{1, 2, 3\}, \{4, 5\}\}$.

1-wise covering array generation is easily converted to a set cover problem by listing all valid configurations of the product line, and having that as S. Each element of U is a pair with the feature name and a Boolean specifying the inclusion or exclusion of the feature. Solving this set cover problem then yields a 1-wise covering array. This can be done similarly for $t > 1$ by having tuples of assignments in U.

The set cover problem has a known approximation algorithm. (An approximation in this context is not the degree of t-wise coverage, which is 100% for all the discussion in this paper; but how many more products are selected than absolutely necessary.) The approximation algorithm was presented in Chvátal 1979 [3]. It is a greedy algorithm with a defined upper bound for the degree of approximation which grows with the size of the problem, but the degree of approximation remains acceptable. The algorithm is quite simple; it selects the set in S which covers the most uncovered elements until all elements are covered. For t-wise testing, this means selecting the product which covers the most uncovered tuples.

The set cover problem assumes that the sets with which to cover are already available so that one can look at all of them. For feature models, the solution space grows exponentially with respect to the number of features. Thus, it is infeasible to iterate through all the valid configurations.

And it gets worse, even generating a single configuration of a feature model is equivalent to the Boolean satisfiability problem (SAT), an NP-hard problem. SAT is the problem of assigning values to the variables of a propositional formula such that the formula evaluates to true. Batory 2005 [1] showed that ordinary feature models are equivalent to propositional formulas with respect to expressiveness, and that a feature model can easily be converted to a propositional formula.

Approximating the SAT problem is not possible: either we have the solution or we do not. This is also why the literature on combinatorial interaction testing classifies the generation of covering arrays as intractable.

3.2 Quick Satisfiability of Realistic Feature Models

Nie and Leung 2011 [16] is a recent survey of combinatorial testing. They state that their survey is the first complete and systematic survey on this topic. They found 50 papers on the generation of covering arrays. Covering array generation is reported to be NP-hard, but no detailed analysis is given. Such an analysis is given in both Perrouin et al. 2010 [18] and Garvin et al. 2011 [6] which both classify covering array generation as intractable because finding a single configuration of a feature model is equivalent to the Boolean satisfiability problem.

And this is indeed the general case given an arbitrary, grammatically valid feature model, but is it so in practice? It was observed by Mendonca et al. 2009 [15] that SAT-based analysis of realistic feature models with constraints is easy, but they did not identify the theoretical reason for this nor whether it is necessarily so and suggested finding the theoretical explanation as future work.

We propose that the theoretical explanation simply is that realistic feature models must be easily configurable by customers in order for them to efficiently use them. Configuring a feature model is equivalent to solving the Boolean satisfiability problem for the feature model.

The primary role of feature models in software product line engineering is for a potential customer to be able to sit down and configure a product to fit his or her needs. Imagine the opposite case. A company has developed a product line, but finding a single product of the product line takes a million years since there is no tractable solution to NP-hard problems. This situation is absurd. If it is really that difficult to find even a single product in a product line, then the feature model is too difficult for customers to use. If the customers cannot configure a feature model by hand assisted by a computer, is not an important point of the product line approach lost?

The same argument also shows that finding the solution to a partially configured feature model remains quick. If not, a customer might come into the situation that he or she cannot manage to complete the product configuration.

The kind of complexity that gives rise to modern computers being unable to solve a Boolean satisfiability problem in a timely manner would start challenging what is understandable by an engineer maintaining the product line.

Therefore, for the class of feature models intended to be configured by humans assisted by computers, which we think at least is a very large part of the realistic feature models, quick satisfiability is also a property.

3.3 Configuration Space

Even if the satisfiability of a realistic feature model is quick, traversal of the configuration space is still an issue. The configuration space of a feature model grows exponentially with the number of features, so one cannot traverse this space looking for the configuration that covers the most uncovered tuples, as required by Chvátal's greedy approximation algorithm.

Even if one only manages to cover one tuple per iteration, the upper bound for both time and the numbers of products is polynomial, since the number of tuples is $\binom{f}{t}$ (where f is the number of features and t the coverage strength; for example, $\binom{f}{2}$ gives the number of ways we can select a pair out of the configured features where order does not matter.) It is highly likely, however, that one is able to quickly cover many tuples. For pair-wise coverage, finding the first product covers $\binom{f}{2}$ out of $4\binom{f}{2}$ pairs for the worst case scenario. This is at least 25% of the possible pairs.

Covering many tuples at each iteration is still a challenge, but the upper bound of the penalty is polynomial. Since it is not feasible to traverse the configuration space to find the product which covers the most tuples, neither is it possible to guarantee the upper bound for the approximation with Chvátal's greedy algorithm. As we will see in the section on empirics, this does not seem to be a problem as one is usually able to cover many tuples per iteration.

3.4 Tractable Approximation of Covering Arrays

Since finding a covering array consists of two parts, finding valid configurations and solving the set cover problem, and since the former was shown to be tractable and the second is approximable by Chvátal's algorithm, we conclude that finding an approximation of the covering array is also tractable for realistic feature models.

4 Performance of Chvátal's Algorithm for Covering Array Generation

Even if the generation of covering arrays can be shown to be tractable, some improvement of the algorithms still have to be done in order to generate covering arrays from some of the largest known feature models. Let us look at how a basic implementation of Chvátal's algorithm for generating covering arrays performs and then discuss how to improve it.

The following algorithm assumes a feature model, FM, has been loaded, and a strength, t, of the wanted coverage strength has been given. From the set of assignments, (f, i), where f is a feature of FM, and i is a Boolean specifying whether f is included, all combinations of t assignments are generated and placed in a set, U. This set then includes all valid and invalid tuples.

An Adaption of Chvátal's Algorithm for Covering Array Generation.

```
While U is not empty:
  c is a configuration of FM with no variables assigned.
  For each tuple e in U:
    Satisfy FM assuming the assignments in both c and e.
    If satisfiable: Fix the assignments of e in c. Remove e from U.
  Satisfy FM assuming c, add the solution to the covering array C.
  //At some point, decide to remove the invalid tuples from U.
  If the number of newly covered tuples < number of features:
    For each tuple e in U:
      If FM is not satisfiable assuming e, remove e from U.
//C now holds the covering array of FM of strength t.
```

4.1 Models

Sometimes in papers discussing combinatorial interaction testing, experiments are run on randomly generated feature models. The problem with that is that one is assuming things about feature models that might not be realistic. Here, performance measurements will be run on realistic feature models, so that no assumptions are made on the nature of realistic feature models.

Models[2] were gathered from some available sources within software product line engineering research where the models are open and available. All the feature models are either of actual product lines or related to publications. The models are listed in Table 2 together with the product line name and their source.

Table 2. Models and Sources

System name	Model File Name	Source
X86 Linux kernel 2.6.28.6	2.6.28.6-icse11.dimacs	[23]
Part of FreeBSD kernel 8.0.0	freebsd-icse11.dimacs	[23]
eCos 3.0 i386pc	ecos-icse11.dimacs	[23]
e-Shop	Eshop-fm.xml	[12]
Violet, graphical model editor	Violet.m	`http://sourceforge.net/projects/violet/`
Berkeley DB	Berkeley.m	`http://www.oracle.com/us/products/database/berkeley-db/index.html`
Arcade Game Maker Pedagogical Product Line	arcade_game_pl_fm.xml	`http://www.sei.cmu.edu/productlines/ppl/`
Graph Product Line	Graph-product-line-fm.xml	[13]
Graph Product Line Nr. 4	Gg4.m	an extended version of the Graph Product line from [13]
Smart home	smart_home_fm.xml	[27]
TightVNC Remote Desktop Software	TightVNC.m	`http://www.tightvnc.com/`
AHEAD Tool Suite (ATS) Product Line	Apl.m	[24]
Fame DBMS	fame_dbms_fm.xml	`http://fame-dbms.org/`
Connector	connector_fm.xml	a tutorial [26]
Simple stack data structure	stack_fm.xml	a tutorial [26]
Simple search engine	REAL-FM-12.xml	[14]
Simple movie system	movies_app_fm.xml	[17]
Simple aircraft	aircraft_fm.xml	a tutorial [26]
Simple automobile	car_fm.xml	[28]

4.2 Tool and Transformations

The models gathered were of many different formats. Software product line engineering is an active field of research, and there are many research tools for different purposes and with various strengths and weaknesses.

In order to measure the performance of covering array generation on the gathered models, integration and some modification of existing tools and libraries were needed to make them cooperate. Figure 2 shows the overview of the tool

[2] The models are available at the following URL: `http://heim.ifi.uio.no/martifag/models2011/fms/`

that was constructed for this purpose[3]. The figure is of no particular graphical modeling notation. The diamonds symbolize files with a certain suffix, the boxes symbolize internal data structures and the arrows symbolize transformations between the formats.

The tool accepts feature models in three different formats: GUI DSL (model names suffixed with '.m'), as shipped with earlier versions of Feature IDE; SXFM, the Simple XML Feature Model format (model names suffixed with '.xml') and dimacs (model names suffixed with '.dimacs'), a file format for storing propositional formulas in conjunctive normal form (CNF).

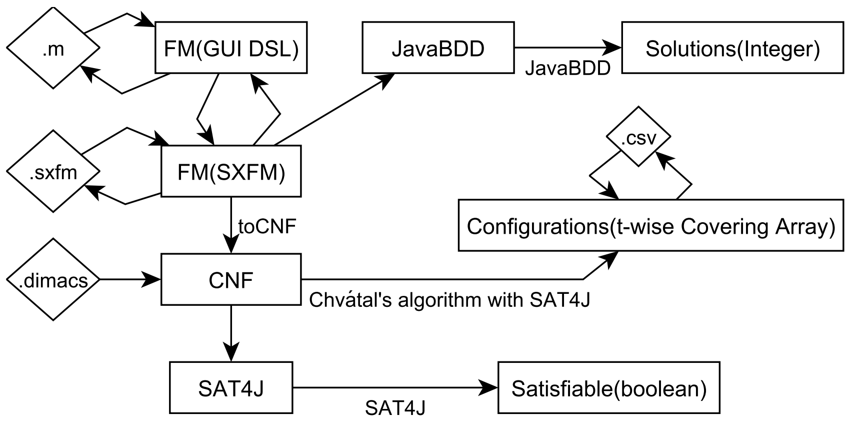


Fig. 2. Transformations in the tool

The GUI DSL files can be loaded using the Feature IDE library. This library allows writing and reading of SXFM files. Thus, they can be loaded into the SPLAR library[4] along with other SXFM files.

The SPLAR library provides an export to conjunctive normal form (CNF), a canonical way of representing general propositional constraints. Thus all the previously loaded models can be converted into CNF formulas, along with other formulas stored in dimacs files.

Once a model is in the form of a CNF formula, it can be given to SAT4J, an open source tool for solving the SAT problem. Thus, all the feature models can be input to the covering array algorithm discussed above. (SAT4J is also used to calculate satisfiability time for the feature models.)

The covering arrays are written to a comma separated values (CSV) file, which can be viewed in Microsoft Excel, Open Office Calc, etc. The covering arrays are then ready to be used to configure products for which single system testing is applied.

(Another interesting thing to know about a feature model is the number of possible configurations. The SPLAR library makes it possible to generate a

[3] The tool is available as open source at `http://heim.ifi.uio.no/martifag/models2011/spltool/`

[4] `http://splar.googlecode.com`

binary decision tree (BDD) which JavaBDD can work with. It then calculates the number of possible configurations of the feature model.)

4.3 Results

Table 3 shows the results from running[5] our tool on the feature models in Table 2. The feature models are ordered after the number of features. The next column shows the number of unique constraints in the model as the number of clauses of the conjunctive normal form of the constraints. (Constraints implied by the structure of the feature diagrams were not included in the count.) The number of valid products for each feature model is available for some of the smaller models, and as can be seen, quickly increases. The next column shows the time, in milliseconds, for running SAT4J on the feature model to find a single valid solution. The following columns show both the size and time for generating covering array of strengths 1–4. Some of the results are not available because the current implementation of the tools to not scale well to these sizes.

Boolean Satisfiability Times for Feature Models. Satisfiability in general has a worst case of about $O(2^n)$ according to Pătraşcu and Williams 2010 [20]. Table 3 shows the satisfiability times for the feature models. Empirically the satisfiability time of the feature models remains low. Thus, our conclusion regarding the quickness of satisfiability of realistic feature models is consistent with these few observations. Note that this is not meant as a validation, but merely as a demonstration of what we discussed in Section 3; that is, it follows from the fact that the feature models are meant to be configured manually.

Covering Array Generation. The following are the statistically significant relations[6] between the number of features and the sizes of the covering arrays. $CA(P, t)$ is the covering array with strength t for the propositional formula, P, representing a feature model with F features. The size function gives the size of the covering array.

$$log(size(CA(P, 2))) = 0.37 * log(F) + 1.30, \text{ adjusted } R^2\text{: } 0.59$$
$$log(size(CA(P, 3))) = 1.09 * log(F) + 0.00, \text{ adjusted } R^2\text{: } 0.63$$

Covering array sizes of strength 1 and 4 did not allow for a statistical model with a decent fit to be made. The fit for strengths 2 and 3 are poor. The reason is that covering array sizes are not really dependent on the number of features but on the structure of the feature model. For example, for 1-wise coverage, a covering array of size 2 might be sufficient: a certain assignment of optional features and the inverse.

[5] The computer on which we did the measurements had an Intel Q9300 CPU @2.53GHz and 8 GB, 400MHz ram. All executions ran in one thread.

[6] Adjusted R^2 is a measure, ranging from 0 to 1, of the goodness of fit of a statistical model. A value of 0.90 means that it is very unlikely a random sample would fit this approximation with the same significance, and a value of 0.20 means that it is very likely.

Table 3. Feature Models, satisfiability times, covering array sizes and generation times

Feature Model \ keys	Features	Constraints	Solutions	SAT time (ms)	1-way size	1-way time (ms)	2-way size	2-way time (ms)	3-way size	3-way times (ms)	4-way size	4-way time (ms)
2.6.28.6-icse11.dimacs	6,888	187,193	n/a	125	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
freebsd-icse11.dimacs	1,396	17,352	n/a	18	7	257,324	n/a	n/a	n/a	n/a	n/a	n/a
ecos-icse11.dimacs	1,244	2,768	n/a	12	6	16,178	n/a	n/a	n/a	n/a	n/a	n/a
Eshop-fm.xml	287	22	n/a	5	4	920	22	364,583	n/a	n/a	n/a	n/a
Violet.m	101	90	1.55E+26	1	4	280	28	21,278	121	3,865,245	n/a	n/a
Berkeley.m	78	47	4.03E+09	1	3	250	23	11,195	96	1,974,741	n/a	n/a
arcade_game_pl_fm.xml	61	35	3.30E+09	3	4	249	17	8,219	63	681,044	n/a	n/a
Gg4.m	38	23	960	1	6	171	22	1,903	63	88,355	156	11,915,393
smart_home_fm.xml	35	1	1,048,576	9	2	141	11	1,046	28	33,010	73	1,995,731
TightVNC.m	30	4	297,252	1	4	109	13	917	46	18,144	124	1,404,756
Apl.m	25	3	4,176	1	3	78	10	583	34	8,865	91	429,207
fame_dbms_fm.xml	21	1	320	3	3	109	9	515	24	5,138	49	121,989
connector_fm.xml	20	1	18	3	6	141	15	485	18	4,147	18	48,086
Graph-product-line-fm.xml	20	15	30	3	5	141	15	512	26	4,390	30	88,022
stack_fm.xml	17	1	432	7	3	109	12	409	41	2,471	96	56,086
REAL-FM-12.xml	14	3	126	7	4	94	13	340	33	1,261	66	18,807
movies_app_fm.xml	13	1	24	3	2	78	6	252	14	963	22	6,065
aircraft_fm.xml	13	1	315	7	3	78	10	286	23	1,131	54	9,597
car_fm.xml	9	3	13	7	3	63	7	200	12	425	13	1,141

The following are the estimated relations between the number of features and the time taken in milliseconds of generating the covering arrays.

$log(time(CA(P,1))) = 1.46 * log(F) + 0.00$, adjusted R^2: 0.84
$log(time(CA(P,2))) = 2.13 * log(F) + 0.00$, adjusted R^2: 0.96
$log(time(CA(P,3))) = 4.03 * log(F) - 3.51$, adjusted R^2: 0.98
$log(time(CA(P,4))) = 6.02 * log(F) - 6.41$, adjusted R^2: 0.97.

5 Discussion

5.1 Memory Requirements

The way our tool deals with the constraints in a feature model is to calculate and store the valid, uncovered tuples in memory. The tuples need to be traversed in order to find the configurations which cover the most uncovered tuples at each iteration. Doing it this way, the number of constraints does not affect the memory requirement significantly, but memory might prove to be a bottle neck.

This effectively sets the memory requirement to $O(F^t)$, where F is the number of features in a feature model and t is the strength of the coverage. For a system with M bytes of memory and assuming each t-tuple requires $t * x$ bytes, the upper bound for t-wise coverage is $F^t = M/(t * x)$.

For pair-wise coverage on a system with 8GB of memory, and assuming that a structure holding the pairs take 20 bytes, the upper bound is $n = \sqrt{8,000,000,000/20}$, $n = 20,000$ features. This is the upper bound of a high-end laptop. More powerful computers are available which can be used for generating covering arrays which increases the upper bound such that even 3-wise coverage of the second largest feature model in our sample is within.

5.2 Accepted Covering Array Size

There is a correspondence between the number of features in a feature model and the size of the team working with it. Thus a team of developers and testers should be able to deal with a covering array of a size around the same size as the number of features. If we look at the data and statistical models for covering array size, we can see that the size of 1–3-wise covering arrays is below or close to the number of features since the coefficient of log(F), and thus the exponent of F, is less than or close to 1.

5.3 Suggested Improvements and Future Work

Given the evaluations up to this point, there are a number of source of improvement for generating covering arrays for software product lines.

Exploiting the Boolean Satisfiability Speed. Nie and Leung 2011 [16] classified handling constraints for covering array generation is an open problem for covering array generation in general. Using SAT-solvers is good way to handle constraints for covering array generation based on feature models. Also, since satisfiability of feature models has been assumed to be intractable up to this point, it might be an unexploited source for improvement of covering array generation speed.

Parallelization. The algorithm that was used to make the measurements in this paper ran in one thread. An algorithm which supports running on several threads will improve the execution time for generating the covering arrays. For example, the step for finding all invalid tuples in the adaption of Chvátal's algorithm above can be run in parallel by splitting the set of tuples in, for example, four and checking each fourth in a separate thread.

Heuristics. Another unexploited source of improvement for covering array generation is knowledge from the domain model. UML-models and annotations on feature models should be taken into account when generating a covering array to make it smaller and its generation time lower. CVL [7] is a variability language with tool support which, in addition to feature diagrams, models the variability of a system on the system model as well. Knowing what a feature refers to in a system model is an unexploited source of improvement for covering array generation.

In a recent publication [8], we show how to exploit one commonly occurring structure in product lines when doing combinatorial interaction testing. Often there are several implementations of the same basic functionality which is used by the other components in the product line through an abstraction layer. These implementations occur as mutual exclusive alternatives in the feature model. Mutual exclusive alternatives are detrimental to combinatorial interaction testing [2], causing a substantial increase in the number of products in the covering arrays. We show that if the increase of products is due to the abstraction layer implementations, then the number of test suites required can be reduced by reusing test suites for several of the products in the array without losing the bug detection capabilities.

6 Conclusion

In this paper we showed that although it is widely held that configuring feature models is intractable, in practice the role of feature models in software product line engineering implies that it is quick. Boolean satisfiability solvers thus provide an efficient way to handle constraints in feature models and should be exploited for doing covering array generation without the fear that the running time will be intractable.

Acknowledgments. We want to thank the anonymous reviewers for their helpful feedback.

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. Cabral, I., Cohen, M.B., Rothermel, G.: Improving the testing and testability of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 241–255. Springer, Heidelberg (2010)
3. Chvátal, V.: A greedy heuristic for the Set-Covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
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. Engström, E., Runeson, P.: Software product line testing - a systematic mapping study. Information and Software Technology 53(1), 2–13 (2011)
6. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Softw. Engg. 16, 61–102 (2011)
7. 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)
8. Johansen, M.F., Haugen, Ø., Fleurey, F.: Bow tie testing - a testing pattern for product lines. In: Proceedings of the 16th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2011), Irsee, Germany, July 13-17 (2011)
9. Johansen, M.F., Haugen, Ø., Fleurey, F.: A survey of empirics of strategies for software product line testing. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW, pp. 266–269 (March 2011)
10. 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)
11. 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)
12. Lau, S.Q.: Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, ECE Department, University of Waterloo, Canada (2006)
13. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Dannenberg, R.B. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
14. Mendonca, M.: Efficient Reasoning Techniques for Large Scale Feature Models. Ph.D. thesis, School of Computer Science, University of Waterloo (January 2009)
15. Mendonca, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: Proceedings of the 13th International Software Product Line Conference, pp. 231–240. Carnegie Mellon University (2009)
16. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. 43, 11:1–11:29 (2011)
17. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 131–140. Carnegie Mellon University, Pittsburgh (2009)

18. Perrouin, G., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Automated and scalable t-wise test case generation strategies for software product lines. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST 2010, pp. 459–468. IEEE Computer Society, Washington, DC (2010)
19. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
20. Pătraşcu, M., Williams, R.: On the possibility of faster sat algorithms. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1065–1075. Society for Industrial and Applied Mathematics, Philadelphia (2010)
21. 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)
22. Rivieres, J., Beaton, W.: Eclipse Platform Technical Overview (2006)
23. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 461–470. ACM, New York (2011)
24. Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 191–200. ACM, New York (2006)
25. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Transactions on Software Engineering 36(3), 309–322 (2010)
26. Voelter, M.: Using domain specific languages for product line engineering. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 329–329. Carnegie Mellon University, Pittsburgh (2009)
27. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 211–220. Carnegie Mellon University, Pittsburgh (2009)
28. White, J., Dougherty, B., Schmidt, D.C., Benavides, D.: Automated reasoning for multi-step feature model configuration problems. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 11–20. Carnegie Mellon University, Pittsburgh (2009)

Chapter 8

Paper 2: An Algorithm for Generating t-wise Covering Arrays from Large Feature Models

Errata

1. The sum of 3-wise covering array sizes for IPOG should have been 345, not 601. Consequently, the text should say that IPOG produces comparable covering array sizes to three of the other top algorithms for 3-wise. Thanks to Linbin Yu for identifying this mistake.
2. The version of NIST ACTS used was not mentioned. The version used was version 2 beta, revision 1.5 acquired 2012-01-04.

An Algorithm for Generating t-wise Covering Arrays from Large Feature Models

Martin Fagereng Johansen[1,2], Øystein Haugen[1] and Franck Fleurey[1]
[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

ABSTRACT

A scalable approach for software product line testing is required due to the size and complexity of industrial product lines. In this paper, we present a specialized algorithm (called ICPL) for generating covering arrays from feature models. ICPL makes it possible to apply combinatorial interaction testing to software product lines of the size and complexity found in industry. For example, ICPL allows pair-wise testing to be readily applied to projects of about 7,000 features and 200,000 constraints, the Linux Kernel, one of the largest product lines where the feature model is available. ICPL is compared to three of the leading algorithms for t-wise covering array generation. Based on a corpus of 19 feature models, data was collected for each algorithm and feature model when the algorithm could finish 100 runs within three days. These data are used for comparing the four algorithms. In addition to supporting large feature models, ICPL is quick, produces small covering arrays and, even though it is non-deterministic, produces a covering array of a similar size within approximately the same time each time it is run with the same feature model.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Algorithms, Verification

Keywords

Product Lines, Testing, Feature Models, Combinatorial Interaction Testing

1. INTRODUCTION

A software product line (SPL) is a collection of systems with a considerable amount of code in common. The commonality and differences between the systems are commonly modeled as a feature model. Testing product lines is a challenge since testing all possible products is intractable. Yet, one has to ensure that any valid product will function correctly. There is no consensus on how to efficiently test software product lines [5], but there are a number of suggested approaches. Each of the approaches still has problems of scalability; this is covered in Section 2.

Combinatorial interaction testing [3] is a promising approach for performing interaction testing between the features of a product line. Most of the difficulties of combinatorial interaction testing have been sorted out.

In [11], we resolved a previously considered bottleneck of combinatorial interaction testing and concluded that generating covering arrays from realistic feature models is tractable. We evaluated a basic algorithm for generating covering arrays. The performance of this basic algorithm was measured on a corpus of 19 realistic feature models. It was concluded that it is possible to make an algorithm that could generate covering arrays from the larger feature models, and that the basic algorithm has insufficient performance.

In this paper we present an algorithm, called ICPL[1] that is able to generate covering arrays from large feature models. Based on the basic algorithm presented in [11], the new algorithm contains optimizations that significantly increases the performance of it, described in detail in Section 4.

We then present the results from an experiment[2], Section 5, using the same corpus of 19 feature models from [11] on ICPL and on three of the leading algorithms for t-wise covering array generation. In this experiment, each of the five algorithms was given an opportunity to run 100 times on each of the 19 feature model for each strength of 1–3. The analysis of these data is presented in Section 5 and gives an extensive insight into how ICPL performs on its own and in comparison to the other algorithms. The experiment shows that ICPL successfully generates a pair-wise covering array for a feature models of about 7,000 features and 200,000 constraints, the Linux Kernel, one of the largest product lines where the feature model is available. In addition to supporting large feature models, ICPL produces small covering arrays overall and has the highest level of scalability. For ICPL, the standard deviation is low for both the time taken to generate the covering arrays and their sizes.

[1]ICPL is a recursive accronym that stands for "ICPL Covering array generation algorithm for Product Lines".
[2]An open source implementation of ICPL is available at `http://heim.ifi.uio.no/martifag/splc2012/` along with all the measurements, the feature models in the corpus and scripts used in the experiment.

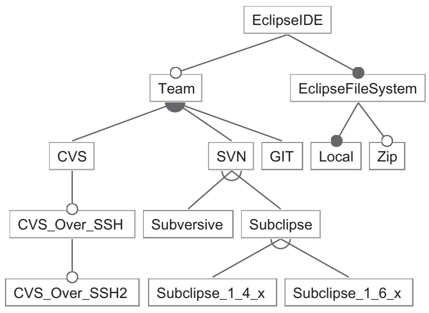


Figure 1: Feature Model for a Subset of the Eclipse IDE Product Line

2. BACKGROUND AND RELATED WORK

2.1 Software Product Lines

A software product line [19] is a collection of systems with a considerable amount of code in common. The primary motivation for structuring a system 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 system for all customers.

For example, the Eclipse IDEs [21] can be seen as a software product line. Today, the Eclipse project lists 11 products on their download page[3].

One way to model the commonalities and differences in a product line is using a feature model[4] [13]. 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 a part of the feature model for the Eclipse IDEs. The figure uses the common notation for feature models; for a detailed explanation of feature models, see Czarnecki and Eisenecker 2000 [4].

2.2 Software Product Line Testing

Testing a software 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 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, the number of possible configurations is 64, but this is a relatively simple product line.

There is no single recommended approach available today for testing product lines efficiently [5], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [3], discussed below; reusable component testing, which we have seen in industry [12], 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 [20]; 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 [23]. Kim et al. 2011 [14] presented a technique where they can identify irrelevant features for a test case using static analysis. They use that information to reduce the combinatorial number of product to test.

[3]`http://eclipse.org/downloads/`, retrieved 2012-01-16

[4]Throughout this paper, the term feature models is used to mean basic feature models which have equivalent expressiveness as propositional logic formulas.

Table 1: 2-wise Covering array of the Feature Model in Figure 1.

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
Team	X	X	-	X	X	X	X	X	X	X	-	X
CVS	X	X	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH	X	-	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH2	-	-	-	X	X	-	X	-	X	-	-	-
SVN	X	X	-	X	X	X	-	X	X	X	-	-
Subversive	X	-	-	X	-	-	-	X	-	-	-	-
Subclipse	-	X	-	-	X	X	-	-	X	X	-	-
Subclipse_1_4_x	-	X	-	-	X	-	-	-	-	X	-	-
Subclipse_1_6_x	-	-	-	-	-	X	-	-	X	-	-	-
GIT	X	-	-	-	X	X	-	X	-	-	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	X	X	X	-	-	-	X	X	-	-	-

2.3 Combinatorial Interaction Testing for Product Lines

Combinatorial interaction testing [3] is one of the most promising approaches. The benefits of this approach are that it deals directly with the feature model to derive a small set of products which can then be tested using single system testing techniques, of which there are many good ones. 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 1 shows the 12 products that must be tested to ensure that every pairwise 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 excluded. 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. Since every pair is present, this set of products is called a *pairwise covering array*, or *2-wise covering array*. (The latter terminology will be used in this paper.) Testing a product line based on a 2-wise covering array is called *2-wise testing* of the product line.

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 is present, etc. For our running example, 4, 12 and 27 products are sufficient to achieve 1-wise, 2-wise and 3-wise coverage, respectively.

The main motivation behind combinatorial interaction testing are the empirics collected by Kuhn et al. 2004 [15]. The empirics they collected indicate that most bugs are caused by simple interactions between features.

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 (for example as in Figure 1). Second, the subset of products must be generated from the feature model for some coverage strength (for example as in Table 1). Last, after having built each product, a single system testing technique must be selected and applied to each product.

The first and last of these stages are well understood. For the second stage, however, no scalable algorithm is known[5], thereby rendering the approach less usable for larger industrial size software product lines. An algorithm for generating covering arrays from large feature models, Section 4, and an evaluation of it, Section 5, are the contributions of this paper.

3. THE BASIC ALGORITHM

We have previously presented Algorithm 1 in [11]. Using a greedy algorithm for generating a covering array in this way has certainly been known since the publication of Chvátal's algorithm [2], which presents a greedy heuristic for the set cover problem, of which covering array generation is a one type of. It has been previously discussed in the context of single system testing in Grieskamp et al. 2009 [8] and in Calvagna and Gargantini 2009 [1] among others.

3.1 Groundwork

To explain the algorithms in the following sections, some groundwork is needed.

An assignment a is a pair $(f, included)$, where f is a feature of a feature model FM, and *included* is a Boolean specifying whether f is included or not. A configuration, C, is a set of assignments where all features in the feature model, FM, have been given an assignment. A *t-set* is a set of t assignments, $\{a_1, a_2, ...\}$. The set of all t-sets of a feature model, FM, is denoted T_t, e.g. $T_1, T_2, T_3, ...$ The set of invalid t-sets is denoted I_t, e.g. $I_1, I_2, I_3, ...$ The set of valid t-sets, the universe of the set cover problem, is thus $U_t = T_t \setminus I_t$. A covering array of strength t is a set of configurations, C_t, in which all valid t-sets, all t-sets in U_t, are a subset of (or the same set as) at least one of the configurations.

The feature model is stored in an object and has some methods:

- $size()$. In this context, the size of the feature model is the number of features. $|FM|$ is an alternative notation to get the size of a feature model.
- $genT(t)$ returns the set of all combinations of t feature assignments (both valid and invalid), T_t.
- $satisfy(c)$ returns a valid configuration of the feature model given the assignments in c. Although not a requirement, this function gives a configuration that tends to have no more features included than necessary. This is the default behavior of SAT4J which is used in our concrete implementation of FM's satisfy method.
- $is\_satisfiable(c)$ returns true if the feature model is satisfiable given the assignments in c.

[5]Scalable algorithms are known for special cases, such as when there are no constraints.

3.2 Algorithm

Algorithm 1 requires that a feature model, FM, has been loaded. It also assumes a strength, t, of the wanted coverage strength has been given.

As long as there are more t-sets to cover, the algorithm fills up a new configuration, C, with many uncovered t-sets. The covered t-sets are first added to the set CO, line 9, before being removed from T, line 12.

At a certain point, line 17, all the invalid t-sets will be removed from T, leaving only valid t-sets in T. This point has been empirically determined to be when the order of magnitude of the covered t-sets is the same as the order of magnitude of the size of the feature model, when $(\lfloor log_{10}|CO| \rfloor \leq \lfloor log_{10}|FM| \rfloor)$. For example, with a feature model with 500 features, the invalid t-sets are removed when the number of t-sets covered is 999 or less. This condition could have been removed, meaning that all invalid t-sets would be removed in the first round. This is inefficient since the satisfiability solver would have to check a lot of valid t-sets, line 19. We found that it is better to wait until a lot of valid t-sets have been removed from T. It is also inefficient to wait until the algorithm has covered all valid t-sets, at which point CO would be empty. The reason is that the satisfiability solver will be burdened with a lot of invalid t-sets at line 7. The condition at line 17 was empirically found to be an efficient middle point between these two extremes.

Algorithm 1 Pseudo Code of Chvátal's Algorithm Adapted for Covering Array Generation

```
 1: $T \leftarrow FM.genT(t)$
 2: $(invalidRemoved, C_t) \leftarrow (false, \emptyset)$
 3: while $T \neq \emptyset$ do
 4:   $C \leftarrow \emptyset$
 5:   $CO \leftarrow \emptyset$
 6:   for each t-set $e$ in $T$ do
 7:     if $FM.is\_satisfiable(C \cup e)$ then
 8:       $C \leftarrow C \cup e$
 9:       $CO \leftarrow CO \cup \{e\}$
10:     end if
11:   end for
12:   $T \leftarrow T \setminus CO$
13:   if $CO \neq \emptyset$ then
14:     $C \leftarrow FM.satisfy(C)$
15:     $C_t \leftarrow C_t \cup \{C\}$
16:   end if
17:   if $(\neg invalidRemoved) \wedge (\lfloor log_{10}|CO| \rfloor \leq \lfloor log_{10}|FM| \rfloor)$
      then
18:     for each t-set $e$ in $T$ do
19:       if $\neg FM.is\_satisfiable(e)$ then
20:         $T \leftarrow T \setminus \{e\}$
21:       end if
22:     end for
23:     $invalidRemoved \leftarrow true$
24:   end if
25: end while
```

Algorithm 1 is guaranteed to finish with the complete t-wise covering array of FM in C_t. First, all combinations of t assignments are generated at line 1. Then, a t-set is either covered, line 8–9, and therefore removed from T, line 12, or found to be invalid and therefore removed from T at line 20. CO tends towards 0 as more and more valid t-sets are covered. This guarantees that the condition on line 17 will pass eventually. Since all t-sets are either valid or invalid, the condition 3 is guaranteed to evaluate to false eventually, at which point all t-sets have either been covered or classified as invalid.

4. THE NEW ALGORITHM: ICPL

ICPL works essentially the same way and for the same reasons as Algorithm 1, but ICPL does several things to avoid redundant work, learn from intermediate results and allows doing some things in parallel.

Algorithm 2 shows the highest level of the ICPL algorithm. Just as Algorithm 1, ICPL takes a feature model, FM, and a strength t and generates the covering array of strength t, C_t. In addition, the complete set of invalid t-sets, I_t, is returned. Some parts of ICPL have been put into sub-algorithms that will each be explained in turn in this section.

Lines 10–20 is the same greedy main-loop at in Algorithm 1, except that generating a single configuration, line 11, and finding all covered t-sets, line 13, have been separated from each other and put into sub-algorithms. Finding all invalid t-sets, lines 16–17, has also been separated out into a sub-algorithm. The point at which the invalid t-sets are removed, line 15, is exactly as in Algorithm 1.

Lines 1–9 are different than in Algorithm 1. It has been previously treated and utilized in Fouche et al. 2009 [6], that a covering array of strength $t-1$ is a subset of (or the same set as) a covering array of strength t, that is that $CA(FM,t-1) \subseteq CA(FM,t)$. Thus, before starting to cover for strength t, the algorithm covers for all values of n in $1 \leq n < t$. Thus, before generating C_t, the algorithm knows C_{t-1} and I_{t-1}. This is done in lines 6–8.

Line 2–4 is the bottom of the recursion. Here, the complete set of invalid 1-sets, I_1, is identified, line 2, which gives us the complete U_1 (stored in T_t), line 3. Since the whole I_1 is found, *invalidRemoved* is set to true, line 4.

Algorithm 2 $ICPL(FM,t) : (C_t, I_t)$

```
 1: if $t = 1$ then
 2:     $(C_t, I_t) \leftarrow genCompleteI_1(FM)$
 3:     $T_t \leftarrow FM.genT(1) \setminus I_t$
 4:     $invalidRemoved \leftarrow true$
 5: else
 6:     $(C_t, I_{t-1}) \leftarrow ICPL(FM, t-1)$
 7:     $(T_t, I_t) \leftarrow generateTSets(FM.getT(t), I_{t-1}, C_t)$
 8:     $invalidRemoved \leftarrow false$
 9: end if
10: while $T_t \neq \emptyset$ do
11:     $C \leftarrow genConfiguration(FM, T_t)$
12:     $C_t \leftarrow C_t \cup \{C\}$
13:     $CO \leftarrow getCovered(FM, C, T_t)$
14:     $T_t \leftarrow T_t \setminus CO$
15:     if $(\neg invalidRemoved) \wedge (\lfloor log_{10}|CO| \rfloor \leq \lfloor log_{10}|FM| \rfloor)$
        then
16:         $(T_t, TempI) \leftarrow genInvalid(FM, T_t)$
17:         $I_t \leftarrow I_t \cup TempI$
18:         $invalidRemoved \leftarrow true$
19:     end if
20: end while
21: return $(C_t, I_t)$
```

4.1 Finding Core and Dead Features Quickly

A step in the algorithm of generating a covering array is finding all core and dead features. Core features are features that must always be included, and dead features are features that can never be included. Therefore, if a feature is core, any assignment excluding it is invalid. Also, if a feature is dead, any assignment including it is invalid. (There should never to be dead features in a feature model, but it is possible to have them.)

The basic way to do this is simply trying to set each feature to included and excluded. Then, if the feature model is unsatisfiable, it implies that the feature is either dead or core, respectively. This is done in Algorithm 3. This algorithm takes the feature model, FM, and a set of 1-sets, T_1. It then checks each assignment to see if it is valid or not, line 3. Those that are invalid are added to the set of invalid 1-sets, I_1. This, hence, is guaranteed to give us all invalid 1-sets of T_1.

Algorithm 3 $getInvalidAssignments(FM, T_1) : I_1$

```
1: $I_1 \leftarrow \emptyset$
2: for each t-set $e$ in $T_1$ do
3:     if $\neg FM.is\_satisfiable(e)$ then
4:         $I_1 \leftarrow I_1 \cup \{e\}$
5:     end if
6: end for
7: return $I_1$
```

There is a way to speed this up. Algorithm 4 shows how this is done.

Algorithm 4 $genCompleteI_1(FM) : (C, I)$

```
 1: $(I, NotCore, NotDead) \leftarrow (\emptyset, \emptyset, \emptyset)$
 2: $T \leftarrow FM.genT(1)$
 3: $c_1 \leftarrow FM.satisfy(\emptyset)$
 4: for each assignment $a$ in $c_1$ do
 5:     if $\neg a.included$ then
 6:         $NotCore \leftarrow NotCore \cup \{a\}$
 7:     end if
 8: end for
 9: $IC \leftarrow \{ \forall e \in T$ : e is a not included feature$\} \setminus NotCore$
10: $I \leftarrow getInvalidAssignments(FM, IC)$
11: $c_2 \leftarrow FM.satisfyManyInclusions(\emptyset)$
12: for each assignment a in $c_2$ do
13:     if $a.included$ then
14:         $NotDead \leftarrow NotDead \cup \{a\}$
15:     end if
16: end for
17: $IC \leftarrow \{ \forall e \in T$ : e is an included feature$\} \setminus NotDead$
18: $I \leftarrow I \cup getInvalidAssignments(FM, IC)$
19: return $(\{c_1, c_2\}, I)$
```

Algorithm 4 gives the same result as Algorithm 3. The difference is that Algorithm 4 learns from two concrete configurations that were constructed a special way in order to do the same thing, only faster. (In addition, Algorithm 4 returns these two configurations.) The first part of the learning experience is line 3–9. Given, for example, a feature model with 7,000 features, the call at line 3 will give a configuration with, for example, 500 features included. This informs us that the remaining 6,500 features are not core features. If they were, they must have been included in a valid configuration. Thus, we do not have to check that 6,500 assignments with these features not included are invalid. Thus they are removed from the set of candidates for invalid assignments, IC.

The same things is repeated again, line 10–18. But now, a configuration with, for example, 5,000 features included is returned instead, at line 11. This informs us that these 5,000 features are not dead. If they were dead, they could not be included in a valid configuration. Thus, 5,000 assignments with included features can be removed from the set of candidate invalid assignments, line 17. ($satisfyManyInclusions$ inverts the feature model before calling $satisfy$ and then inverts the assignments in the configuration [10].)

4.2 Early Identification of Invalid t-sets

In Algorithm 1, line 1, the complete T_t was given. However, it is possible to be more clever in this step. How this is done is shown in Algorithm 5. In ICPL we use the knowledge from the covering array and the set of invalid t-sets from the $t-1$ step of the recursion, given as C_t and I_{t-1}.

Algorithm 5 $generateTSets(T, I_{t-1}, C_t) : (T, I_t)$

```
 1: $(NewT, I_t) \leftarrow (\emptyset, \emptyset)$
 2: Loop:
 3: for each t-set $e$ in $T$ do
 4:    for each configuration $C$ in $C_t$ do
 5:       if $e \subseteq C$ then
 6:          continue Loop
 7:       end if
 8:    end for
 9:    for each t-set $i$ in $I_{t-1}$ do
10:       if $i \subseteq e$ then
11:          $I_t \leftarrow I_t \cup \{e\}$
12:          continue Loop
13:       end if
14:    end for
15:    $NewT \leftarrow NewT \cup \{e\}$
16: end for
17: return $(NewT, I_t)$
```

If we removed lines 4–14, what we would get is the set T_t returned as $NewT$. This would give us the same result as Algorithm 1, line 1. But, two things are done in ICPL:

On lines 4–8, if a t-set has already been covered by a configuration in C_t, we do not have to cover it again. Thus, the t-set is skipped.

On lines 9–14, if i is an invalid (t-1)-set of I_{t-1}, so are all t-sets $e \in T$, for which $i \subseteq e$. Thus, those t-sets can be added to the set of invalid t-sets, line 11.

4.3 Generating a Single Configuration

Algorithm 6 finds a single configuration, which assignments are collected in C, that covers many uncovered t-sets.

This is essentially the same as is done in Algorithm 1, line 6–8. These three lines appear again in Algorithm 6 as lines 8, 16 and 17, but ICPL learns from the outcome of line 16 to avoid redundant work. It does not record which t-sets have been covered, as that is done in Algorithm 7 instead.

First of all, the set CF maintains a list of features that have been given an assignment. This enables ICPL to skip a t-set without having to pass it to the satisfiability solver, lines 9–15. It also enables us to break the search early, lines 28–30, if all features of T, found in lines 2–6 and stored in PF, have been given an assignment.

Algorithm 6 $genConfiguration(FM, T) : C$

```
 1: $(C, CF, PF) \leftarrow (\emptyset, \emptyset, \emptyset)$
 2: for each t-set $e$ in $T$ do
 3:    for each assignment $a$ in $e$ do
 4:       $PF \leftarrow PF \cup \{a.f\}$
 5:    end for
 6: end for
 7: Loop:
 8: for each t-set $e$ in $T$ do
 9:    $F \leftarrow \emptyset$
10:    for each assignment $a$ in $e$ do
11:       $F \leftarrow F \cup \{a.f\}$
12:    end for
13:    if $F \subseteq CF$ then
14:       continue
15:    end if
16:    if $FM.is\_satisfiable(C \cup e)$ then
17:       $C \leftarrow C \cup e$
18:       $CF \leftarrow CF \cup F$
19:    else
20:       for each assignment $a$ in $e$ do
21:          if $((e \setminus \{a\}) \subseteq C) \wedge (a \notin C)$ then
22:             $b \leftarrow a$ with assignment inverted
23:             $C \leftarrow C \cup \{b\}$
24:             $CF \leftarrow CF \cup \{b.f\}$
25:          end if
26:       end for
27:    end if
28:    if $|CF| = |PF|$ then
29:       break Loop
30:    end if
31: end for
32: return $FM.satisfy(C)$
```

When a t-set leads to unsatisfiability, line 19, and all assignments in the t-set except one are already in the configuration, C, line 21, we know that the opposite assignment of the only exception must be valid. This allows us to update the configuration, lines 22–24.

Algorithm 6 is guaranteed to give us a valid configuration that covers many uncovered t-sets. In line 32, we return a complete, valid configuration by calling the satisfiability solver. The main loop, lines 8–31, is guaranteed to end since there are only a finite number of t-sets in T which is not modified inside the loop. If there are valid, uncovered t-sets, then at least one will get covered on line 17, since set CF is empty, guaranteeing the check at lines 13 and 28 to fail until at least one t-set has been covered.

4.4 Finding the Covered t-sets

It is necessary to find the covered t-sets after a configuration was generated by Algorithm 6 because of the optimizations applied to it. In Algorithm 1, the covered t-sets were collected as they were covered. The result is the same in both cases: A configuration, C, and the set of covered t-sets, CO; but the combined speed is faster since Algorithm 6 is so much more efficient. How the covered t-sets are found in ICPL is shown in Algorithm 7.

Since ICPL deals with large feature models, it is important that the outer iteration of Algorithm 7, line 2, is on the set

Algorithm 7 $genCovered(FM, C, T) : CO$

```
1: $CO \leftarrow \emptyset$
2: for each t-set $e$ in $T$ do
3:   if $e \subseteq C$ then
4:     $CO \leftarrow CO \cup \{e\}$
5:   end if
6: end for
7: return $CO$
```

of uncovered tuples, T. T shrinks drastically as our greedy algorithm covers more and more t-sets. By iterating over T, Algorithm 7 speeds up towards the end of generating the covering array.

4.5 Identifying the Invalid t-sets

In Algorithm 1, the invalid t-sets were identified by checking each remaining t-set in turn, lines 18–22. Algorithm 8 does the same thing, but faster.

Algorithm 8 takes the feature model, FM, and the set of uncovered t-sets, T_t at line 16 of Algorithm 2, and classifies all the remaining t-sets as either valid, V, or invalid, I, and returns them.

Algorithm 8 $genInvalid(FM, T) : (V, I)$

```
1: $(I, V) \leftarrow (\emptyset, \emptyset)$
2: while $T \neq \emptyset$ do
3:   $e \leftarrow$ any t-set in T
4:   if $\neg FM.is\_satisfiable(e)$ then
5:     $I \leftarrow I \cup \{e\}$
6:     $T \leftarrow T \setminus \{e\}$
7:   else
8:     $V \leftarrow V \cup \{e\}$
9:     $T \leftarrow T \setminus \{e\}$
10:    $C \leftarrow FM.satisfy(e)$
11:    $NewV \leftarrow \emptyset$
12:    for each t-set $g$ in $T$ do
13:      if $g \subseteq C$ then
14:        $NewV \leftarrow NewV \cup \{g\}$
15:      end if
16:    end for
17:    $V \leftarrow V \cup NewV$
18:    $T \leftarrow T \setminus NewV$
19:  end if
20: end while
21: return $(V, I)$
```

Each t-set is classified in turn, line 4, and added to and removed from the respective sets, lines 5–6 and 8–9. Lines 11–18 contain an optimization. If a configuration, found in line 10, contains one of the unclassified t-sets, we can classify it as valid.

Algorithm 8 is guaranteed to finish since no elements are added to T, and each element picked is guaranteed to be either classified as valid or invalid.

4.6 Parallelization

Modern computers now commonly support truly parallel execution of programs. Algorithm 1 runs in one process only. It is neither trivial nor certain that an algorithm or parts of it can be parallelized, and no automatic way of parallelizing an arbitrary algorithm exists.

In ICPL, Algorithm 3, 5, 7 and 8 are all *data parallel*. Data parallel algorithms are easy to parallelize [9]. For example, if we have an array of integers, the algorithm that increases each element by 1 is data parallel. If the array has 1000 integers, then, for example, 4 processors can in parallel increase 250 elements each, giving the same result as if all 1000 had been increased by the same processor.

A similar parallel execution is easily done with the four data parallel sub-algorithms of ICPL. Each get a set of t-sets to work with and works on one t-set in each iteration.

Although not shown in the pseudo-code in this paper, our implementation of ICPL used in the experiment in the next section uses data parallel execution of these four algorithms, supporting any number of processors. It is not shown in the pseudo code because it requires uninteresting boiler-plate code to set things up properly. Instead, we give a summary of the parameters that can be split:

Algorithm 3 takes two parameters, the feature model object and the set T_1. T_1 is split up into n chunks, and the algorithm is executed in parallel in n threads with each of the n chunks and a copy of the feature model object. When all executions are done, there are n results available. These can be combined into the final result, I_1.

Algorithms 5, 7 and 8 are data parallel on parameter T. Their return values can be combined on completion.

5. RESULTS AND COMPARISON

We have set up an experiment[6] in order to demonstrate the performance of ICPL and to compare it to the performance of existing algorithms. Since the scope of ICPL is realistic feature models, we have reused without modification a collection of 19 realistic feature models from academia and industry previously presented in Johansen et al. 2011 [11]. The three largest feature models in our collection were presented in and provided by She et al. 2011 [22][7].

We compared ICPL with Algorithm 1 [11], CASA [7], IPOG [16], as implemented in the tool NIST ACTS, and MoSo-PoLiTe [17]. Each algorithm was run 100 times for each supported coverage strength, 1–3, with 100% coverage. For each run we recorded the size of the covering array and the time taken to generate it[8].

5.1 Software Setup

We have implemented an open source tool[9] with ICPL supporting 1–3-wise covering array generation from feature models. The algorithm is implemented as described in Section 4, including parallelization where applicable. In addition, our tool supports exporting input to Algorithm 1, CASA, NIST ACTS (IPOG) and MoSo-PoLiTe, and then invoking the tools to generate covering arrays. The results were interpreted by our tool and exported as a Comma Separated Values (CSV) file.

[6]The experiment was performed on a machine with four Intel Xeon CPU L7555 @1.87GHz with 8 cores with hyper threading, 128 GiB RAM. The activity on the machine was limited to 6 threads at 100% activity and 32 GiB of RAM. All the measurements have been done with open and freely available software.

[7]The FreeBSD feature model is partial.

[8]All the measurements are available online at `http://heim.ifi.uio.no/martifag/splc2012/`.

[9]The tool is available at `http://heim.ifi.uio.no/martifag/splc2012/`, implemented in Java.

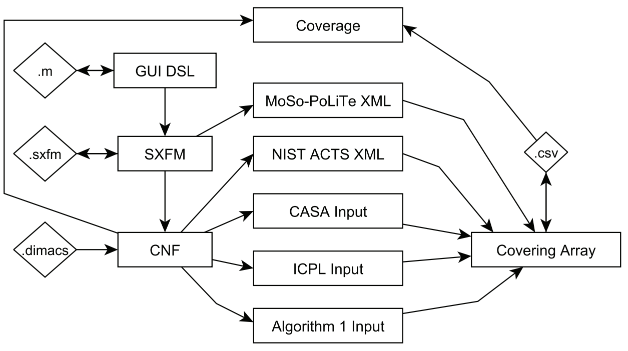


Figure 2: Transformations Used in the Experiment

Figure 2 shows the overview of the tool. The diagram is of no particular graphical modeling notation. The diamonds symbolize files with a certain suffix, the boxes symbolize internal data structures and the arrows symbolize transformations between the formats.

Since our model collection consists of feature models stored in three different formats, the tool accepts feature models in these formats: GUI DSL (model names suffixed with '.m'), as shipped with earlier versions of Feature IDE; SXFM, the Simple XML Feature Model format (model names suffixed with '.xml') and dimacs (model names suffixed with '.dimacs'), a file format for storing propositional formulas in conjunctive normal form (CNF).

The GUI DSL files can be loaded using the Feature IDE library. This library allows writing and reading of SXFM files. Thus, they can be loaded into the SPLAR library[10] along with other SXFM files.

The SPLAR library provides an export to conjunctive normal form (CNF), a canonical way of representing general propositional constraints. Thus all the previously loaded models can be converted into CNF formulas, along with other formulas stored in dimacs files.

Once a model is in the form of a CNF formula, it can be given to SAT4J, an open source tool for solving the SAT problem. Thus, all the feature models can be input to Algorithm 1 and ICPL, which works with feature models in the CNF format.

The three other tools that implement the algorithms with which ICPL is compared are available from their respective providers[11].

The covering arrays are written to a Comma Separated Values (CSV) file. The covering arrays can then be used to configure products for which single system testing is applied.

5.2 Verification of Covering Arrays

Our tool can measure the coverage of the covering arrays. It was ensured that each tool generated a covering array with 100% coverage by using a separate algorithm independent of any of the compared algorithms. The configurations in the covering arrays were also separately verified to be valid configurations. These two features are implemented in the tool used in the experiment and are available as a part of it.

[10]`http://splar.googlecode.com`

[11]MoSo-PoLiTe is a part of pure::variants provided by Pure Systems. CASA is available online at `http://cse.unl.edu/citportal/tools/casa/`. NIST ACTS is available online at `http://csrc.nist.gov/`

5.3 Generating Covering Arrays from Large Feature Models

Starting from the smallest feature model, we gave each algorithm 12 hours to complete the generation of a covering array[12].

ICPL is the only algorithm in our comparison that manages to generate a 1-wise covering array from the Linux kernel feature model (called `2.6.28.6-icse11.dimacs`), a 2-wise covering array from the three largest feature models, and a 3-wise covering array containing only t-sets with included features for two of the largest feature models. The results are shown in Table 2.

In Table 2, the measurements that are only available in the experiment from ICPL are shown with a gray background[13]. Notice that a 2-wise covering array can now be generated from a feature model with almost 7,000 features and 200,000 constraints. This is a big leap up from generating covering arrays from a feature model with close to 300 features and only 22 constraints, which is the largest feature model that one or more of the other algorithms managed to produce a 2-wise covering array from.

5.4 Comparison of the Five Algorithms

Using our experimental setup discussed earlier, all five algorithms were run through the same 19 feature models 100 times. If the time taken to generate 100 covering arrays was longer than three full days, the results were not included in the comparison. Figure 3 shows what part of the feature model corpus from which 100 covering arrays could be generated correctly[14] within three full days[15].

5.4.1 Comparison of Time Performances

Figure 4 shows the statistically estimated time performances. The numbers shown are the x'es in $O(f^x)$. Thus, for generating a covering array for a feature model with 1,000 features, a difference of 1 in the estimated value of x means that it takes 1,000 times longer to generate the covering array. To make the comparison fair, the basis for the statistical estimates were covering array sizes from the same 16, 14 and 12 feature models were compared for 1, 2 and 3-wise, respectively.

The fastest algorithm for all three strengths of covering ar-

[12]The available version of MoSo-PoLiTe does not handle feature models with propositional constraints above a certain level of complexity. Thus, we could not use it on five of the feature models in our corpus, including the four largest ones.

[13]In Perrouin et al. 2011 [18], it was reported that MoSo-PoLiTe generated a 3-wise covering array from the feature model with 287 features. That covering array was of size 841. In our experiments, we observed longer execution times for MoSo-PoLiTe than what was reported in Perrouin et al. 2011 [18], but it produced similar covering array sizes. ICPL produces a 3-wise covering array of size 108, an improvement in covering array size with a factor of 7.8. It was determined in personal correspondence with the authors that the available version of MoSo-PoLiTe might be different than the one used to generate the results in Perrouin et al. 2011 [18]. For our study, we used the only version available to us and provided by the authors on January 25. 2012.

[14]For five of the models, measurements are unavailable for MoSo-PoLiTe due to the complexity of the constraints in them. IPOG produced erroneous results for two of the features models. These results were therefore not included.

[15]IPOG and MoSo-PoLiTe does not support generating 1-wise covering arrays.

Table 2: Performance of ICPL on Large Feature Models: Feature Models, Satisfiability Times, Covering Array Sizes and Generation Times.
***Covering arrays for t-sets containing with only included features (1/8th of the t-sets), the memory usage was limited to 128 GiB instead of 32 GiB, and the algorithm was allowed to run in 64 processors at 100% activity instead of 6.**

Feature Model\keys	Features	Constraints (CNF clauses)	SAT time (ms)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	125	25	89	480	33,702	n/a	n/a
freebsd-icse11.dimacs	1,396	17,352	18	9	10	77	240	*78	*2,540
ecos-icse11.dimacs	1,244	2,768	12	6	2	63	185	*64	*973
Eshop-fm.xml	287	22	5	3	0.16	21	5	108	457

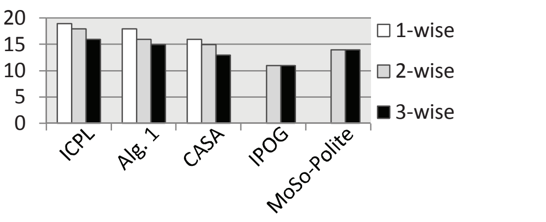


Figure 3: Out of the 19 feature models, how many of them the algorithms managed to correctly produce 100 covering arrays within three full days.

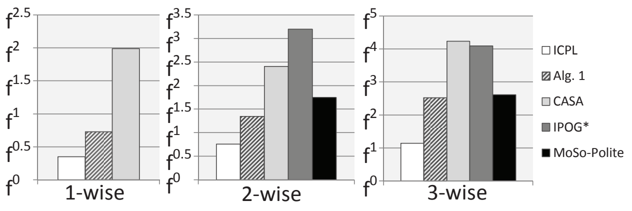


Figure 4: The statistically estimated values of x for the time taken for generating a covering array in the form $O(f^x)$ from a feature model with f Features. The times were for the same 16, 14 and 12 feature models for 1–3-wise covering arrays, respectively.
***The estimates for IPOG are extrapolated since it, as can be seen in Figure 3, managed considerably less feature models than the other algorithms.**

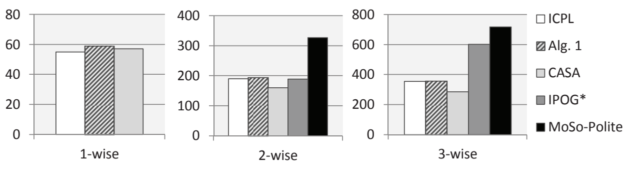


Figure 5: The sum of the sizes of the covering arrays for the same 16, 14 and 12 feature models for 1, 2 and 3-wise covering arrays, respectively.
***The estimates for IPOG are extrapolated since it, as can be seen in Figure 3, managed considerably less feature models than the other algorithms.**

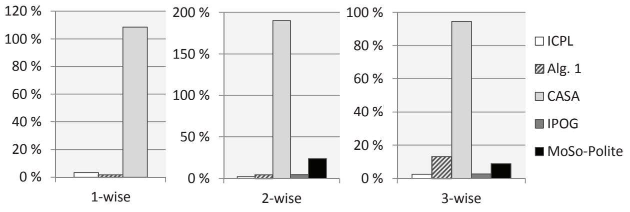


Figure 6: The sum of standard deviation of the 100 measurements of time over the sum of the average covering array generation times.

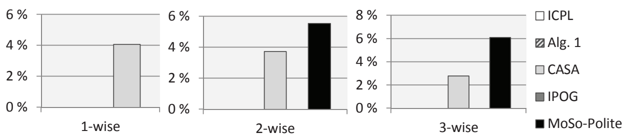


Figure 7: The sum of standard deviation of the 100 measurements of size over the sum of the average covering array sizes.

rays is clearly ICPL. Its statistically estimates performance is $O(f^{0.35})$, $O(f^{0.76})$ and $O(f^{1.14})$ for 1–3-wise covering array generation, respectively. Except for Algorithm 1 developed by us, the second fastest algorithm is MoSo-PoLiTe with a statistically estimated performance of $O(f^{1.75})$ and $O(f^{2.62})$ for 2 and 3-wise covering array generation.

5.4.2 *Comparison of Size Performances*

Figure 5 shows the sum of the sizes of covering arrays for the same 16, 14 and 12 feature models for 1, 2 and 3-wise, respectively. The sizes of 1–3-wise covering arrays are approximately the same for ICPL, Algorithm 1 and CASA. CASA is slightly better in this comparison. MoSo-PoLiTe produces larger covering arrays overall for both 2 and 3-wise covering arrays and IPOG for 3-wise covering arrays.

An interesting thing to notice here is that the size performance of ICPL and Algorithm 1 are almost identical. This is not a coincidence, as ICPL is Algorithm 1 with several speed optimizations.

5.4.3 *Effects of Non-Determinism*

Each algorithm might have some degree of non-determinism. It was in order to identify such effects that 100 measurements were taken for each tool for each feature model. These measurements provide us with the standard deviation both in covering array sizes and in generation times. This is shown in Figure 6 and 7. Note that the standard deviations for ICPL and for IPOG are some places as low as 0, meaning that they run the same time each time they are run.

Only CASA shows a significant standard deviation for the time taken. A standard deviation of 100% means that the time taken to generate a covering array is often as long as double that of the average time taken.

For the sizes of the covering arrays, the standard deviation is small for all algorithms.

6. THREATS TO VALIDITY

The authors of ICPL are also those who performed the comparison[16] with the other algorithms. Below we discuss things that might have influenced the result of the comparison.

Corpus Selection.

The feature model corpus consists of feature models not made by the authors of this paper. They are all used as originals except for some changes to the names due to differences in naming requirements in the different formats.

Corpus Completeness.

No feature models were excluded from the corpus except from their lack of realism. All feature models the authors were aware of and could verify the origin of were included in the corpus within the time available for conducting the experiment.

It is a possibility that the measurements are different given a more extensive corpus.

Adaptors.

Some of the tools came with limited documentation. The adaptors[17] that convert from the three formats of feature models in our corpus to input for each of the three other algorithms were written for the experiment in this paper and are available as open source. The results produced by each algorithm were confirmed to be complete and valid by the same, separate algorithms.

It is, however, possible that the adaptors are unfair to the other algorithms, and that the algorithms would have performed better with more suited converters. Had they been available from the implementations, they would have been used instead of the ones written by the authors of this paper.

Other Algorithms.

There are other algorithms which were not included in the comparison. The algorithm mAETG was neither available online nor by correspondence with the author. Some other algorithms which were not considered due to time constraints are the ATGT algorithm described in Calvagna and Gargantini 2009 [1], Microsoft's PICT Tool, and the algorithm in Grieskamp et al. 2009 [8] which is integrated into Microsoft's Spec Explorer for Visual Studio.

Other Strengths.

Empirics were not collected for $t >= 4$. One of the reasons for this is that the sizes of the covering arrays, using any algorithm, then becomes significantly larger than the number of features in the feature model from which it is generated. A second reason is that empirics indicate that 95% of bugs can be attributed to the interaction of 3 features [15]. If the quality requirements are above this level, it is not yet established that combinatorial interaction testing is the approach that should be taken in the first place.

[16]A complete, functional implementation of ICPL is provided as open source along with the scripts used to automatically accumulate the results presented. This makes it possible to completely reproduce all results reported in this paper. Also, all the measurements done in our experiment are available online. Implementations of the other algorithms are available from the respective sources.

[17]The adaptors are available as open source from `http://heim.ifi.uio.no/martifag/splc2012/`.

7. FUTURE WORK

Make a Specialized Feature Model Satisfier.

Since feature models are easy to configure but might be hard to convert to CNF [24], a specialized configurator for feature models that exploits the specific properties of feature models to satisfy it without converting it to a CNF formula would solve the problem of not converting the formula to CNF.

Find Additional Improvements.

We have not ruled out further improvements upon ICPL. ICPL is made available as open source so that it can be used for further research.

Finding a way to parallelize the part of ICPL which is not parallelized would increase the speed further.

Find a More Efficient t-set Storage.

A clear improvement to ICPL is more effectively storing the t-sets. Storing the t-sets is the cause of the high memory requirement of the algorithm.

8. CONCLUSION

In this paper, an algorithm for t-wise covering arrays, called ICPL, was presented. ICPL handles some of the largest feature models available, produces covering arrays of an acceptable size and has low standard deviation due to its non-determinism.

ICPL was compared to three other algorithms that handle the same problem. Within 12 hours, ICPL was the only one to handle the three largest feature models for 2-wise covering array generation. Within 3 whole days, each algorithm was run 100 times for each feature model. ICPL has the highest scalability overall for 1–3-wise covering array generation and generates covering array quickly. No variation in covering array size was observed, and the time taken varies insignificantly. Thus ICPL enables the application of combinatorial interaction testing to significantly larger software product lines than before.

9. 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 Ina Schaefer and the anonymous reviewers for their helpful feedback.

The authors would also like to thank Daniel Le Berre for his assistance on SAT4J, Brady J. Garvin for answering questions related to CASA and mAETG, and Sebastian Oster for answering questions related to MoSo-PoLiTe.

10. REFERENCES

[1] A. Calvagna and A. Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In C. Dubois, editor, *Tests and Proofs*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer Berlin / Heidelberg, 2009.

[2] V. Chvátal. A greedy heuristic for the Set-Covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[3] M. B. Cohen, M. B. Dwyer, and J. Shi. 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.

[4] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

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

[6] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 177–188, New York, NY, USA, 2009. ACM.

[7] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Softw. Engg.*, 16:61–102, February 2011.

[8] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. Cohen. Interaction coverage meets path coverage by smt constraint solving. In M. Núñez, P. Baker, and M. Merayo, editors, *Testing of Software and Communication Systems*, volume 5826 of *Lecture Notes in Computer Science*, pages 97–112. Springer Berlin / Heidelberg, 2009.

[9] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[10] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[11] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In J. Whittle, T. Clark, and T. Kuehne, editors, *Proceedings of Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*, pages 638–652, Wellington, New Zealand, October 2011. Springer, Heidelberg.

[12] M. F. Johansen, Ø. Haugen, and F. Fleurey. A survey of empirics of strategies for software product line testing. In L. O'Conner, editor, *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, pages 266–269, Washington, DC, USA, 2011. IEEE Computer Society.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[14] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[16] Y. Lei, R. Kacker, D. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 549–556. IEEE, 2007.

[17] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In J. Bosch and J. Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2010.

[18] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 2011.

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

[20] A. Reuys, S. Reis, E. Kamsties, and K. Pohl. The scented method for testing software product lines. In T. Käkölä and J. C. Dueñas, editors, *Software Product Lines*, pages 479–520. Springer, Berlin, Heidelberg, 2006.

[21] J. Rivieres and W. Beaton. Eclipse Platform Technical Overview, 2006.

[22] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 461–470. ACM, 2011.

[23] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309 –322, may-june 2010.

[24] T. Walsh. Sat v csp. In R. Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer Berlin / Heidelberg, 2000.

Chapter 9

Paper 3: Bow Tie Testing: A Testing Pattern for Product Lines

Bow Tie Testing - A Testing Pattern for Product Lines

MARTIN F. JOHANSEN, SINTEF and Institute for Informatics, University of Oslo
ØYSTEIN HAUGEN, SINTEF
FRANCK FLEUREY, SINTEF

Verification of highly configurable systems poses a significant challenge, the challenge of knowing that every configuration works when there often are intractably many different configurations. When a homogeneous abstraction layer has many mutually exclusive alternative implementations, we might, according to the polymorphic server test pattern, test these implementations using one test suite targeted towards the abstraction layer which is then run for each concrete implementation of the abstraction layer. But, the pattern does not handle interaction testing. Combinatorial interaction testing is one of the more promising techniques for doing interaction testing of a software product line. The BOW TIE TESTING PATTERN describes how the configurations which differ only in the implementation layer require one test suite. In addition, comparing the execution results of one product with another provides for a test oracle. The pattern reduces the effort of testing a highly configurable system without reducing the error detection capabilities provided by ordinary combinatorial interaction testing. We present an example of a subset of the Eclipse IDE product line, and show that only 20 test suites is required to test 41 products, a significant reduction.

Categories and Subject Descriptors: D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Testing tools*; D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures—*Patterns*; D.2.13 [**SOFTWARE ENGINEERING**]: Reusable Software—*Domain engineering*

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Software product lines

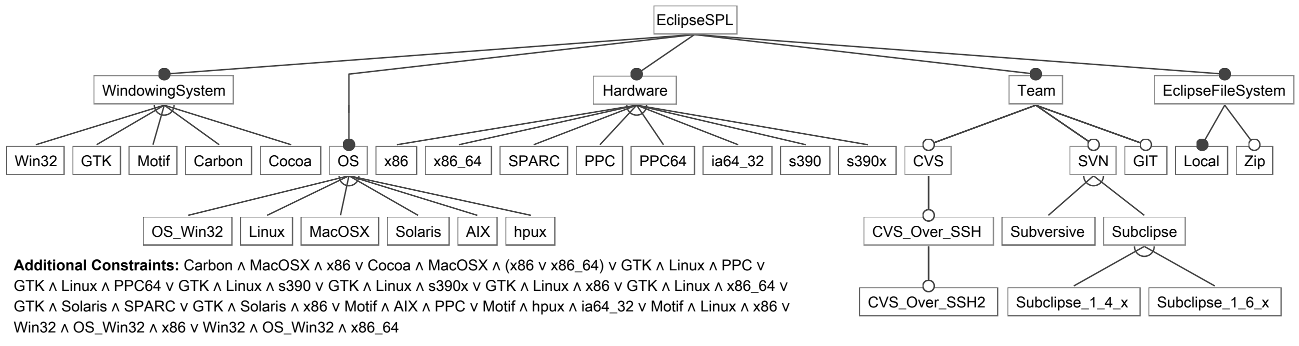
ACM Reference Format:

Johansen, M. F., Haugen, Ø and Fleurey, F. 2011. Bow Tie Testing - A Testing Pattern for Product Lines jn 16, 1, Article (July 2011), 13 pages.

1. INTRODUCTION

Today, there is no recommended approach from the software product line research community on how to verify software product lines, but there are a number of suggestions. The BOW TIE TESTING PATTERN is a partial solution to the problem of verifying product lines within the context of one of the most promising approaches, combinatorial interaction testing, explained in Section 2. A complete approach for verifying product lines would probably consist of a number of patterns and techniques. The BOW TIE TESTING PATTERN identifies a commonly occurring structure found within product lines, which we call a bow tie, presented in Section 3. We exploit the nature of this structure when present to improve upon combinatorial interaction testing. This is presented in Section 4 as the BOW TIE TESTING PATTERN. The paper finishes with a short discussion of how a product line can be restructured to enable the application of the BOW TIE TESTING PATTERN, Section 5; and a look at similar patterns, Section 6.

The work presented here has been developed within the VERDE project ITEA 2 - ip8020. VERDE is a project within the ITEA 2 - Eureka framework.
Author's address: Martin F. Johansen, Øystein Haugen, Franck Fleurey; SINTEF ICT, Pb. 124 Blindern, 0314 Oslo, Norway; email: {Martin.Fagereng.Johansen, Oystein.Haugen, Franck.Fleurey}@sintef.no



Additional Constraints: Carbon ∧ MacOSX ∧ x86 ∨ Cocoa ∧ MacOSX ∧ (x86 ∨ x86_64) ∨ GTK ∧ Linux ∧ PPC ∨ GTK ∧ Linux ∧ PPC64 ∨ GTK ∧ Linux ∧ s390 ∨ GTK ∧ Linux ∧ s390x ∨ GTK ∧ Linux ∧ x86 ∨ GTK ∧ Linux ∧ x86_64 ∨ GTK ∧ Solaris ∧ SPARC ∨ GTK ∧ Solaris ∧ x86 ∨ Motif ∧ AIX ∧ PPC ∨ Motif ∧ hpux ∧ ia64_32 ∨ Motif ∧ Linux ∧ x86 ∨ Win32 ∧ OS_Win32 ∧ x86 ∨ Win32 ∧ OS_Win32 ∧ x86_64

Fig. 1. Feature model for a subset of Eclipse

2. BACKGROUND AND MOTIVATING EXAMPLE

2.1 Software Product Lines

A software product line [Pohl et al. 2005] is a collection of systems with a considerable amount of code 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 code during production. It is also common for customers to have conflicting requirements; in which case it is not even possible to ship one system for all customers.

The Eclipse products [Rivieres and Beaton 2006] can be seen as a software product line. Today, Eclipse lists 12 products on their download page, `eclipse.org/downloads/`. 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 would not likely need to use the PHP 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. A subset of the Eclipse product line will be used as a running example in this paper.

One way to model the commonalities and differences in a product line is using a feature model [Kang et al. 1990]. 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. See for example 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, for Eclipse, one have to chose one windowing system which one wants Eclipse to run under. This is modeled as an empty semi-circle on the outgoing edges. When one is to choose a team-functionality provider, one can choose none or all. This is modeled as an empty circle at the top of each node. This means that that feature is optional. A filled circle means that the feature is mandatory. One have to configure the feature model from the root, and one can only include a feature when the preceding feature is selected. For example, supporting CVS over SSH requires that one has CVS. In addition, if there are constraints that are impossible or impractical to put in the three-structure of the feature model, textual constraints can be added below the three. These are written in common propositional logic. In the working example, not all combinations of windowing systems, OSes and hardware are supported. Therefore, only those that are supported are allowed in a long propositional constraint.

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. Such a configuration consists of specifying whether each feature is included or not.

2.2 Verification of Software Product Lines

Verifying a software product line poses a number of new challenges compared to verifying single systems. One has to ensure 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 software product line is simply a collection of many systems. One cannot test each possible system, since the number of systems in general grows exponentially with the number of optional features in the product line. In our running example, the number of possible configurations is 1024, and this is a relatively simple product line.

There is no single recommended approach available today for verifying product lines efficiently [Engström and Runeson 2011], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [Cohen et al. 2008], discussed below; reusable component testing, frequently seen in industry [Johansen et al. 2011], 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 [Reuys et al. 2006], which has seen recent development as ScenTED-DF [Stricker et al. 2010] for detecting redundant test cases based on data path analysis of the product line's UML model; 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 [Uzuncaova et al. 2010].

2.3 Combinatorial Interaction Testing

One of the most promising approaches for verifying product lines is called *combinatorial interaction testing* [Cohen et al. 2008]. The benefit of this approach is that it deals directly with the feature model and then allows for the application of single system testing techniques, of which there are many good ones. 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 interaction between a pair of features is tested. This includes testing the cases where both features are present, when one is present, and when none are present. Table I shows the 41 products that one must test to ensure that every pair of interaction between the features in the Eclipse product line 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 feature are included for every product because they are mandatory. The last row contains 'R's. The significance of these will be explained later.

Testing every pair is called 2-wise testing, or pair-wise testing. A covering array that contains products with every pair is called a 2-wise, or pair-wise, covering array. This is a special case of t-wise coverage where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in one product. 3-wise coverage means that every combination of three features are tested. For our running example, 119 products are required to achieve 3-wise coverage.

[Kuhn et al. 2004] showed 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%, for 3-wise around 95%, etc. Thus, it is reasonable for product lines not highly critical to aim for 3-wise testing.

It should be noted that [Kuhn et al. 2004] result is about combinations of program input and configuration variables. This is not precisely feature interactions, but until proper confirmation of these results for product lines are available, these are the best indications for the probability of faults occuring in feature interactions of a low degree.

Table I. Pair-wise coverage of the feature model in Figure 1 the test suites numbered

Feature\Product	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
EclipseSPL	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WindowingSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Win32	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	-	-	-
GTK	-	X	X	X	X	X	-	-	-	X	X	-	X	-	-	X	X	-	X	-	-	-	-	X	X	X	-	X	-	-	X	X	-	-	-	-	X	X	X	-	-
Motif	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	X	-	-	X	X	X	-	-	-	-	-	X	-	-	-	X	-	X	X	-	-	-	-	-
Carbon	-	-	-	-	-	-	-	X	-	-	-	X	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-
Cocoa	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	X
OS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OS_Win32	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	-	-	-
Linux	-	X	-	X	-	X	-	-	-	X	-	-	X	-	-	X	X	-	X	-	-	-	-	X	X	X	-	X	-	-	-	X	X	-	-	-	X	X	-	-	-
MacOSX	X	-	-	-	-	-	-	X	-	-	-	X	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	X	X
Solaris	-	-	X	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-
AIX	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
hpux	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-
Hardware	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
x86	X	-	X	-	-	-	-	X	-	-	-	X	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	X	-	-	-	-	-	-	X	-
x86_64	-	-	-	-	-	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	X	-	-	-	X	-	-	-	-	-	-	X
SPARC	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-
PPC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	X	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
PPC64	-	X	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-
ia64_32	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-
s390	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	X	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
s390x	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-
Team	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	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	X	X	X	X
CVS_Over_SSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS_Over_SSH2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X
SVN	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	X	X	-	X	X	X	X	X	X	X	X	X
Subversive	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-	X	X	X
Subclipse	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	X	X	-	X	X	X	X	X	X	-	-	-
Subclipse_1_4_x	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-	-	-
Subclipse_1_6_x	-	-	-	-	-	-	-	-	-	X	X	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	X	X	X	X	X	-	-	-	-
GIT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	X	-	X	X	-	-	X	X	-	-	X	X	X	-	-	X	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	X	-	-	X	X	-	X	X	-	-	-	X	X	-	-	X	X
		R	R	R	R	R	R	R	R		R	R	R		R	R	R	R	R				R	R										R			R				R

3. THE BOW TIE STRUCTURE

There is a certain structure that is repeatedly occurring in industrial software product lines: Using an abstraction layer with multiple implementations is a way of supporting various different systems generically, so that one can on one side use the same code to interact with different systems. If this interaction is generic then one is avoiding writing redundant code and one can work with one code base above the abstraction layer. The mechanism providing this generic interface to various different implementations is called a homogeneous abstraction layer.

A homogeneous abstraction layer is an interface which implementations yield essentially the same functionality no matter which concrete implementation of it one is using. The code accessing the functionality through the abstraction layer does not have to be concerned with which concrete implementations it is. Examples are the widgets of Linux and Windows which both provide buttons, radio buttons and check boxes. The rendering and the technology for showing these to the user is different, but the same basic functionality is provided to software using an abstract button, radio button and check boxes.

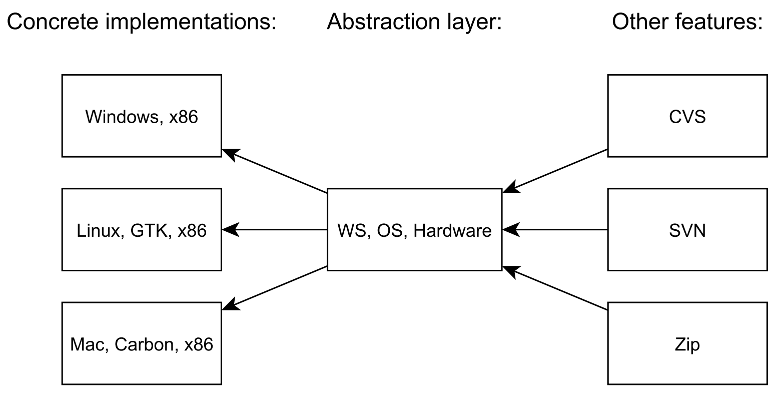


Fig. 2. The Bow tie structure

If the functionality is essentially different, then the abstraction layer is not homogeneous. For example, two layout algorithms might yield different layouts, in that case what is returned by the concrete layout algorithms is different, and thus the abstraction layer is not homogeneous.

In the running example, the particular features for each windowing system, OS and Hardware are all used through an abstraction layer. The features implementing support for team functionality, CVS, SVN and git in our example, all interact with the abstraction layer only, but for each product of Eclipse, one of the concrete implementations of the abstraction layer must be present.

This forms a structure similar to a bow tie, as shown in Figure 2. The components on the right side are using one of various implementations of an abstraction layer, but only depend on and interact with the abstraction layer.

In the following quote, [Cabral et al. 2010] explains the difficulty of testing configurable systems with mutually exclusive alternative features, of which alternative implementations of an abstraction layer is a special case. "Alternative features are features that are mutually exclusive and present a more difficult challenge for testability. We argue that these are the true deterrents to testability since only one feature can be present in an SPL [Software Product Line] at a time."

4. PATTERN: BOW TIE TESTING

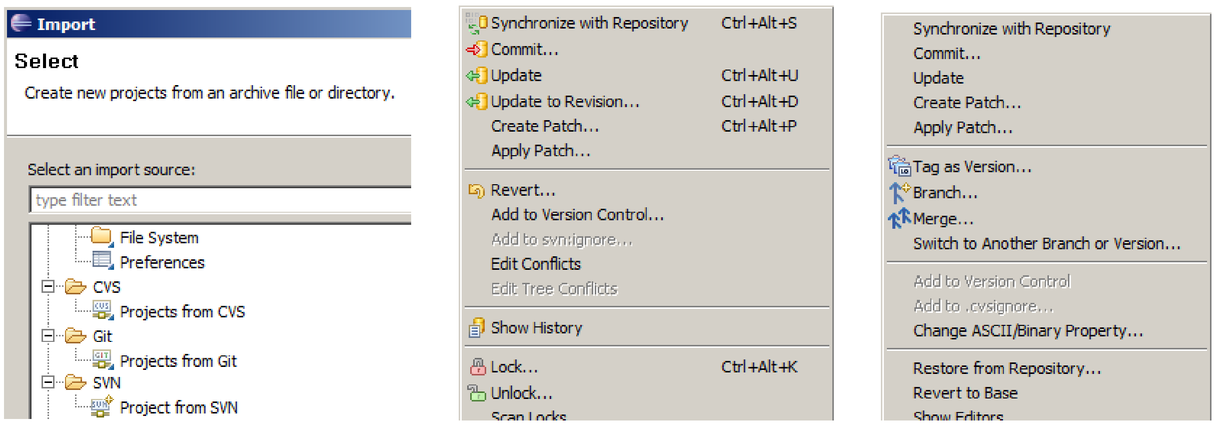
4.1 Context: The circumstances under which the problem appears

One's software product line includes different implementations of one or more homogeneous abstraction layers. It is important that the other parts of the product line use the concrete implementations through the abstraction layer only. Thus, one is able to identify a bow tie structure where the left side contains the concrete implementations of the abstraction layer, and the right side which includes the other features that interact with the concrete implementations of the abstraction layer. The abstraction layer itself is the knot of the bow tie.

Discussion. A software product line is typically configured by the provider of the product line. Highly configurable systems typically allow the customer to configure the system at run time. This does not lessen the importance of testing that every configuration works.

Having to support multiple versions of the same feature is usual in software development. The alternative versions of the same feature can be seen as mutually exclusive alternatives, and thus qualify as (partial) implementations of a homogeneous abstraction layer (the left side the bow tie structure).

Supported configuration alternatives deferred to the user also applies. For example, a 32-bit Windows executable will run on all supported versions of the Windows operating system today. This does not lessen the importance of testing interactions between the features of a product line and the different versions of windows.


Import

Select

Create new projects from an archive file or directory.

Select an import source:

(a) Import a new project (b) Part of the SVN Team Menu (c) Part of the CVS Team Menu

Fig. 3. Example of an invalid context in the Eclipse IDE

In our running example we have identified the abstraction layer and the left and right sides of the bow tie structure in Figure 2.

If one has a structure where some of the interactions are directly to a concrete implementation, or the abstraction layer is only partly homogeneous, one might consider restructuring the product line so that one gets a clear separation and a homogeneous abstraction layer. Such restructurings are discussed in Section 5. Then, one can apply the pattern.

Examples of valid contexts

—The Eclipse IDE, exemplified by our running example, runs on 16 different combinations of windowing system, operating system and hardware, but the behavior of the system is the same regardless of how the buttons and windows are rendered.

—Apache Commons Virtual File System is an abstraction layer to more than ten different files systems. Being a part of Apache Commons, this abstraction layer is used by many other components.

—One of our case studies is a tax reporting system that requires accounting information. It has a homogeneous abstraction layer to 17 different accounting systems. The accounting system abstraction layer is used by many other components, and it is an important part of its architecture that the abstraction layer is homogeneous.

Examples of an invalid context. In our running example, the three types of version control systems: CVS, SVN and git, could potentially be accessed through a homogeneous abstraction layer. But, the Eclipse project has decided to allow the different version control systems to provide varying functionality. Thus, that part of the product line is not within the context of the pattern.

Figure 3 shows three windows from Eclipse. Figure 3(a) shows that when importing a project, one does not have a generic import wizard, rather one have one for each type of team functionality, one for CVS, SVN and git. This is because they provide different functionality. Figure 3(b) and 3(c) shows the context menus for SVN and CVS project respectively. Many of the alternatives are similar, such as commit and update, but many of the alternatives are also different, such as specifying whether you are working with text or binary files, not needed for SVN. Providing a single common context menu for CVS and SVN clearly would retain the majority of the features.

4.2 Problem

The homogeneous abstraction layers in a software product line requires that one has exactly one implementation present at any time, the number of possible configurations of the software product line

grows quickly with the number of alternative implementations. How can one execute tests against a software product line containing such a structure to know that all products in the product line will work? Combinatorial interaction testing allows us to test feature interactions, but alternative features deters its performance. Can one deal with the increase caused by the alternative implementations of an abstraction layer?

4.3 Forces

—**Testing Several Feature Interactions:** Highly configurable systems usually have an enormous number of potential configurations. One cannot simply build all of the systems and test them. The number of systems grows exponentially with the number of features in, for example, a feature model defining a product line. Mutually exclusive alternative configuration options increase the number of possible configurations significantly. The reason is that *the ability to test several feature interactions at once is reduced* due to the mutual exclusion between the options.

—**Redundant Work:** Testing each alternative independently *results in redundant work* in the creation of the test suites because of the similarities between the implementations of the abstraction layer.

—**Test Suite Increase:** If one adds a new configuration option to the system, how then must the test suite of the system be modified, now maybe with an order of magnitude more possible configurations? In essence, *adding a new option increases the number of test suites* required to test the system.

—**Almost Identical Test Suites:** Even though one has several implementations of the same abstraction layer, how can one reuse the knowledge from testing one of them when testing another? Even if the abstraction layer is homogeneous, the concrete implementations are different. *The test suites for each particular alternative are not exactly the same*, nor do they necessarily execute in a similar manner such that their execution traces can be compared.

—**Versions Add Variability:** Often, configurable systems must operate with several versions of the same component. Different companies have different policies about adopting a new version of a component. A company delivering a system might have to support an old version of a component for some time. *Different versions of the same implementation of an abstraction layer increases the configuration space* of the system even more.

—**Configuration Diversity:** One must make sure that a particular configuration within the intractable configuration space does not contain a bug, and *one cannot predict how the next customer will configure the system*.

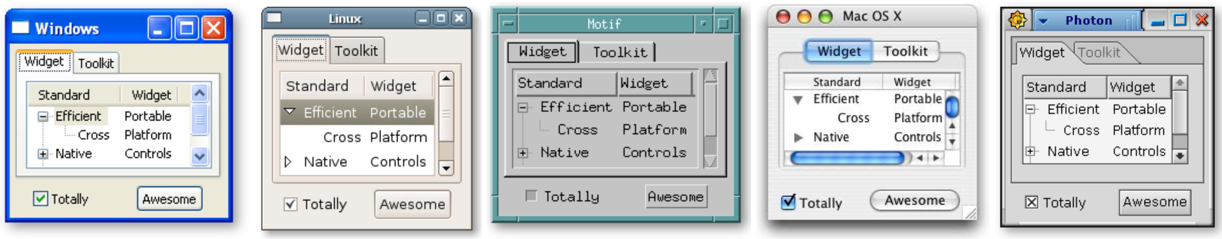
4.4 Solution

Perform interaction testing by testing each product of a t-wise coverage array of the product line. In the coverage arrays, whenever configurations differ only inside a homogeneous abstraction layer, create one test suite and execute it against those products and expect it to pass for all these products. Make these products log the generic parts of their execution, and have them vote for the correct execution using a voting oracle.

4.4.1 *Further Details.* It is a good idea to start with 1-wise and proceed as far as necessary according to how well tested the system must be. [Kuhn et al. 2004] reports empirics about the percentage of bugs expected at each stage.

Expect the number of products that can be tested by the same test suite to increase as one increases the coverage strength, but to decrease as one increases the number of optional features not inside an abstraction layer.

Additionally, one can add a voting oracle [Binder 1999]. Say one is executing the same test suite against three products. Make the products log the generic parts of their execution. After the three test suites have been run, compare the executions of each product. If there is a disagreement in the execution, then one has either logged non-generic information or there is a bug in at least one of the implementations of the abstraction layer.



(a) Windows (b) Linux, GTK (c) Linux, Motif (d) Mac (e) Mac, Photon

Fig. 4. Different products in a product line using SWT's abstraction layer (Taken from http://www.eclipse.org/swt/, May 2011)

4.4.2 *Example 1.* The pair-wise covering array of the running example is shown in Table I. The products differing only in the implementations of the abstraction layer are the 21 products with an R written in the last row. For the products with an R, the test suite from the previous product can be used to test it. Instead of 41 test suites, one now only require 20. For example, test suite number 1 tests Eclipse without any optional plug-ins. The eight following products only differ below the abstraction layer. These products can be configured and set up on their respective platforms. The test suite is programmed and as the tests are run, the execution traces for each is written to a file. If one of the test suites fails for one product, but succeeds for the other, then one knows there is a problem in one of the implementations of the abstraction layer. If the execution logs differ, one must look into the differences and examine why it happened.

Covering the system with 1-wise testing gives 10 products and only 5 test suites required.

For 2-wise testing, with 41 configurations, only 20 test suites were required as just discussed.

For 3-wise testing, with 119 configurations, only 45 test suites are required.[1]

4.4.3 *Example 2.* Say one has configured the five applications shown in Figure 4. These applications, as can be seen, differ only in their implementation of the rendering engine, the hardware and the interaction with the OS.

If one makes a test case which asks to select the fourth element in the list, select the check box and press the button, one expects the same result in terms of the action invoked by pressing the button in all five cases. It is likely that a bug in one of the interactions with the OS, one of the renderings of the graphics or one causes an error in one of the five applications.

4.5 Consequences

4.5.1 *What happens when the solution is applied*

—**Testing Several Feature Interactions:** The number of test suites needed to do combinatorial interaction testing of the product line is reduced. When two configurations in, for example, a pair-wise coverage of a product line has different combinations inside the abstraction layer, but the same outside, one test suite can be written that can be executed for both configurations inside the abstraction layer. T-wise coverage ensures that, for example, every pair of features is tested together. By applying the pattern, one *acquires the ability to test several feature interactions* with the same test suite.

—**Redundant Work:** One knows how to test in situations of highly configurable systems where one has an abstraction layer and multiple implementations. *One avoids redundant work* since one does not have to create tests that tests similar functionality many times.

[1] A tool for generating covering arrays and the reduction is freely available online as open source at `http://heim.ifi.uio.no/martifag/europlop2011/splcatool/`

—**Test Suite Increase:** If one decides to add a new implementation of the abstraction layer, *the test suite for this implementation exists already and can be reused*. In regards to interaction testing, some of the configurations required for, for example, pair-wise coverage can be reused.

—**Almost Identical Test Suites:** *It is possible to compare the generic execution traces of the abstraction layer* to verify that the implementations of the abstraction layer provide functionality that makes the abstraction layer homogeneous.

—**Versions Add Variability:** The pattern also applies to configurable systems where there are several versions of the same feature in use with compatible interfaces. These *differing versions can be tested with the same partial test suite*, and the results for each version can be compared, a well-known regression testing technique.

—**Configuration Diversity:** Applying combinatorial interaction testing to cover all pair and triples etc. of interaction *makes one know that whatever pairs and triples etc. the customer chooses, they will work together*.

4.5.2 *Liabilities of the solution*

—The solution does not make sense if the abstraction layer contains the union of functionality of the implementations. The intersection has to be considerable in order for the application of this pattern to pay off.

—The abstraction layer cannot be too general. For example, the team provider functionality in Eclipse has multiple implementations: CVS, SVN, git, etc. But, the functionality the implementations offer is different. Thus, one have to supply one test suite per provider, and that is also what is done in the Eclipse project.

—If one decides to make available implementation-specific behavior in the abstraction layer, then the benefits of the pattern no longer applies, and one will have to significantly change the test suites.

—Even though the test cases might be the same for all implementations, the initializations of the test suites might be different. An example of this is the Apache Virtual File System test suite. It needs to set up different file systems before executing the generic test suite.

4.6 Known uses

There are no known uses of this pattern in its entirety, but the following three examples partly use the BOW TIE TESTING PATTERN, and is therefore included to show that all benefits of the pattern is present partly when the pattern is partly used.

4.6.1 *Eclipse Standard Widget Toolkit (SWT).* A part of this pattern is used to test the Eclipse Standard Widget Toolkit (SWT 2011), used in Eclipse as an abstraction layer on top of differing windowing systems, operating systems and hardware. Figure 4 shows the same application running in five different configurations.

Even though there are many more possible combinations, Eclipse has only implemented 16 of the combinations of windowing system, operating system and hardware; thus, the feature model contains additional constraints restricting which combinations can be chosen (see Figure 1). Eclipse SWT is 14% of the total Eclipse Platform Source Code.

The test suite of SWT contains over 5,600 test cases, with statement coverage of 32%. This test suite package only requires the abstraction layer package in order to execute.

This test suite is reused for each configuration with a unique combination of windowing system, OS and platform.

In Eclipse, testing of other components as a part of Eclipse is not dependent on a specific windowing system, OS or platform. Thus, these test suites can be reused for each concrete implementation of the abstraction layers towards windowing system, operating system and hardware.

Table II shows the resulting statement coverage for running the single SWT test suite on three different configurations of Eclipse. Notice that the coverage is similar for the concrete implementations. (It is unrealistic to achieve 100% statement coverage since, for example, many special cases are handled in exception blocks.)

Table II. Statement coverage of executing the SWT generic test suite for three concrete implementations

Element (Java only)	Statement Coverage	Covered Instructions	Total Instructions
org.eclipse.swt.internal.motif (Linux, 32-bit)	59.80 %	5,242	8,763
org.eclipse.swt.internal.win32 (64-bit)	44.30 %	2,455	5,537
org.eclipse.swt.internal.gtk (Linux, 32-bit)	45.40 %	9,938	21,877

4.6.2 *Apache VFS.* Apache Commons Virtual File System (VFS) supports ten different file systems and uses one test suite which interfaces with the abstraction layer. Even though the test cases are the same for all the implementations, the initializations of the test suites are specific for each file system. The implementation that enables file system access to tar-files sets up an example tar file, while the memory file system loads a file into memory. The specific initialization accounts for a small amount of the total test code for each implementation.

4.6.3 *Tax reporting systems.* A tax reporting system has been studied which is a product line consisting of nine main products which address many problems in the reporting of accounting data to the public authorities. These nine products defer many configuration options to users, making the actual number of configurations intractable. The product line is developed by Finale Systems, a Norwegian company which specializes in this domain.

They use the checking system discussed above to compare the execution of their system on different platforms. They support all the versions of Windows that Microsoft supports. Their system is a 32-bit binary for windows which is supposed to run on all supported versions of Windows, 7 in total that in addition are supplied as both 32-bit and 64-bit. They have implemented a special mode of execution-tracing called the "test mode" which trace is logged and then compared using a series of diff-scripts. They only log partial information about the execution. They log those parts of the execution that is meaningful to compare. They also filter out information that is difficult to compare, such as time stamps.

If the diffs of the execution shows a difference, then they know that they should pay attention to the difference, something that the domain experts have a simple time understanding and handling at the company.

5. RESTRUCTURING TO ENABLE THE APPLICATION OF THE PATTERN

If one has a highly configurable system with an abstraction layer that is only partially homogeneous, one might consider restructuring into a more homogeneous abstraction layer where all implementations have the same or almost the same intersection of functionality. This will enable the application of the BOW TIE TESTING PATTERN with the benefits it provides.

Take our running example in Figure 1. Many companies only use one source control system in house. Thus, a thing that can be done is to create a homogeneous abstraction layer for the team provider functionality and demand that there has to be exactly one team provider available. The new context menus would then look something like Figure 6 instead of Figure 3. We can see that there is only one generic alternative for importing a new project and only one generic context menu for all source control systems.

The change of the team feature and its sub-features is shown in Figure 5. The sub-features of Team are now mutually exclusive alternatives, and the CVS options are set to mandatory. Thus, there are five different alternatives for a source control systems behind a homogeneous abstraction layer.

Such a restructuring will have a dramatic effect on the testability of the software product line. For pair-wise coverage, only two test suites are required to test 50 different products for pair-wise coverage.

This would not be a refactoring since we have changed the possibilities of configuration, but it might be a valid trade-off to decrease the number of test suites required for t-wise coverage.

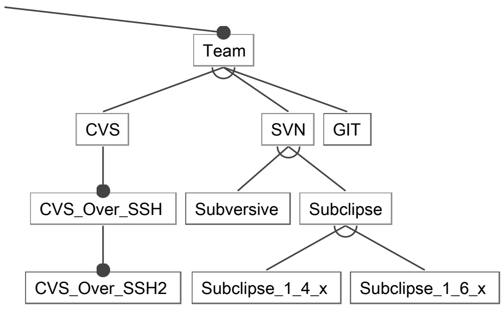
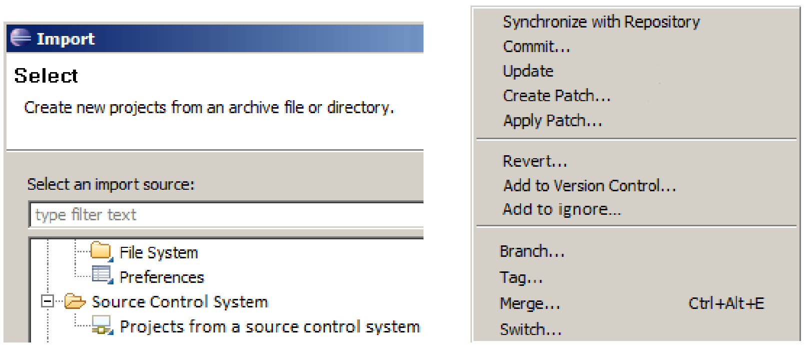


Fig. 5. A part of the running example has been changed to enable the application of the BOW TIE TESTING PATTERN



(a) Import a new project (b) Part of the Generic Team Menu

Fig. 6. Restructured team menus from Figure 3 for the Eclipse IDE

6. SIMILAR PATTERNS

6.1 Polymorphic Server Test Pattern

The purpose of this pattern [Binder 1999] is to verify that a collection of classes are compliant to the Liskov substitution principle. Compliance to this principle basically means that any of the classes can be substituted for the others and the rest of the system should continue to function in the same way. For example, if one has an abstract class Window, then the code using an instance of this class continues to function equally well whether it is an instance of GTKWindow or WindowsWindow. Thus, since the code using an instance of the abstract class should work equally well for any concrete instance, we can make a single test suite for the abstract class, and rerun it for every subclass. The test suite should give the same verdict no matter which concrete class is an instance of the abstract class.

Binder calls the collection of classes implementing an abstract class a polymorphic server hierarchy. It is polymorphic because that is the basic functionality used for inheritance, it is a server since it provides functionality, and it may be a hierarchy since more than one level of inheritance is possible.

For the running example, we may apply the pattern to the windowing system by creating a single test suite which interacts with the windowing system abstraction layer. We can then instantiate this abstraction layer with each windowing system and rerun the test suite.

Thus, the pattern does not deal with interaction testing, which is what the pattern in this paper does.

6.2 Voting Oracle

The voting oracle [Binder 1999] compares the output of several versions of the same system, and decides that the one which is likely faulty is the result with the fewest votes. This pattern integrates into our pattern by having the products tested by the same test suite vote for the correct result. The use of a voting oracle with our pattern was discussed in the description of the pattern above.

6.3 Parallel Test Oracle

The parallel test oracle [Binder 1999] is kind of gold standard oracle. It is similar to the Voting Oracle except that one knows that one implementation gives the correct results. Our pattern integrates with the parallel test oracle in cases when one knows that one implementation is correct. If a feature has been verified and one decides to upgrade the system using more modern technologies, the old version will serve as a gold standard oracle. This is realistic in a product line setting, since not all customers might be ready to upgrade to the new technology, or one simply wants to continue to support the old technology.

6.4 Abstract Class Test

Abstract class testing [Binder 1999] deals with the many uses of an abstract class. Using an abstract class as an abstraction layer is only a special case. Binder does discuss this special case briefly.

> Test cases may be developed and run on the superclass method, and then they may be reused for each subclass. Integration testing should be done at flattened scope for each subclass of the abstract class.

However, Binder does not elaborate any more on this.

In the running example, some abstract classes might serve as the abstraction layer for the concrete windowing system implementations. Then, the abstract class test is applicable to those abstract classes. However, when doing integration testing in the context of a product line, additional guidance is needed in order to know which integrations to focus on. This is what the BOW TIE TESTING PATTERN provides in addition.

6.5 New Framework Test Pattern

The new framework test pattern [Binder 1999] deals with testing a newly developed framework. A framework is typically a collection of classes serving as a skeleton implementation of a wide range of application. For example, a framework for a website might include the basic functionality of navigation and content administration. Since this is common among many websites, implementing this functionality well once and then reusing it makes sense.

The problem one faces after having developed a new framework is verifying that any accepted usage of the framework classes works. In this sense, testing a framework is similar to testing a software product line. It is, however, different in the sense that one does not have a single implementation to work on. The new framework test pattern suggests making a demo application and then test it to also test the framework.

6.6 Generic Class Test Pattern

Generic classes are used to not have to implement concrete subclasses for every supported type, and to support an indefinite number of types. Thus, one typically has only one generic class and no concrete implementations. Thus, the BOW TIE TESTING PATTERN does not apply because it relies on the fact that one has a lot of specific code that one should not have to redundantly test. Testing a generic class is a different challenge which is addressed by the generic class test pattern [Binder 1999].

7. CONCLUSION

The BOW TIE TESTING PATTERN shows how mutual exclusive alternatives might be tested using the effort of testing one implementation while keeping the error detection capabilities of having one test suite for each implementation.

We found several benefits of our approach. We found that:

—Using a homogeneous abstraction layer with many alternatives increases the variability of a product line while it does not reduce the testability.

—The number of products required to test when performing t-wise testing of a product line is reduced in many cases when there are abstraction layers in the product line.

—Adding new implementations of an abstraction layer in a product line does not increase the number of test suites required to perform t-wise testing.

—In the cases where one test suite can be used to test multiple products, the repeated executions provide a voting oracle. An oracle is a program that can determine in certain cases whether the execution of a program is valid or not.

ACKNOWLEDGMENT

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 Uwe Zdun for his helpful feedback during the shepherding process of EuroPLoP 2011. The auhtors would also like to thank those who attended the writers' workshop at EuroPLoP 2011 for their helpful feedback.

REFERENCES

BINDER, R. V. 1999. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BOSCH, J. AND LEE, J., Eds. 2010. *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Lecture Notes in Computer Science Series, vol. 6287. Springer.

CABRAL, I., COHEN, M. B., AND ROTHERMEL, G. 2010. Improving the testing and testability of software product lines. See Bosch and Lee [2010], 241–255.

COHEN, M. B., DWYER, M. B., AND SHI, J. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering 34*, 633–650.

ENGSTRÖM, E. AND RUNESON, P. 2011. Software product line testing - a systematic mapping study. *Information and Software Technology 53,* 1, 2 – 13.

JOHANSEN, M. F., HAUGEN, Ø., AND FLEUREY, F. 2011. A Survey of Empirics of Strategies for Software Product Line Testing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, L. O'Conner, Ed. IEEE Computer Society, Washington, DC, USA, 266–269.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute. November.

KUHN, D. R., WALLACE, D. R., AND GALLO, A. M. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering 30,* 6, 418–421.

POHL, K., BÖCKLE, G., AND LINDEN, F. J. V. D. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

REUYS, A., REIS, S., KAMSTIES, E., AND POHL, K. 2006. The scented method for testing software product lines. In *Software Product Lines*, T. Käkölä and J. C. Dueñas, Eds. Springer, Berlin, Heidelberg, 479–520.

RIVIERES, J. AND BEATON, W. 2006. Eclipse Platform Technical Overview.

STRICKER, V., METZGER, A., AND POHL, K. 2010. Avoiding redundant testing in application engineering. See Bosch and Lee [2010], 226–240.

UZUNCAOVA, E., KHURSHID, S., AND BATORY, D. 2010. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on 36,* 3, 309 –322.

Received Feb. 2011; revised Jan. 2012; accepted May 2012

Chapter 10

Paper 4: Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines

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

Martin Fagereng Johansen[1,2], Øystein Haugen[1], Franck Fleurey[1], Anne Grete Eldegard[3], and Torbjørn Syversen[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] 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.

R.B. France et al. (Eds.): MODELS 2012, LNCS 7590, pp. 269–284, 2012.

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[1] 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[2]. 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].

[1] The Eclipse IDE is provided by the Eclipse Foundation and is independent from the TOMRA case.

[2] `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[3]. The configurations of these products are shown in Table 1a[4]. 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

[3] `http://eclipse.org/downloads/` as of 2012-03-09.

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

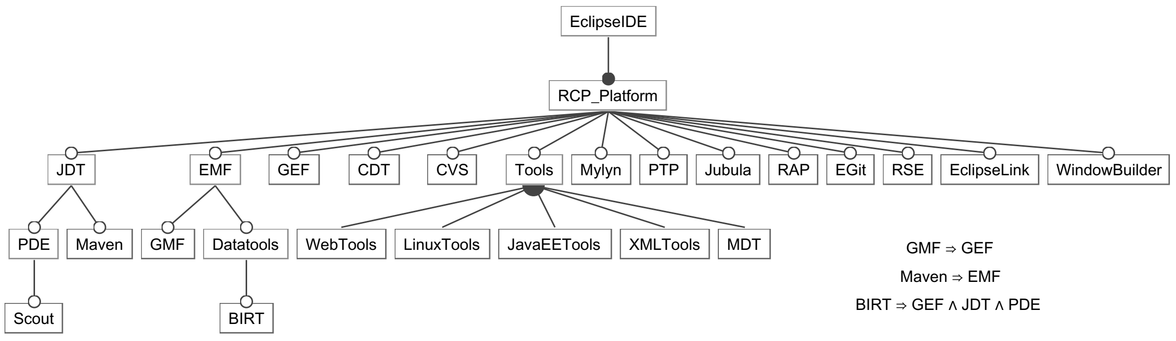


Fig. 1. Feature model for a significant part of the Eclipse IDE product line supported by the Eclipse Project

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	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	-
Tools	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	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	-	-
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	X	-	-	-
WebTools	X	-	X	X	-	-	X	-	X	-	-	-
LinuxTools	X	-	X	-	-	X	X	-	X	-	-	-
JavaEETools	X	-	-	X	-	X	-	-	X	-	-	-
XMLTools	-	X	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	-	-	X
CDT	X	-	X	-	X	X	-	-	-	-	-	-
BIRT	X	-	-	-	X	-	-	X	X	-	-	-
GMF	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.

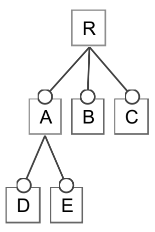
3 Weighted Combinatorial Interaction Testing

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



(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[5]. 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.

[5] 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 very 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.

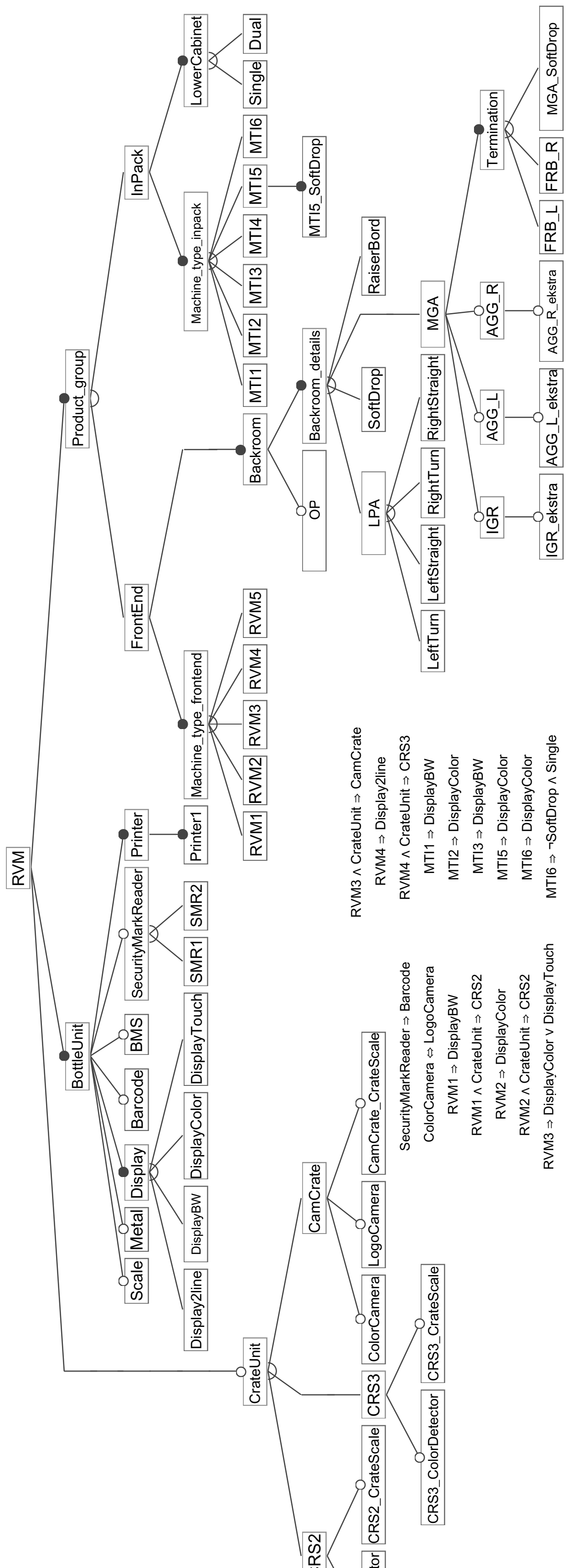


Fig. 2. Part of the Feature model of TOMRA's Product Line of Reverse Vending Machines

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	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	50	33	33	25	75	120	58	1	81	81	525	100	1500	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[6] took 49s, 141s and 1,090s for an improvement in 2-wise covering of 1, 2 and 3 suggestions of changes respectively.

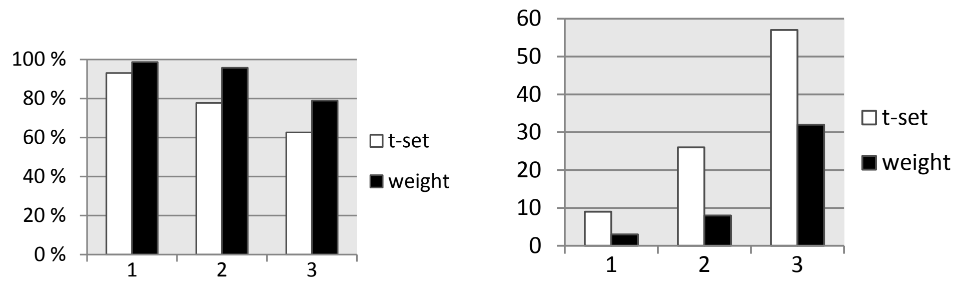
The next set of runs (4–6) is on an improved version of the original test lab. We chose to apply suggestion 1, to exclude feature *Metal* for product 11, and got a new test lab of 2-wise weight coverage of 96.0%. The improvements can be read in the same way as the previous set of suggestions.

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%.

[6] 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.



(a) t-set and Weight Coverage of Original Lab

(b) Size of Labs with 95% t-set and Weight Coverage

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[7]. These can be used as weights[8]. 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.

[7] `http://eclipse.org/downloads/` as of 2012-03-09.

[8] A better source of weights and sub-product lines is the data aquired 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/`.

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 that 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 (ISSRE), 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: Käköla, 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)

Chapter 11

Paper 5: A Technique for Agile and Automatic Interaction Testing for Product Lines

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].

B. Nielsen and C. Weise (Eds.): ICTSS 2012, LNCS 7641, pp. 39–54, 2012.

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.

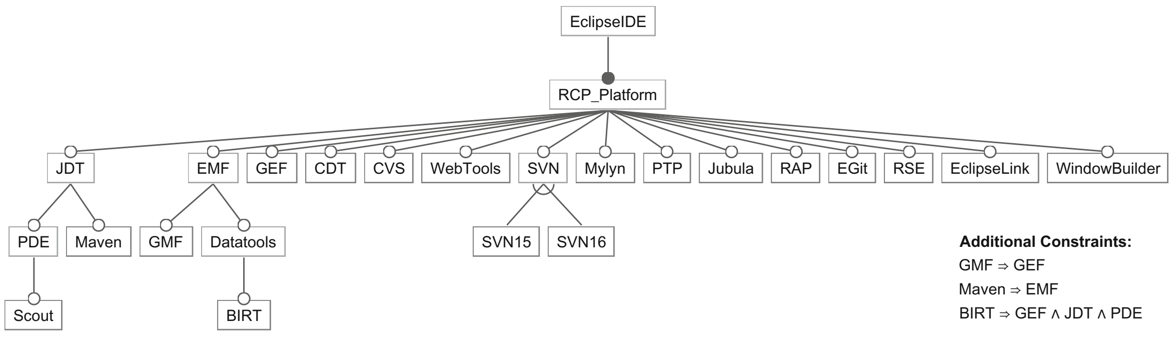


Fig. 1. Feature Model for the Eclipse IDE Product Line

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.

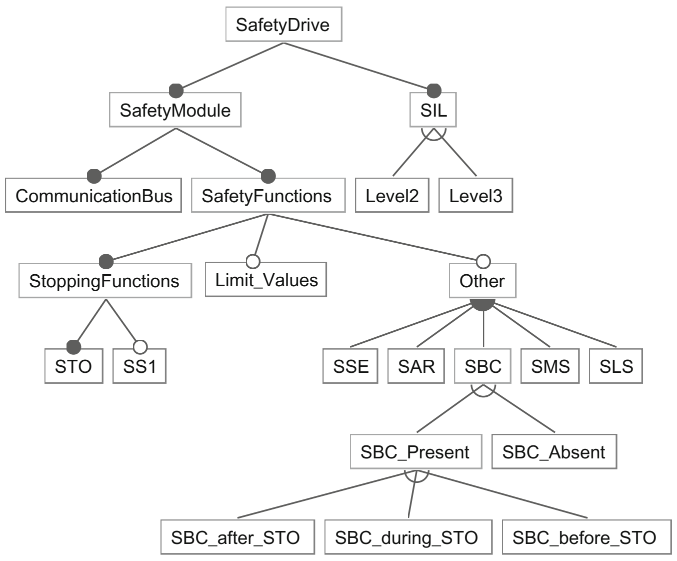
A simulated version of the ABB Safety Module was built—independently of the work in this paper—for experimenting with testing techniques. It is this version of the ABB Safety Module which testing is reported in this paper.

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

```
$CA \leftarrow FM.GenerateCoveringArray(t)$
for each configuration $c$ in $CA$ do
    $p \leftarrow GetBasicEclipsePlatform()$
    for each feature $f$ in $c$ do
        $p.installFeature(f)$
    end for
    for each feature $f$ in $c$ do
        $tests \leftarrow f.getAssociatedTests()$
        for each test $test$ in $tests$ do
            $p.installTest(test)$
            $result \leftarrow p.runTest(test)$
            $table.put(result, c, f)$
        end for
    end for
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)

Part III

Appendices

Appendix A

Resource Pages for Papers

[Resource Page for Paper 1]

Johansen et al. 2011 SPL Covering Array Tool discussed in "Properties of Realistic Feature Models make Combinatorial Testing of Product Lines Feasible"

Download the SPL Covering Array Tool v0.2 (MODELS 2011).

It includes the following software as dependencies, bundled with the tool:

- Feature IDE
- SPLAR - Software Product Lines Automated Reasoning library
- SAT4J
- JSON
- Apache Commons Math

Script for reproducing results:

Download a script for reproduction of the results presented in the paper. The models presented in the paper are required to run the script.

Usage:

java -jar SPLCATool-v0.2-MODELS2011.jar

```
SPL Covering Array Tool v0.2 (MODELS 2011)
http://heim.ifi.uio.no/martifag/models2011/spltool/
Usage: <task>
Tasks:
 -t count_solutions -fm <feature_model>
 -t sat_time -fm <feature_model>
 -t t_wise -fm <feature_model> -s <strength> (-BTR "Bow Tie Reduction")
 -t calc_cov -fm <feature_model> -s <strength> -ca <covering array>
 -t verify_solutions -fm <feature_model> -check <covering array>
 -t help (this menu)
Supported Feature models formats:
 - Feature IDE GUI DSL (.m)
 - Simple XML Feature Models (.xml)
 - Dimacs (.dimacs)
```

Examples:

```
> java -jar SPLCATool-v0.2-MODELS2011.jar -t count_solutions -fm TightVNC.m
SPL Covering Array Tool v0.2 (MODELS 2011)
http://heim.ifi.uio.no/martifag/models2011/spltool/
Loading GUI DSL: TightVNC.m
Successfully loaded and converted the model:
Features: 30
Constraints: 4
Counting solutions
Solutions: 297252.0

> java -jar SPLCATool-v0.2-MODELS2011.jar -t t_wise -fm Eshop-fm.xml -s 2
SPL Covering Array Tool v0.2 (MODELS 2011)
```

```
http://heim.ifi.uio.no/martifag/models2011/spltool/
Loading SXFM: Eshop-fm.xml
Successfully loaded and converted the model:
Features: 287
Constraints: 22
Generating 2-wise covering array
Running algorithm: Chvatal's algorithm adopted for Covering Array
generation
Uncovered pairs left: 164164
0/164164
...
Done. Size: 22, time: 364583 milliseconds
Wrote result to Eshop-fm.xml.ca2.csv

> java -jar SPLCATool-v0.2-MODELS2011.jar -t sat_time -fm 2.6.28.6-
icse11.dimacs
SPL Covering Array Tool v0.2 (MODELS 2011)
http://heim.ifi.uio.no/martifag/models2011/spltool/
Loading dimacs: 2.6.28.6-icse11.dimacs
CNF: Given p and c: 6888 and 343944
Successfully loaded and converted the model:
Features: 6888
Constraints: 187193
Satisfying the feature model
SAT done: 125457 microseconds, sat: true
```

Johansen et al. 2011 feature models discussed in "Properties of Realistic Feature Models make Combinatorial Testing of Product Lines Feasible"

Download the feature models discussed in the paper here.

Contains the following models from various sources

- 2.6.28.6-icse11.dimacs - "X86 Linux kernel 2.6.28.6", from "Feature Models in the Wild" [3]
- ecos-icse11.dimacs - "eCos 3.0 i386pc, from "Feature Models in the Wild" [3]
- freebsd-icse11.dimacs - FreeBSD kernel 8.0.0, from "Feature Models in the Wild" [3]
- Eshop-fm.xml - e-Shop, From "Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates" by Sean Quan Lau, 2006,
- Violet.m - "Violet, graphical model editor" [4]
- Berkeley.m - "Berkeley DB" [5]
- arcade_game_pl_fm.xml - "Arcade Game Maker Pedagogical Product Line" [6]
- Graph-product-line-fm.xml - From "A Standard Problem for Evaluating Product-Line Methodologies", by Lopez-Herrejon and Batory, 2001
- Gg4.m - "Graph Product Line Nr. 4", a more complex version of the Graph Product line from 2006
- smart_home_fm.xml - From "A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements" by Nathan Weston et. al, 2009
- TightVNC.m - "TightVNC Remote Desktop Software" [7]
- Apl.m - "AHEAD Tool Suite (ATS) Product Line", from Trujillo et al. 2006
- fame_dbms_fm.xml - "Fame DBMS" [8]

- connector_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Völter, 2009
- stack_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Völter, 2009
- REAL-FM-12.xml - From "Efficient Reasoning Techniques for Large Scale Feature Models" by Marcilio Mendonca, 2009
- movies_app_fm.xml - From "Context Awareness for Dynamic Service-Oriented Product Lines" by Carlos Parra et. al 2009
- aircraft_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Völter, 2009
- car_fm.xml - From "Automated Reasoning for Multi-step Feature Model Configuration Problems" by Jules White et. al, 2009

[3]: http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas
[4]: http://sourceforge.net/projects/violet/
[5]: http://www.oracle.com/us/products/database/berkeley-db/index.html
[6]: http://www.sei.cmu.edu/productlines/ppl/
[7]: http://www.tightvnc.com/
[8]: http://fame-dbms.org/

[Resource Page for Paper 2]

Resource Page for "An Algorithm for Generating t-wise Covering Arrays from Large Feature Models"

Welcome to the resource page for the SPLC 2012 publication titled "An Algorithm for Generating t-wise Covering Arrays from Large Feature Models" written by Martin Fagereng Johansen, Øystein Haugen and Franck Fleurey.

Contents

Tool - implementation of ICPL

The SPL Covering Array Tool v0.3 (SPLC 2012) contains an implementation of the ICPL algorithm for 1-3 wise covering array generation. The source code is available licensed under the Eclipse Public License - v 1.0.

Since CASA, NIST ACTS and MoSo-PoLiTe are only available through their providers, they are not distributed with this download. If you want to test the full version of the tool that includes CASA, NIST ACTS and MoSo-PoLiTe, contact the respective sources to obtain the tools, then contact us to get a version of the tool with the proper adaptors.

When running the tool, ICPL is called 'ICPL', and the algorithm discussed as 'Algorithm 1' in the paper is called 'Chvatal'. ICPL is of course set as the default algorithm. If you want to generate a covering array using ICPL run:

```
java -jar SPLCATool-v0.3-SPLC2012.jar -t t_wise -a ICPL -s <strength> -fm
<feature model>
```

The tool includes the following software as dependencies, bundled with the tool:

- Feature IDE
- SPLAR - Software Product Lines Automated Reasoning library
- SAT4J
- JSON
- Apache Commons Math
- google-gson
- JavaBDD
- jgraph
- JUnit
- opencsv
- guidsl
- Jakarta

Usage:

java -jar SPLCATool-v0.3-SPLC2012.jar

```
SPL Covering Array Tool v0.3 (SPLC 2012) (ICPL edition)
http://heim.ifi.uio.no/martifag/splc2012/
Args: {limit=100%, t=help, a=ICPL}
Usage: <task>
Tasks:
 -t count_solutions -fm <feature_model>
 -t sat_time -fm <feature_model>
 -t t_wise -a Chvatal -fm <feature_model> -s <strength, 1-4> (-startFrom
<covering array>) (-limit <coverage limit>) (-sizelimit <rows>) (-onlyOnes)
(-noAllZeros)
 -t t_wise -a ICPL -fm <feature_model> -s <strength, 1-3> (-startFrom
<covering array>) (-onlyOnes) (-noAllZeros) [Inexact: (-sizelimit <rows>)
(-limit <coverage limit>)]
 -t t_wise -a CASA -fm <feature_model> -s <strength, 1-6>
 -t calc_cov -fm <feature_model> -s <strength> -ca <covering array>
 -t verify_solutions -fm <feature_model> -check <covering array>
 -t help (this menu)
Supported Feature models formats:
 - Feature IDE GUI DSL (.m)
 - Simple XML Feature Models (.xml)
 - Dimacs (.dimacs)
 - CNF (.cnf)
```

Examples

```
> java -jar SPLCATool-v0.3-SPLC2012.jar -t t_wise -a ICPL -s 2 -fm Eshop-
fm.xml
SPL Covering Array Tool v0.3 (SPLC 2012) (ICPL edition)
http://heim.ifi.uio.no/martifag/splc2012/
Args: {limit=100%, fm=Eshop-fm.xml, t=t_wise, s=2, a=ICPL}
Loading SXFM: Eshop-fm.xml
Successfully loaded and converted the model:
Features: 287
Constraints: 22
Generating 2-wise covering array with algorithm: ICPL
Running algorithm: ICPL
Covering 100%
--- 1-wise ---
...
1-wise done, solutions: 3, invalid: 28
--- 2-wise ---
...
Uncovered: 0, progress: 100% with solutions: 21
2-wise done, solutions: 21, invalid: 15638
Done. Size: 21, time: 3765 milliseconds
Wrote result to Eshop-fm.xml.ca2.csv
```

Feature Model Corpus

These are the feature models discussed in the paper:

- 2.6.28.6-icse11.dimacs - "X86 Linux kernel 2.6.28.6", from "Feature Models in the Wild" [1]
- ecos-icse11.dimacs - "eCos 3.0 i386pc, from "Feature Models in the Wild" [1]
- freebsd-icse11.dimacs - FreeBSD kernel 8.0.0, from "Feature Models in the Wild" [1]
- Eshop-fm.xml - e-Shop, From "Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates" by Sean Quan Lau, 2006,
- Violet.m - "Violet, graphical model editor" [2]
- Berkeley.m - "Berkeley DB" [3]
- arcade_game_pl_fm.xml - "Arcade Game Maker Pedagogical Product Line" [4]
- Graph-product-line-fm.xml - From "A Standard Problem for Evaluating Product-Line Methodologies", by Lopez-Herrejon and Batory, 2001
- Gg4.m - "Graph Product Line Nr. 4", a more complex version of the Graph Product line from 2006
- smart_home_fm.xml - From "A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements" by Nathan Weston et. al, 2009
- TightVNC.m - "TightVNC Remote Desktop Software" [5]
- Apl.m - "AHEAD Tool Suite (ATS) Product Line", from Trujillo et al. 2006
- fame_dbms_fm.xml - "Fame DBMS" [6]
- connector_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Voelter, 2009
- stack_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Voelter, 2009

- REAL-FM-12.xml - From "Efficient Reasoning Techniques for Large Scale Feature Models" by Marcilio Mendonca, 2009
- movies_app_fm.xml - From "Context Awareness for Dynamic Service-Oriented Product Lines" by Carlos Parra et. al 2009
- aircraft_fm.xml - From "Using Domain-Specific Languages for Product Line Engineering" by Markus Voelter, 2009
- car_fm.xml - From "Automated Reasoning for Multi-step Feature Model Configuration Problems" by Jules White et. al, 2009

[1]: http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas
[2]: http://sourceforge.net/projects/violet/
[3]: http://www.oracle.com/us/products/database/berkeley-db/index.html
[4]: http://www.sei.cmu.edu/productlines/ppl/
[5]: http://www.tightvnc.com/
[6]: http://fame-dbms.org/

Measurements, Scripts and Adaptors

Adaptors

In order to run the test scripts for the three algorithms CASA, NIST ACTS and MoSo-PoLiTe, contact the respective providers to get the implementations. Then, contact us to get the tool that includes adaptors for the tools to run the test scripts for them.

Measurements

These files contain all the measurements done for each strength, algorithm and feature model. The measurements are stored as Office Open XML-files. The measurements for each strength are found in separate tabs.

- ICPL
- Alg. 1
- CASA
- IPOG
- MoSo-PoLiTe
- Comparison Table

Scripts

These scripts allow the reproduction of the measurements for each strength, algorithm and model.

- ICPL
- Alg. 1
- CASA
- IPOG
- MoSo-PoLiTe

[Resource Page for Paper 3]

Johansen et al. 2011 SPL Covering Array Tool discussed in "Bow Tie Testing - A Testing Pattern for Product Lines"

Download the SPL Covering Array Tool v0.1 (EuroPLoP 2011).

It includes the following software as dependencies, bundled with the tool:

- Feature IDE
- SPLAR - Software Product Lines Automated Reasoning library
- SAT4J
- JSON
- Apache Commons Math

Usage:

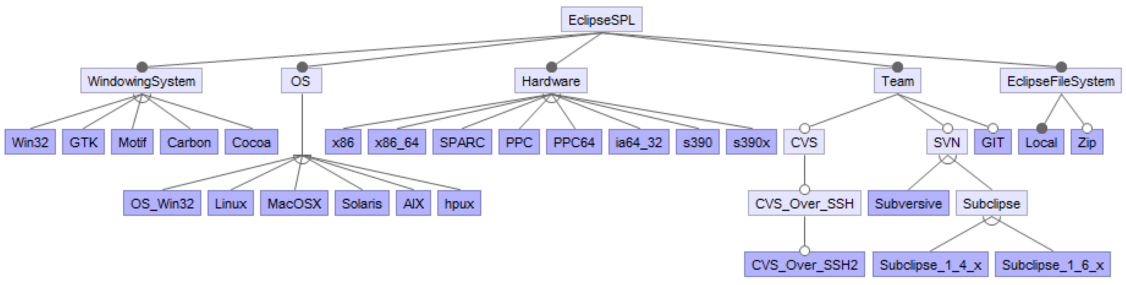
java -jar SPLCATool-v0.1-EuroPLoP2011.jar

```
SPL Covering Array Tool v0.1 (EuroPLoP 2011)
http://heim.ifi.uio.no/martifag/europlop2011/splcatool/
Args: {limit=100%, t=help, threads=1, a=Chvatal}
Usage: <task>
Tasks:
 -t count_solutions -fm <feature_model>
 -t t_wise -fm <feature_model> -s <strength, 1-3> (-startFrom <covering
array>) (-limit <coverage limit>) (-sizelimit <rows>) (-onlyOnes) (-
noAllZeros) (-BTR (Bow-Tie Reduction))
 -t help (this menu)
Supported Feature models formats:
 - Feature IDE GUI DSL (.m)
 - Simple XML Feature Models (.xml)
 - Dimacs (.dimacs)
 - CNF (.cnf)
```

Example Feature Model and Annotations

Feature Model

The running example from the paper is available:



Carbon ∧ MacOSX ∧ x86 ∨ Cocoa ∧ MacOSX ∧ (x86 ∨ x86_64) ∨ GTK ∧ Linux ∧ PPC ∨ GTK ∧ Linux ∧ PPC64 ∨
GTK ∧ Linux ∧ s390 ∨ GTK ∧ Linux ∧ s390x ∨ GTK ∧ Linux ∧ x86 ∨ GTK ∧ Linux ∧ x86 64 ∨ GTK ∧ Solaris ∧ SPARC ∨
GTK ∧ Solaris ∧ x86 ∨ Motif ∧ AIX ∧ PPC ∨ Motif ∧ hpux ∧ ia64_32 ∨ Motif ∧ Linux ∧ x86 ∨ Win32 ∧ OS_Win32 ∧ x86 ∨
Win32 ∧ OS_Win32 ∧ x86_64

Annotations

These annotations were discussed in the paper.

```
AL WindowingSystem
AL Hardware
AL OS
```

Example run

The following run will produce the result discussed in the paper. The 2-wise covering array is written to EclipseSPLRef.m.ca2.csv and the reduced version is written to EclipseSPLRef.m.ca2.csv.btr.csv.

```
> java -jar SPLCATool-v0.1-EuroPLoP2011.jar -t t_wise -fm EclipseSPLRef.m -
s 2 -BTR
http://heim.ifi.uio.no/martifag/europlop2011/splcatool/
Args: {limit=100%, BTR=, fm=EclipseSPLRef.m, t=t_wise, s=2, threads=1,
a=Chvatal}
Loading GUI DSL: EclipseSPLRef.m
Successfully loaded and converted the model:
Features: 37
Constraints: 610
Generating 2-wise covering array with algorithm: Chvatal
Running algorithm: Chvatal's algorithm adopted for Covering Array
generation
Covering 100%
Uncovered pairs left: 2664
0/2664
...
Done. Size: 41, time: 1045 milliseconds
BTR. Size: 20
Wrote result to EclipseSPLRef.m.ca2.csv
Wrote result to EclipseSPLRef.m.ca2.csv.btr.csv
```

[Resource Page for Paper 4]

Resource Page for "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines"

Welcome to the resource page for the MODELS 2012 publication titled "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines" written by Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen.

Contents

Tool

Download the SPL Covering Array Tool v0.4 (MODELS 2012). The source is available freely. The source is available upon request.

It includes the following software as dependencies, bundled with the tool:

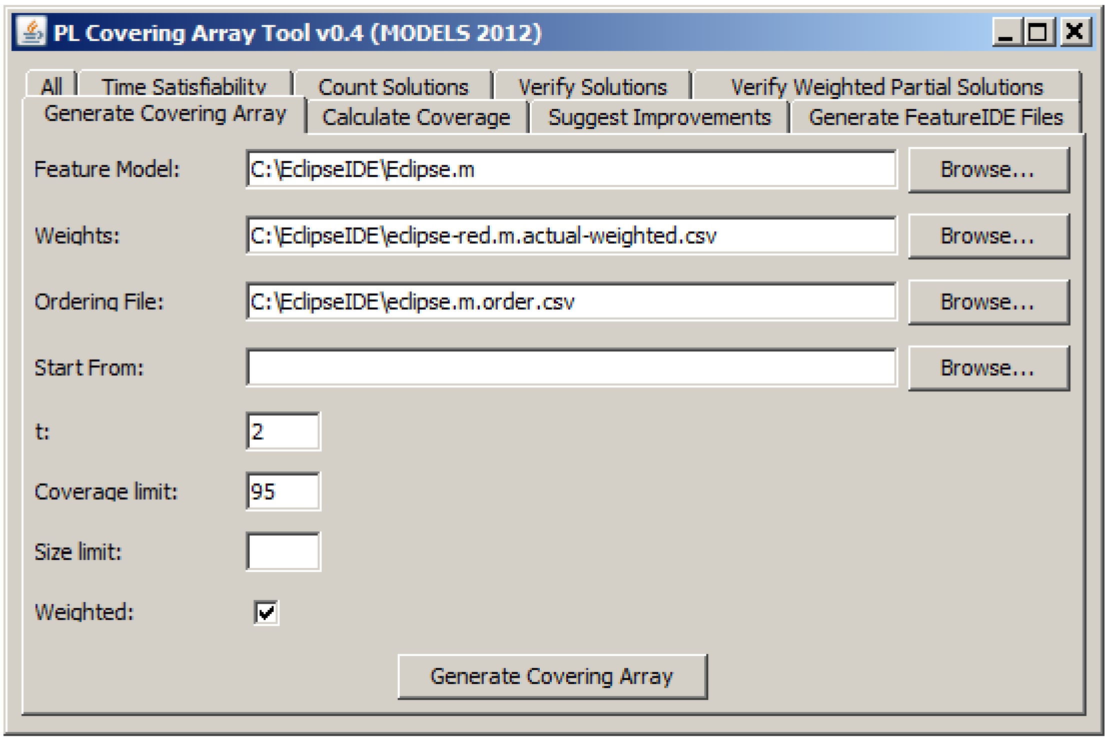
- Feature IDE
- SPLAR - Software Product Lines Automated Reasoning library
- SAT4J
- JSON
- Apache Commons Math
- google-gson
- JavaBDD
- jgraph
- JUnit
- opencsv
- guidsl
- Jakarta

Examples:

The tool has a GUI interface. The interface consists of a number of tabs, each including the fields for the required and optional input for each functionality. Three of the most relevant features are listed here as examples. Note: Output is written to standard output. Start the GUI in a command shell using java (and not javaw). Alternatively, redirect output to a log file.

Generate a new Covering Array:

The following example shows the tab for generating covering arrays. When the "weighted" checkbox is checked, the generation is based on weights instead of tuples. The feature model is entered first, followed by the weights file and the optionally the ordering file to give the result a specific feature order. If you want to extend an existing covering array, enter the basis in the "start from"-field. "t" specifies the strength. "Coverage limit" specifies the coverage to be achieved. A blank entry defaults to 100%. "Size limit" is the maximum number of products to generate. Press "Generate Covering Array"



PL Covering Array Tool v0.4 (MODELS 2012)

All | Time Satisfiability | Count Solutions | Verify Solutions | Verify Weighted Partial Solutions | Generate Covering Array | Calculate Coverage | Suggest Improvements | Generate FeatureIDE Files

Feature Model: C:\EclipseIDE\Eclipse.m

Weights: C:\EclipseIDE\eclipse-red.m.actual-weighted.csv

Ordering File: C:\EclipseIDE\eclipse.m.order.csv

Start From:

t: 2

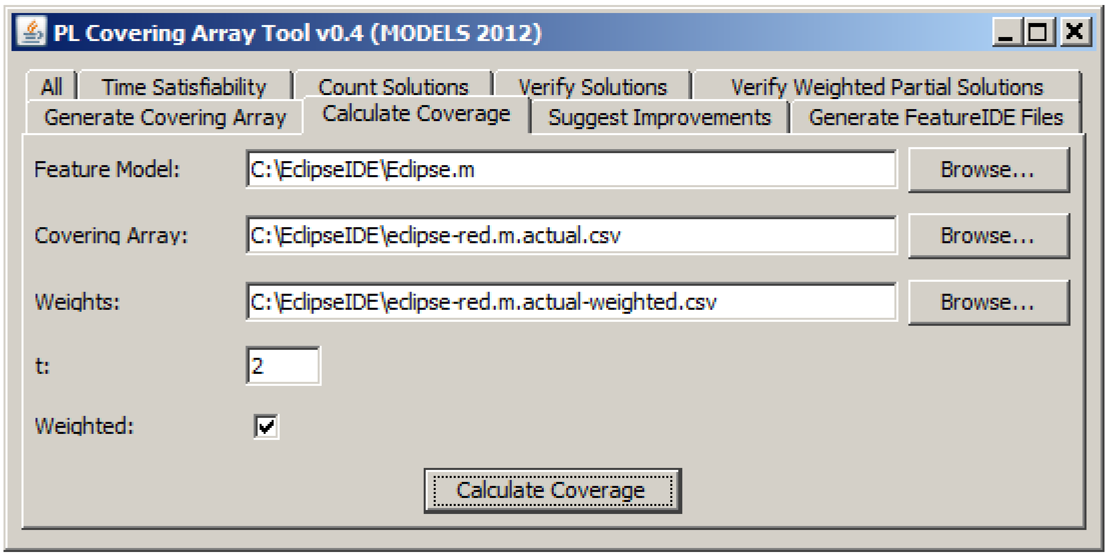
Coverage limit: 95

Size limit:

Weighted: ✓

Calculate Coverage of an existing set of products:

The following example generates the 3-wise coverage of a set of products. When the "weighted" checkbox is checked, the weighted coverage is calculated, else the tuple coverage is calculated.



PL Covering Array Tool v0.4 (MODELS 2012)

All | Time Satisfiability | Count Solutions | Verify Solutions | Verify Weighted Partial Solutions | Generate Covering Array | Calculate Coverage | Suggest Improvements | Generate FeatureIDE Files

Feature Model: C:\EclipseIDE\Eclipse.m

Covering Array: C:\EclipseIDE\eclipse-red.m.actual.csv

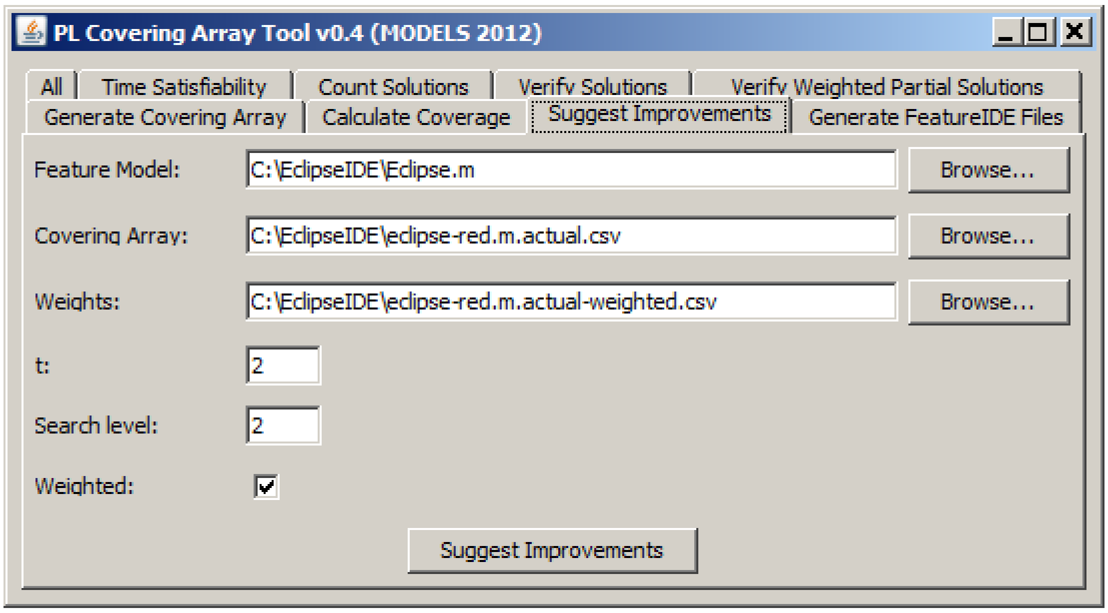
Weights: C:\EclipseIDE\eclipse-red.m.actual-weighted.csv

t: 2

Weighted: ✓

Suggest improvements to a set of products:

The following example generates either single or double changes to the set of products to increase the 3-wise weight coverage of the set of products. "search level" is the maximum number of changes to search for. Only level 1-3 is supported by the tool in the current version.


PL Covering Array Tool v0.4 (MODELS 2012)

Feature Model: C:\EclipseIDE\Eclipse.m

Covering Array: C:\EclipseIDE\eclipse-red.m.actual.csv

Weights: C:\EclipseIDE\eclipse-red.m.actual-weighted.csv

t: 2

Search level: 2

Weighted: ✓

Models

ABC example

- Feature model
- Weights
- Ordering

TOMRA product line case

The complete models from TOMRA discussed in the paper are unfortunately not available due to sensitivity concerns.

Eclipse IDE product line case

- Feature model
- Products
- Weights
- Ordering

[Resource Page for Paper 5]

Resource page for "A Technique for Agile and Automatic Interaction Testing for Product Lines"

On this website you will find information on how to reproduce the results from the ICTSS 2012 paper titled "A Technique for Agile and Automatic Interaction Testing for Product Lines" written by Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien.

Applying the Technique for Testing of Eclipse-based Product Lines

1. Setting up the Directory Structure

Decide on a root directory in which all files related to the testing will be stored. In this directory, create these directories:

- "models", contains all input to the testing tools.
- "packages", contains all packages to be used to construct the software to be tested.
- "products", will store all built and tested products and their associated workspaces.
- "results", will store all generated results.
- "scripts", contains all scripts used for testing.
- "tools", contains all tools used for testing.
- "workspace", workspace for the Eclipse tool used to build the package repository.

2. Acquiring the Required Tools

Download and extract the following free tools in the "tools" directory.

1. 7-Zip Command Line Version (9.20)
2. Eclipse Platform (3.7.0)
3. UnxUtils (of 2007-03-01), Port of the most important GNU utilities to Windows
4. An implementation of Algorithm 3 from the paper (Contact us to get the source code.)

3. Building the Package Repository

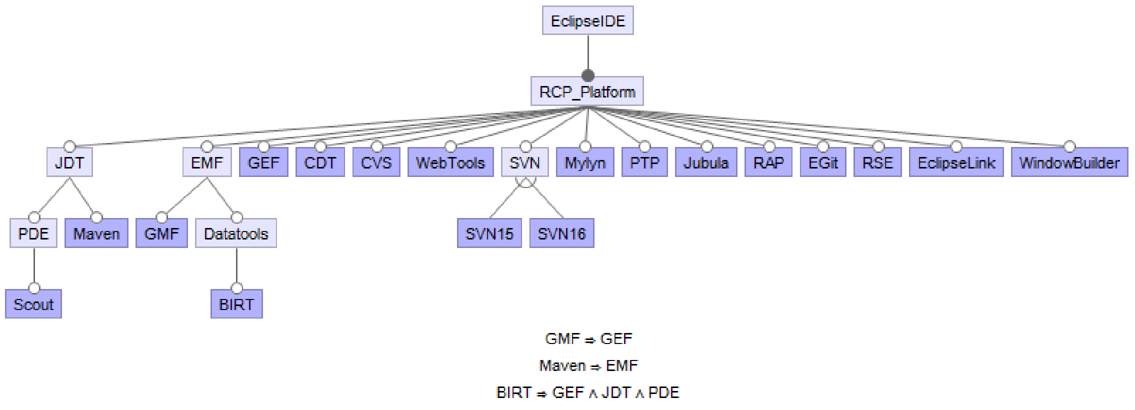
1. Download the mirroring-script and place it in the scripts directory. Download and edit the environment set up script in which the path to the java.exe, cmd.exe and the root directory must be specified. Executing the mirroring script will construct mirrors for the Eclipse Indigo repositories in "%basedir%/packages/repos/". This will take a while, and might require rerunning the script several times to ensure that all packages were downloaded correctly. The size of these repositories during the experiments presented in the paper was 3.6 GiB.
2. Place a copy of the package containing the Eclipse Platform in "packages".

3. Download the Eclipse Automated Tests for Eclipse 3.7.0 and extract them in the "packages" directory.

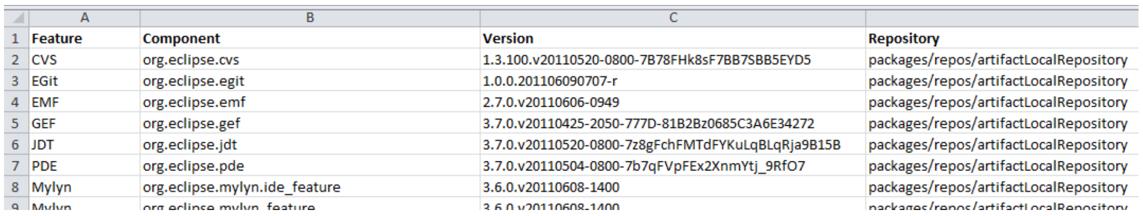
4. Setting up the Required Models

The technique requires three files to be built.

1. The feature model, for example such as the one for the Eclipse IDEs.

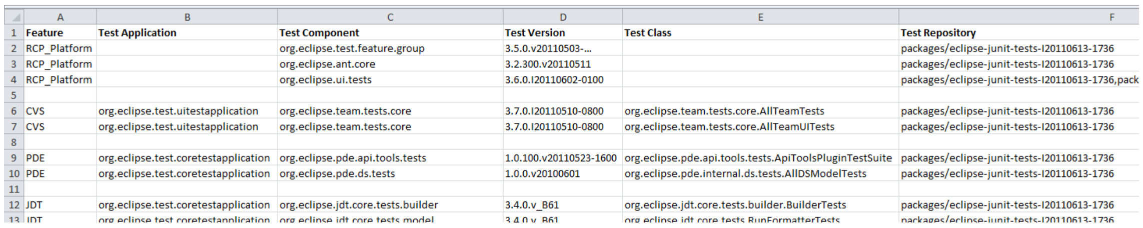


2. The mapping between features are code artifacts, as for example the one used in the experiment in the paper.



	A	B	C	
1	**Feature**	**Component**	**Version**	**Repository**
2	CVS	org.eclipse.cvs	1.3.100.v20110520-0800-7B78FHk8sF7BB7SBB5EYD5	packages/repos/artifactLocalRepository
3	EGit	org.eclipse.egit	1.0.0.201106090707-r	packages/repos/artifactLocalRepository
4	EMF	org.eclipse.emf	2.7.0.v20110606-0949	packages/repos/artifactLocalRepository
5	GEF	org.eclipse.gef	3.7.0.v20110425-2050-777D-81B2Bz0685C3A6E34272	packages/repos/artifactLocalRepository
6	JDT	org.eclipse.jdt	3.7.0.v20110520-0800-7z8gFchFMTdFYKuLqBLqRja9B15B	packages/repos/artifactLocalRepository
7	PDE	org.eclipse.pde	3.7.0.v20110504-0800-7b7qFVpFEx2XnmYtj_9RfO7	packages/repos/artifactLocalRepository
8	Mylyn	org.eclipse.mylyn.ide_feature	3.6.0.v20110608-1400	packages/repos/artifactLocalRepository
9	Mylyn	org.eclipse.mylyn_feature	3.6.0.v20110608-1400	packages/repos/artifactLocalRepository

3. The mapping between features and tests, as for example the one used in the experiment in the paper.



	A	B	C	D	E	F
1	**Feature**	**Test Application**	**Test Component**	**Test Version**	**Test Class**	**Test Repository**
2	RCP_Platform		org.eclipse.test.feature.group	3.5.0.v20110503-...		packages/eclipse-junit-tests-I20110613-1736
3	RCP_Platform		org.eclipse.ant.core	3.2.300.v20110511		packages/eclipse-junit-tests-I20110613-1736
4	RCP_Platform		org.eclipse.ui.tests	3.6.0.I20110602-0100		packages/eclipse-junit-tests-I20110613-1736,pack
5						
6	CVS	org.eclipse.test.uitestapplication	org.eclipse.team.tests.core	3.7.0.I20110510-0800	org.eclipse.team.tests.core.AllTeamTests	packages/eclipse-junit-tests-I20110613-1736
7	CVS	org.eclipse.test.uitestapplication	org.eclipse.team.tests.core	3.7.0.I20110510-0800	org.eclipse.team.tests.core.AllTeamUITests	packages/eclipse-junit-tests-I20110613-1736
8						
9	PDE	org.eclipse.test.coretestapplication	org.eclipse.pde.api.tools.tests	1.0.100.v20110523-1600	org.eclipse.pde.api.tools.tests.ApiToolsPluginTestSuite	packages/eclipse-junit-tests-I20110613-1736
10	PDE	org.eclipse.test.coretestapplication	org.eclipse.pde.ds.tests	1.0.0.v20100601	org.eclipse.pde.internal.ds.tests.AllDSModelTests	packages/eclipse-junit-tests-I20110613-1736
11						
12	JDT	org.eclipse.test.coretestapplication	org.eclipse.jdt.core.tests.builder	3.4.0.v_B61	org.eclipse.jdt.core.tests.builder.BuilderTests	packages/eclipse-junit-tests-I20110613-1736
13	JDT	org.eclipse.test.coretestapplication	org.eclipse.jdt.core.tests.model	3.4.0.v_B61	org.eclipse.jdt.core.tests.RunFormatterTests	packages/eclipse-junit-tests-I20110613-1736

5. Executing the Technique Described in the Paper

The following command will perform the technique described in the paper fully automatically. When the feature model, the features themselves or the test suites are changed, there is no additional required manual work to redo the testing other than executing this command. basedir is the root directory of the testing system, t is the strenght of the combinatorial interaction testing and timeout is the time in seconds after which a test suite execution will be aborted.

```
java no.sintef.ict.splcatool.lasplt.LTSPLT basedir t timeout
```

Output such as the following will be produced on the command line:

```
Converting Package Mapping File to CSV [done]
Converting Feature-Test Mapping File to CSV [done]
Loading Mapping Files [done]
Generating 2-wise Covering Array [done]
Loading Covering Array [done] Products: 14
Test Product 0:
  Installing base product: [done]
  Installing features:
    Installing feature EclipseIDE [nothing to install]
    Installing feature RCP_Platform [nothing to install]
  Installing tests:
    Installing tests for EclipseIDE
    Installing tests for RCP_Platform
      Installing test org.eclipse.test.feature.group [done]
      ...
      Installing test org.eclipse.search.tests [done]
  Running tests:
    Running tests for EclipseIDE
    Running tests for RCP_Platform
    Running test org.eclipse.ant.tests.core.AutomatedSuite [done] 100%
(85/85)
      Running test org.eclipse.compare.tests.AllTests [done] 89% (101/113)
      Running test org.eclipse.core.internal.expressions.tests.AllTests
[done] 100% (108/108)
      ...
      Running test org.eclipse.search.tests.AllSearchTests [already exists]
78% (29/37)
Test Product 1:
  Installing base product: [already exists]
  Installing features:
    Installing feature WindowBuilder [done]
    Installing feature SVN15 [done]
    Installing feature EclipseIDE [nothing to install]
    ...
    Installing feature RCP_Platform [nothing to install]
    Installing feature Datatools [done]
    Installing feature SVN [nothing to install]
...
Test Product 13:
...
```

The result produced by the experiment reported in the paper are available.

6. Generate the web-view for the results

The command in the previous step will produce logs containing the contents of stdour and stderr during test execution called `failure-<product nr>-<test class name>.log` and test result details in files called `product<product nr>-<test class name>-test-result.xml`. The result produced by the experiment reported in the paper are available.

These results can be compiled into a report by executing the following command.

```
java no.sintef.ict.splcatool.lasplt.GenerateWebView basedir
```

This will produce a HTML-document, such as the one for the Eclipse IDE experiment in the paper.

Applying the Technique for Testing of CVL-based Product Lines

The set up required to run the CVL-based version of the technique is found in CVLLASPLTInput.java (Template found in CVL toolchain).

- A CVL model modelling the variability of the product line with bindings to the model artifacts. (Safety Module Case)
- Class path for test execution
- Feature-Test Mapping (Safety Module Case)

It also requires the following tools:

- SPLCATool v0.3 (SPLC 2012)
- Java runtime 1.6
- Eclipse 3.7.0 Eclipse Modeling Tools
- CVL Headless build (Plugin in CVL toolchain)
- JavaFrame Headless build (Plugin in CVL toolchain)

Run the testing technique fully automatically by calling: `java -jar cvlsplt.jar`. The CVL-tool chain is available. Contact us for the source code.

When run, it will produce output as follows:

```
Extracting Feature Model from CVL Model: Done
Generating 2-wise Covering Array: Done
Injecting Covering Array into CVL model: Done
Executing CVL:
  Generated %basepth%/model/product0.uml
  Generated %basepth%/model/product1.uml
  Generated %basepth%/model/product2.uml
  ...
  Generated %basepth%/model/product10.uml
  Generated %basepth%/model/product11.uml
%basepth%/model/product0.uml
  Moving to %basepth%/JFWS/Product0/product.uml
  Running JavaFrame Generator: Exit code: 0
  Building workspace: Exit code: 0
  Building workspace: Exit code: 0
  Run Tests:
    SafetyDrive.GeneralStartUp: [success]
    SafetyDrive.Level3StartUpTest: [success]
Done
%basepth%/model/product1.uml
...
```

Appendix B

Algorithms Based on Weighted Sub-Product Line Models

In this appendix, we include details for algorithms related to weighted sub-product line models. These were not included in Paper 4 because of space constraints. A free and open-source implementation of them was released with SPLCATool v0.4 as resource material for the paper. This appendix provides pseudo-code for the algorithms with some explanation.

B.1 Some New Terms

A weighted t-set is a 2-tuple (w, e) with w being the weight and e being a t-set. The set of all weighted t-sets is called S_t.

A priority queue of weighted t-sets is called P_t. P_t is an object with two methods: $add((w, e))$, a method that adds a 2-tuple to the priority queue, and $pop() : (w, e)$ a method that gets the 2-tuple at the start of the queue and removes it from the queue.

A weighted sub-product line model, W, is a set of 2-tuples, a set of (w, C), where C is a valid, partial configuration (or sub-product line model) of feature model FM and w is a weight given to that sub-product line model.

B.2 Calculating Weights

Algorithm 4 takes a feature model, FM, a set of weighted sub-product line models, W, and a strength t. It makes a priority queue with weighted t-sets.

Line 1 generates the set of all t-sets from the feature model. Line 2 creates an empty priority queue. The loop at line 3 loops through each weighted sub-product line model. This will be our basis when getting t-sets and their weights. If a t-set is not in W, it will not be included in the sets to cover. This is essentially the same as getting a weight of 0. The loop at line 4 goes through each t-set contained in the sub-product line model, detailed as Algorithm 5. The loop at line 5 goes through each possible t-set. At line 6, we check whether the t-set is valid.

Lines 8–19 check whether the partial t-set gotten from W contains s. e contains s if each assignment in e is in s, lines 9–12, or if the feature with a wild-card from e is in s, lines 13–18. At line 14, x, the number of wild-cards, is counted.

Algorithm 4 Get Priority Queue of Valid, Weighted t-sets: $getWPriorityQueue(FM, W, t) : P_t$

```
1: $T_t \leftarrow FM.genT(t)$
2: $P_t \leftarrow PriorityQueue(\emptyset)$
3: for each pair $(w, C)$ in $W$ do
4:     for each t-set $e$ in $getTSets(C, t)$ do
5:         for each t-set $s$ in $T_t$ do
6:             if $FM.isSatisfiable(s)$ then
7:                 $(isIn, x) \leftarrow (true, 0)$
8:                 for each assignment $a$ in $e$ do
9:                     if $a.a \in \{true, false\}$ then
10:                        if $a \notin s$ then
11:                            $isIn \leftarrow false$
12:                        end if
13:                    else
14:                        $x \leftarrow x + 1$
15:                        if $a.f \notin s.getFs()$ then
16:                            $isIn \leftarrow false$
17:                        end if
18:                    end if
19:                end for
20:                if $isIn$ then
21:                    $P_t.add((w/2^x, s))$
22:                end if
23:            end if
24:        end for
25:    end for
26: end for
```

If s is contained within e, then s is given the weight of the sub-product line divided by 2^x, lines 20–22; 2^x is the number of t-sets contained in e.

Algorithm 5 Get t-sets in a Configuration: $getTSets(C, t) : T_t$

```
1: $T_t \leftarrow \emptyset$
2: if $t > 1$ then
3:     $T_{t-1} \leftarrow getTSets(C, t-1)$
4:     for each t-set $e$ in $T_{t-1}$ do
5:         for each assignment $a \in C$ do
6:             $T_t \leftarrow T_t \cup (\{a\} \cup e)$
7:         end for
8:     end for
9: else
10:    for each assignment $a$ in $C$ do
11:        $T_t \leftarrow T_t \cup \{a\}$
12:    end for
13: end if
```

A sub-algorithm of Algorithm 4 is Algorithm 5. The purpose of this algorithm is to generate all t-sets, T_t, covered by a configuration, C, of strength t. The algorithm starts by initializing a new, empty set, T_t, line 1. If the value of t is 1, all the assignments in C are added to T_t, lines 10–12. If t is larger than 1, T_{t-1} is generated, line 3. Then they are combined with every possible assignment, lines 5–7. This recursive algorithm eventually generates all t-sets covered by the configuration.

B.3 Generating a Weighted Covering Array

Ordinary covering arrays are generated by selecting the product that covers the most uncovered t-sets first. A weighted covering array is generated by selecting the product that covers the most weight first. Algorithm 6 is a modification of Algorithm 1 from Paper 2 for weighted covering arrays.

Algorithm 6 takes a feature model, FM, a weighted sub-product line model W, a strength t, and a percentage u. By selecting t-sets to cover from a priority queue of t-sets, prioritized according to weight, the algorithm keeps generating products until it has covered a u-part, at which point it stops.

Algorithm 6 Weighted Covering Array Generation: Adaption of Algorithm 1 from the SPLC 2012-paper
$GenWeightedCoveringArray(FM, W, t, u) : C_t$

```
 1: $T_t \leftarrow FM.genT(t)$
 2: $P_t \leftarrow getWPriorityQueue(FM, T_t, W, t)$
 3: $tw \leftarrow 0$
 4: for each t-set $e$ in $P_t$ do
 5:     $tw = tw + e.w$
 6: end for
 7: $(C_t, cw) \leftarrow (\emptyset, 0)$
 8: while $true$ do
 9:     $(C, CO) \leftarrow (\emptyset, \emptyset)$
10:     for each t-set $e$ in $P_t$ do
11:         if $FM.is\_satisfiable(C \cup e.e)$ then
12:             $C \leftarrow C \cup e.e$
13:             $CO \leftarrow CO \cup \{e.e\}$
14:             $cw \leftarrow cw + e.w$
15:         end if
16:     end for
17:     $P_t \leftarrow P_t \setminus CO$
18:     $C \leftarrow FM.satisfy(C)$
19:     $C_t \leftarrow C_t \cup \{C\}$
20:     if $cw/tw \geq u$ then
21:         break
22:     end if
23: end while
```

Line 1–2 constructs the priority queue of valid t-sets, as described in Algorithm 4. Line 3–6 calculates the total weight of all weighted t-sets. It does this by iterating through the entire priority queue and adding each weight into a total. At line 7, an empty covering array and an integer of the weight covered is initialized. Obviously, having no products, the covered weight is 0. Next, the main loop spans from line 8–23. It keeps adding products until the weight covered over the total weight is larger than or equal u. At this point the algorithm stops, lines 20-22. At lines 10–16, t-sets are covered, beginning at the top of the priority queue and proceeding downwards. When a t-set is covered, its weight is added to cw, the covered weight. At line 17, the covered t-sets are removed from the priority queue. At lines 18–19 the new product is added to the covering array, before deciding whether to stop at lines 20–22.

B.4 Generating Improvement Suggestions

Algorithm 7 describes how to generate suggestions for improvement of an existing covering array. The algorithm takes a feature model, FM, a covering array, CA, a weighted sub-product line model, W, a strength, t, and a number, n, of how many changes of a single product to consider.

Algorithm 7 Generate Suggestions
$genImpSug(FM, CA, W, t, n) : S$

```
1: $S \leftarrow \emptyset$
2: **if** $n > 0$ **then**
3:     $orgcov = getWeightCoverage(FM, CA, W, t)$
4:     $S \leftarrow S \cup genImpSug(FM, CA, W, t, n-1)$
5:     **for** each product $C$ in $CA$ **do**
6:         **for** each t-set $e$ in $getTSets(C, n)$ **do**
7:             **for** each assignment $a$ in $e$ **do**
8:                 $a.a \leftarrow \neg a.a$
9:             **end for**
10:            **if** $FM.isSatisfiable(C)$ **then**
11:                $newcov = getWeightCoverage(FM, CA, W, t)$
12:                **if** $newcov > orgcov$ **then**
13:                    $S.add(newcov, e.copy())$
14:                **end if**
15:            **end if**
16:            **for** each assignment $a$ in $e$ **do**
17:                $a.a \leftarrow \neg a.a$
18:            **end for**
19:        **end for**
20:    **end for**
21: **end if**
```

Line 1 initializes a new set S that will contain the suggestions and their new coverage. If the number of changes wanted is higher than 0, line 2, line 3 calculates the current weight coverage of the covering array given. This algorithm is detailed as Algorithm 8. Then, at line 4, all changes of size one less than n is calculated. The algorithm then, at lines 5 and 6, iterates through all products and all *n-sets* covered by them. (Note, that since we are searching for n changes, we want the n-sets and not the t-sets.) All assignments in this n-set are reversed, lines 7–9, and, if the new configuration is valid (line 10), the new coverage is calculated (line 11). If the new coverage is strictly larger than the original coverage, the suggestion is added to the set of suggestions, S. The suggestion is copied into the set. The assignments are then restored, lines 16–18.

Note that this algorithm is data-parallel. The set gotten at line 7 is huge, easily containing millions of elements; the rest of the algorithm can be run in parallel on as many nodes, at most.

Algorithm 8 is called two places in Algorithm 7. The purpose of this algorithm is to calculate the percentage of the total weight is covered, cov, by a covering array CA of feature model FM of strength t, given the weights and the sub-product lines model W.

The algorithm proceeds by first getting the set of all weighted t-sets, line 1. Line 3–5 calculates the total weight. Lines 6–10 collects all t-sets covered by the covering array into a set T_t. Note that since T_t is a set, duplicates do not occur. Then, the weight of all covered t-sets are summed up, lines 11-17, and the percentage calculated and returned, line 18.

Algorithm 8 Generate Weight Coverage
$getWeightCoverage(FM, CA, W, t) : cov$

```
1: $P_T \leftarrow getWPriorityQueue(FM, FM.genT(t), W, t)$
2: $(tw, cw, cov, T_t) \leftarrow (0, 0, 0, \emptyset)$
3: **for** each pair $e$ in $P_t$ **do**
4:     $tw = tw + e.w$
5: **end for**
6: **for** each product $C$ in $CA$ **do**
7:     **for** each t-set $e$ in $getTSets(C, t)$ **do**
8:         $T_t \leftarrow T_t \cup \{e\}$
9:     **end for**
10: **end for**
11: **for** each t-set $e$ in $T_t$ **do**
12:     **for** each pair $p$ in $P_t$ **do**
13:         **if** $e = p.e$ **then**
14:             $cw = p.w$
15:         **end if**
16:     **end for**
17: **end for**
18: $cov \leftarrow cw/tw$
```

Appendix C

Details for Comparisons

Table C.1, C.2, C.3, C.4 and C.5 shows the details of the average running time of ICPL, CASA, MoSo-PoLiTe, IPOG algorithms and the basic algorithm, respectively.

Figure C.1, C.2 and C.3 show the time performance of the five algorithms with an estimated line on a logarithmic scale. Notice how ICPL's line is characteristically different from all the other lines.

Table C.1: Performance of ICPL on Large Feature Models
*Average based on less than 100 measurements.
†Covering arrays for t-sets containing with only included features (1/8th of the t-sets), the memory usage was limited to 128 GiB instead of 32 GiB, and the algorithm was allowed to run in 64 processors at 100% activity instead of 6.

Feature Model\keys	Features	Constraints (CNF clauses)	SAT time (ms)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	125	25	89	*480	*33,702	–	–
freebsd-icse11.dimacs	1,396	17,352	18	9	10	77	240	†78	†2,540
ecos-icse11.dimacs	1,244	2,768	12	6	2	63	185	†64	†973
Eshop-fm.xml	287	22	5	3	<1	21	5	108	457
Violet.m	101	90	1	2	<1	26	2	122	32
Berkeley.m	78	47	1	4	<1	23	1	99	10
arcade_game_pl_fm.xml	61	35	3	3	<1	18	1	64	5
Gg4.m	38	23	1	6	<1	20	1	61	2
...									

Table C.2: Performance of CASA on Large Feature Models
*Average based on less than 100 measurements.

Feature Model\keys	Features	Constraints (CNF clauses)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	–	–	–	–	–	–
freebsd-icse11.dimacs	1,396	17,352	–	–	–	–	–	–
ecos-icse11.dimacs	1,244	2,768	*11	*51,705	–	–	–	–
Eshop-fm.xml	287	22	5	1,274	*31	*10,040	–	–
Violet.m	101	90	3	38	23	79	*93	*16,310
Berkeley.m	78	47	3	39	19	145	*77	*9,142
arcade_game_pl_fm.xml	61	35	3	22	13	32	46	1,999
Gg4.m	38	23	6	8	17	14	50	134
...								

Table C.3: Performance of MoSo-PoLiTe on Large Feature Models
*Average based on less than 100 measurements.

Feature Model\keys	Features	Constraints (CNF clauses)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	–	–	–	–	–	–
freebsd-icse11.dimacs	1,396	17,352	–	–	–	–	–	–
ecos-icse11.dimacs	1,244	2,768	–	–	–	–	–	–
Eshop-fm.xml	287	22	–	–	–	–	–	–
Violet.m	101	90	–	–	90	<1	1,123	164
Berkeley.m	78	47	–	–	42	<1	421	23
arcade_game_pl_fm.xml	61	35	–	–	38	<1	271	4
Gg4.m	38	23	–	–	–	–	–	–
...								

Table C.4: Performance of IPOG on Large Feature Models
*Average based on less than 100 measurements.

Feature Model\keys	Features	Constraints (CNF clauses)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	–	–	–	–	–	–
freebsd-icse11.dimacs	1,396	17,352	–	–	–	–	–	–
ecos-icse11.dimacs	1,244	2,768	–	–	–	–	–	–
Eshop-fm.xml	287	22	–	–	–	–	–	–
Violet.m	101	90	–	–	*27	*12,655	–	–
Berkeley.m	78	47	–	–	*22	*3,087	–	–
arcade_game_pl_fm.xml	61	35	–	–	*19	*5,858	*64	*45,632
Gg4.m	38	23	–	–	–	–	77	2,442
...								

Table C.5: Performance of Alg. 1 on Large Feature Models
*Average based on less than 100 measurements.

Feature Model\keys	Features	Constraints (CNF clauses)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	–	–	–	–	–	–
freebsd-icse11.dimacs	1,396	17,352	7	165	–	–	–	–
ecos-icse11.dimacs	1,244	2,768	6	13	–	–	–	–
Eshop-fm.xml	287	22	4	1	22	44	–	–
Violet.m	101	90	4	<1	28	4	124	233
Berkeley.m	78	47	3	<1	24	2	95	111
arcade_game_pl_fm.xml	61	35	4	<1	16	2	64	38
Gg4.m	38	23	6	<1	22	<1	63	7
...								

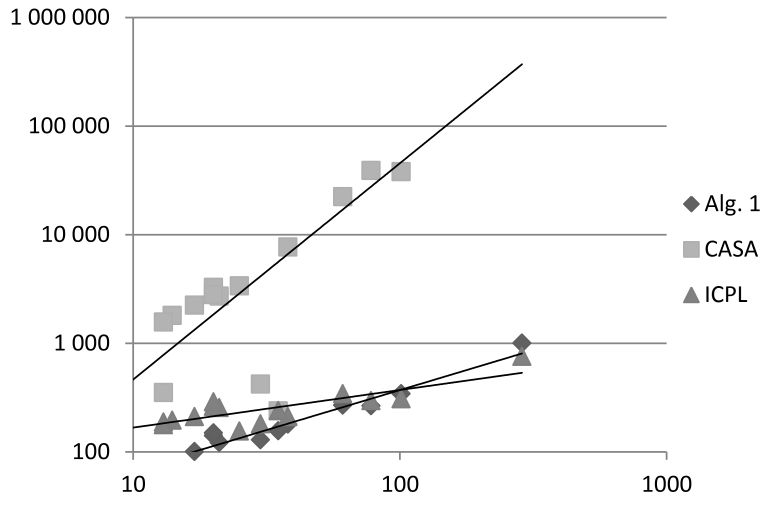


Figure C.1: Covering Array Sizes vs. Generation Time: 1-wise

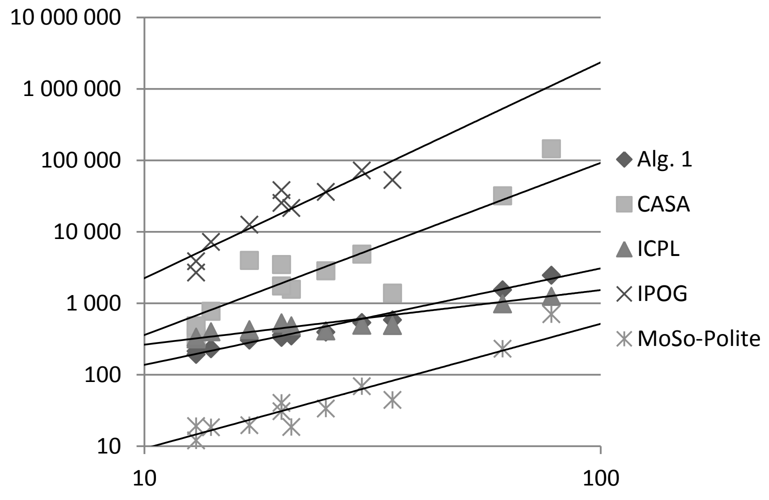


Figure C.2: Covering Array Sizes vs. Generation Time: 2-wise

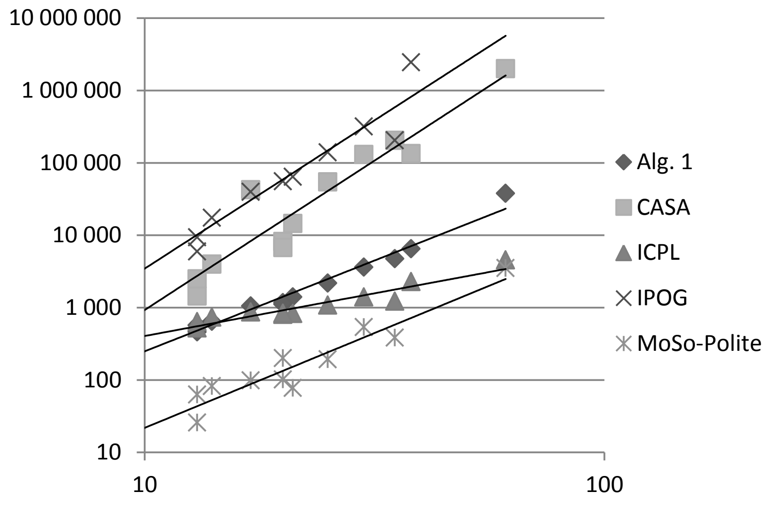


Figure C.3: Covering Array Sizes vs. Generation Time: 3-wise

Appendix D

Technical About the Tools

As a part of the research method, we implemented the algorithms that we made in order to evaluate them in realistic settings. The Software Product Line Covering Array Tool (SPLCA-Tool) implements various algorithms for working with feature models and covering arrays. It includes various libraries from the community; thus, it integrates with formats and algorithms in the research community and from industry.

D.1 Version History

Throughout the PhD project, the tool was updated to include our latest developed and published results. Each release of the tools was named after the conference its implemented results were published at. Here is the release history of the tools:

- **SPLCATool v0.1 (EuroPLoP 2011)** - Released 2011-06-14 as resource material for a paper presented at EuroPLoP 2011 (Paper 3).
- **SPLCATool v0.2 (MODELS 2011)** - Released 2011-05-03 as resource material for paper submitted to MODELS 2011 (Paper 1).
- **SPLCATool v0.3 (SPLC 2012)** - Released 2011-10-28 as resource material for a paper published at SPLC 2012 (Paper 2).
- **SPLCATool v0.4 (MODELS 2012)** - Released 2012-04-03 as resource material for paper submitted to MODELS 2012 (Paper 4).
- **Automatic CIT Tool Collection (ICTSS 2012)** - Released 2012-06-18 as resource material for a paper submitted to ICTSS 2012 (Paper 5).

The tool provided us with a framework onto which new functionality could be added onto and experimented with. It also was used to produce the empirical evaluations in the papers published.

D.2 Architecture

The tool works with three kinds of artifacts given to it by the user along with some instructions of what to do with them.

- **Feature Models:** In essence, any feature model that can be converted to a propositional formula can be loaded by the tool. We have, however, only implemented some of the major formats for feature modeling used in academia and industry. Whenever possible, the libraries associated with a particular format have been used. When this was not possible, custom loaders were developed. When the libraries contained bugs that stopped us, those bugs were fixed.

- **Covering Arrays:** A covering array is a set (or an array) of configurations of a feature model selected such that each t-sets, for some t, are covered (thereby the name *covering array*). Each configuration is a set of assignments such that all features are given an assignment and such that the configuration is valid according to the feature model.

- **Weighted Sub-Product Line Models:** Similarly as covering arrays, sub-product line models give assignments to features, but assignments can be left unassigned. The unassigned features thus maintain some variability, and the configurations of this partial feature model are a sub-set of all possible configurations. Therefore, they are called sub-product line models. The models are called weighted because each sub-product line model is given a positive integer as a weight.

The first set of functionalities provided by the tool is verification of the artifacts.

- **Feature models** have two basic requirements: (1) They must have valid syntax, and (2) they must have at least one valid configuration. Both of these are checked by running the `is_satisfiable` task.

- **Covering arrays** are valid and invalid relative to a feature model. Because a covering array is a set of configurations of the feature model, they must, to be valid, be valid configurations of the feature model. This is verified using the `verify_solutions` task.

- **Weighted sub-product line models** are also valid or invalid relative to a feature model. Because they are not configurations but sub-product lines, they have the same requirement as feature models: They must give at least one valid configuration. This is checked by running the `verify_weighted_solutions` task. In addition, the weights must be positive integers.

The second set of functionalities is the various algorithms:

- **Generating covering arrays:** A covering array is produced from a feature model and a strength t for the strength of the coverage. Various algorithms can be invoked for generating covering arrays. The algorithms all have different characteristics of (1) speed of generation, (2) size of resulting covering array, (3) variance in results between invocations and (4) supported additional features.

- **Calculating the coverage of a set of products:** Given a feature model and a set of configuration and a strength t, the t-wise coverage of the products given the feature model can be calculated. The coverage can be given either as a t-set coverage in percent of t-sets covered by the configurations or as weight coverage in percent of the weight covered by the configurations. To calculate weight coverage, a weighted sub-product line model is also required.

- **Generate a list of suggestions for improving the weight coverage of a set of products:** Given a feature model, a set of products and a weighted sub-product line model, the set of products might be improved by changing it slightly. Suggesting such small changes to improve coverage is the role of this class of functionality.

D.2.1 Feature Model Formats

The following five formats are supported.

- **GUI DSL Models (.m)** This is a feature model format used by the Feature IDE tools suite [155]. It is as a format by the AHEAD and GenVoca tools by Automated Software Design Research Group at the University of Texas at Austin.

- **Simple XML Feature Model (SXFM) (.xml)** SXFM is a feature model format used to store feature models for the Software Product Line Online Tools (SPLOT)[1] developed by the Generative Software Development Lab / Computer Systems Group, University Of Waterloo, Canada, 2010.

- **DIMACS CNF format (.dimacs)** This is the format used by the SAT4J library. It has become a standard format for Boolean formulas in CNF. It was initially proposed as a format for CNF formulas by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) [48].

- **CVL 1.x (.xmi or .cvl)** CVL is a variability-modeling format proposed in 2008 for adding variability mechanisms to DSLs [69]. It is currently being considered as a standard for variability-modeling by the OMG [71]. The version 2 of CVL is currently being developed. However, only the first version of the format is supported by SPLCATool. CVL models can be created using the CVL Tool[2].

  Due to the richness of CVL models, a feature model must be extracted from it before it can be loaded. Download the CVL Tool chain from the ICTSS 2012-paper resource page[3], unzip `CVLUtils.jar` and execute `java -cp CVLUtils;CVLUtils/lib/* CVL2GDSL model.cvl` to produce a Feature IDE GUI DSL file `model.cvl.m`. It can be loaded with the normal loading procedure.

D.2.2 Configuration Set Formats

Covering Arrays are a set of configurations of a feature model. Thus, except for their special property, they are no different from a set of configurations.

[1] `http://gdansk.uwaterloo.ca:8088/SPLOT/sxfm.html`, retrieved 2013-01-05

[2] `http://www.omgwiki.org/variability/doku.php/doku.php?id=cvl_tool_from_sintef`

[3] `http://heim.ifi.uio.no/martifag/ictss2012/`

- **Comma Separated Values (CSV):** Due to the use of the comma as a decimal mark in Norway, Norwegian CSV files are semi-colon separated instead of comma-separated.

  Configurations are stored in the following way, textually: The first entry in the file must be "Feature\Product;". This means that the features will be listed in the rows, and the products will be listed in the columns. This is the most practical arrangement for editing and viewing. Following this entry, the names of the products are listed. This can be as simple as 1, 2, 3 etc., written as "1;2;3;". On each line following the first line is a feature name followed by either an "X" or a "-" when the feature is included or excluded, respectively, for the product in that column. Editing these files in Excel or similar is of course recommended.

- **Excel Office Open XML (XSLX):** These files are edited using Excel or similar. They are loaded by the tool using the "Apache POI - Java API To Access Microsoft Format Files"-library. There are some additional requirements when using this format in order to delimit the product configurations and allow for additional comments and entries in the file: (1) the entry following the lower-most feature must be "#end" and (2) the entry following the last product name must also be "#end".

D.2.3 Weighted Sub-Product Line Formats

Weighted Sub-Product Line models are a set of configurations of a feature model with two additional things added. Thus, except for their two additions, they are stored similarly to covering arrays.

The two additions are: (1) Assignments can be left unassigned using a question mark. (2) The feature on the first row is named "#Weight". On this row, the weight for each sub-product line is given. It must be a positive, non-zero integer.

D.2.4 Versions

There are several versions and editions of the tool. Each version is named after the conference at which the results it implements are published. For one version, there are several editions because of licensing concerns.

- **v0.1 (EuroPLoP 2011):** This version introduced basic covering array generation for $1 \leq t \leq 3$ and implements the Bow-Tie reduction algorithm. The covering arrays were stored in the CSV format.

- **v0.2 (MODELS 2011):** This version introduced 4-wise covering array generation, SAT-timing, t-wise coverage calculation and the verification of the solutions in a covering array.

- **v0.3 (SPLC 2012):** This version of the tool introduced an implementation of the ICPL algorithm for $1 \leq t \leq 3$. In addition, it implements support for running the IPOG, CASA and MoSo-PoLiTe algorithms; however, these are only available in the Full edition of the tool. It cannot be freely distributed because of licensing concerns. However, an edition with the adaptors only (The Adaptors Only Edition) is available into which the three tools can be added to produce the full edition. The IPOG algorithms supports covering array generation for $1 \leq t \leq 6$, CASA for $1 \leq t \leq 6$ and MoSo-PoLiTe for $2 \leq t \leq 3$.

- **v0.4 (MODELS 2012):** This version of the tool introduced implementations of algorithms related to weighted sub-product line models. In addition, a GUI was introduced to make working with the models easier; the output is still written to the command line however. The command-line interface is still available by extracting `SPLCATool-v0.4-MODELS2012.jar` and calling `java -cp SPLCATool-v0.4-MODELS2012; SPLCATool-v0.4-MODELS2012 no.sintef.ict.splcatool.SPLCATool`.

  The tool introduced (1) the generation of covering arrays prioritized using the weighted models, (2) the verification of weighted sub-product line models, and (3) the improvement of existing covering arrays using the weighted sub-product line models. This version also introduced the loading of covering arrays in the XLSX format.

- **Automatic CIT Tool Collection (ICTSS 2012):** The Automatic CIT Tool introduced scripts, formats and the implementation of the algorithms for automatic CIT both for Eclipse- and CVL-based product lines. It also introduced a support for extracting GUI DSL models from CVL-models, thereby enabling the generation of covering arrays from feature models within CVL models.

D.2.5 Overview of Algorithms and Object Structures

Inside the tool, the different formats are imported and converted into other formats in order to be applicable for various algorithms. Figure D.1 shows an overview of the transformations, what formats are fed to the different algorithms and what these algorithms produce. The figure is of no particular modeling notation. First of all, feature models can be given to the tool in the four different formats. They are converted in one direction. After being converted, some of the formats have exporting to file implemented. Covering arrays are read in from CSV and XLSX files. Weighed sub-product line models can also be read in from CSV and XLSX files. The rectangular boxes symbolize internal object-structures; the arrows symbolize algorithms that generate some other data.

The various algorithms available are further described in the basic user manual, Section D.3.

D.3 User Manual

This basic user manual explains how to use the tool using the Eclipse case study as a running example. The four files for the Eclipse case study are available on the resource page of the MODELS 2012-paper[4].

- Products — `eclipse-red.m.actual.csv`
- Feature model — `Eclipse.m`
- Weights — `eclipse-red.m.actual-weighted.csv`
- Ordering — `eclipse.m.order.csv`

[4]`http://heim.ifi.uio.no/martifag/models2012/`

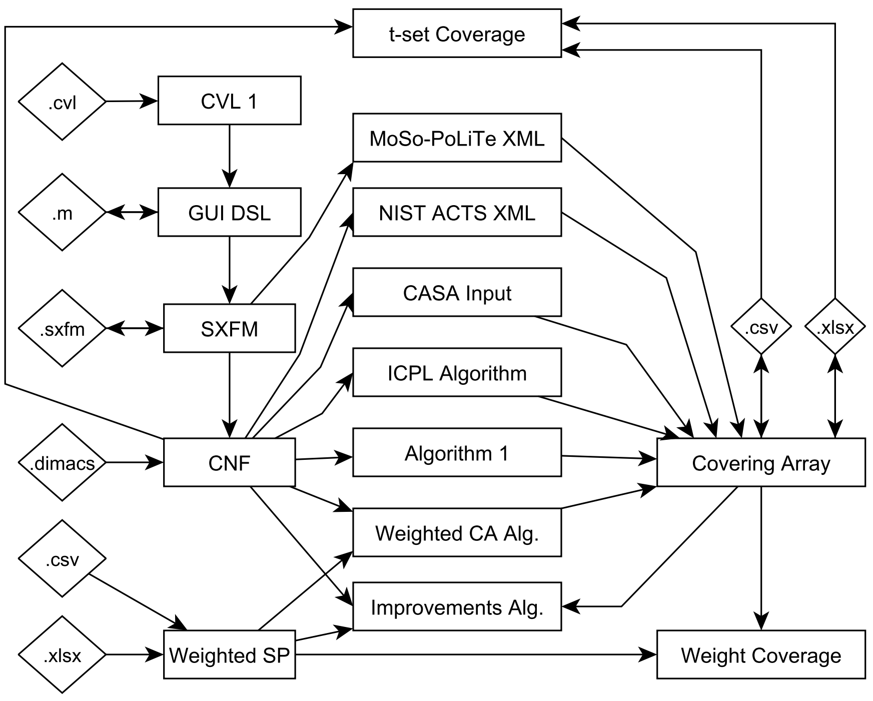


Figure D.1: Transformations in the Tool

The products are the configurations of the 12 products that were available for download on the Eclipse v3.7.0 (Indigo) web site. They are valid configurations of the feature model that captures the relations between the different features of the Eclipse product line. The weights-file contains the current market-situation—it was captured by adding the number of downloads to each product offered. This is a basic approximation because we have no clue as to how the users proceeded to configure their products. The ordering files contain the order the features should be listed in the generated files. This is necessary because there is no intrinsic order to the features in the other files.

Throughout the manual, the command-line will be used. Only the arguments of the command-line invocations will be shown.

D.3.1 Analysis

First, let us check the validity of the files and find out some basic information about them. Let us first check the feature model. The `sat_time` task (v0.4) will attempt to load the feature model; if successful, it will print some basic information about it and try to find a single valid configuration. It will then report whether one exists and how long time it took to find it, its satisfiability-time. Execute `-t sat_time -fm Eclipse.m` (v0.4) to run the task. This information is returned:

- Valid feature model?: yes
- Number of Features: 26
- Number of CNF-clauses of the composed constraint: 6
- Satisfiable?: yes
- Satisfying time: 0.446 ms

Thus, we know several important things: The feature model is valid and it has a very fast satisfiability time. This means it is fully usable for all other tasks.

Let us try to find one additional piece of information, the number of valid configurations. Run `-t count_solutions -fm Eclipse.m` (v0.4). This task does not scale very well; up to about 200 features should work on modern hardware.

- Number of possible configurations: 1,900,544.

This is an enormous number; $log_2(1,900,544) \approx 21$, which is relatively close to 26, the number of features of the feature model. If each feature is a yes-or-no choice, and each choice was independent, then the number of configurations would be 2^f, where f is the number of features. Features normally are not independent, but 2^f still gives us an idea of the magnitude of the number of configurations.

Let us now analyze the products. Run `-t verify_solutions -fm Eclipse.m -check eclipse-red.m.actual.csv` (v0.4). This task will check whether the configurations are valid according to the feature model. It tells us that it is. Had it been invalid, the tool would output the CNF-clauses that were not satisfied. This gives us a good guide as to where to look for the problem.

Let us analyze the weighted sub-product line model. Run `-t verify_weighted_solutions -fm Eclipse.m -check eclipse-red.m.actual-weighted.csv` (v0.4). This task will check whether the partial configurations are valid. It tells us that it is. Again, had it been invalid, it would tell us which CNF-clauses were not satisfied.

- Configurations valid?: yes
- Weighted sub-product line models valid?: yes

Finally, let us find the t-set and weight coverage of the products. Run `-t calc_cov -fm Eclipse.m -s <t> -ca eclipse-red.m.actual.csv` (v0.4) to find the t-set coverage and `-t calc_cov_weighted -fm Eclipse.m -s <t> -ca eclipse-red.m.actual.csv -weights eclipse-red.m.actual-weighted.csv` (v0.4) to find the weight coverage, both for $1 \leq t \leq 3$. That gives the following coverages:

- 1-set coverage: $49/50 \approx 98.0\%$
- 2-set coverage: $929/1,188 \approx 78.2\%$
- 3-set coverage: $10,034/17,877 \approx 56.1\%$
- 1-wise weight coverage: $52,949,676/52,949,676 = 100\%$
- 2-wise weight coverage: $661,870,950/661,870,950 = 100\%$
- 3-wise weight coverage: $2,623,612,957/2,623,612,957 = 100\%$

It is not a surprise that the weight coverage is 100%. It is because the weighted sub-product line model contains only the products that are offered. It would be better to have actual information about the market, but such information is unavailable, unfortunately. The TOMRA models are unavailable due to sensitivity concerns.

The denominators of the t-set coverage are 50, 1,188 and 17,877. These are the number of valid 1, 2 and 3-sets, respectively. When all are covered, we have a complete t-wise covering array.

The denominators of the weight coverage are 52,949,676, 661,870,950 and 2,623,612,957. These are the total weight for all valid 1, 2 and 3-sets, respectively. When all the weight is covered, the weighted covering array has a coverage of 100%.

D.3.2 Generating Covering Arrays

Now that we have analyzed all our models and checked their validity, let us generate some new things and then verify them.

Generate a t-wise covering array for $1 \leq t \leq 3$ using ICPL by running `-t t_wise -a ICPL -fm Eclipse.m -s <t>` (v0.3 (ICPL Edition)). This results in the following files containing the covering arrays.

- Eclipse.m.ca1.csv, products: 2, generation time: 26 ms
- Eclipse.m.ca2.csv, products: 12, generation time: 136 ms
- Eclipse.m.ca3.csv, products: 39, generation time: 663 ms

Calculating the coverage of each using the same method as discussed above yields 100% 1, 2 and 3-wise coverage, respectively, as expected. Notice that the 12 products offered by Eclipse have a 2-wise coverage of 78.2% while 12 products are also needed to yield 100% 2-wise coverage!

It is also possible to calculate covering arrays using other algorithms. Because of licensing issues, we cannot distribute the full edition of the tool. However, once these tools have been acquired, the following commands can be used to calculate covering arrays using other algorithms: `-t t_wise -a <a> -fm Eclipse.m -s <t>` (v0.3 (Full Edition)) where `a` is `IPOG`, `CASA` or `MoSoPoLiTe`. These three support $1 \leq t \leq 6$, $1 \leq t \leq 6$ and $2 \leq t \leq 3$, respectively.

If 12 products, a complete 2-wise covering array, are too costly for testing in your context, and you would like to get a smaller set that is still related to the market situation, you can use weighted covering array generation. Let us say we decide to cover 95% of the weight. Run `-t t_wise_weighted -a ChvatalWeighted -fm Eclipse.m -s <t> -weights eclipse-red.m.actual-weighted.csv -limit 95%` (v0.4) for $2 \leq t \leq 3$ to produce the following results:

- Eclipse.m.ca2.csv, products: 4, generation time: 341 ms
- Eclipse.m.ca3.csv, products: 3, generation time: 260 ms

Calculating the weighted coverage using the methods discussed above yields 97.05% and 95.08% 2 and 3-wise weight coverage, respectively. Both are, as required, equal to or above 95%.

D.3.3 Extending Towards a Covering Array

A realistic situation when doing product line testing is that certain popular and important products must be tested. For example, for Eclipse, 3 products account for 88% of the downloaded products. Instead of just adding these 3 products to the already complete covering arrays, we could start from them to probably make them be a part of the covering array.

Extract the "Java", "Java EE" and "Classic" products into a file called `eclipse-red.m.important.csv`. To include them in a covering array run `-t t_wise -a ICPL -fm Eclipse.m -s <t> -startFrom eclipse-red.m.important.csv` (v0.3 (ICPL Edition)). The extended versions are:

- Eclipse.m.ca1.csv, products: 5, time: 27 milliseconds
- Eclipse.m.ca2.csv, products: 13, time: 120 milliseconds
- Eclipse.m.ca3.csv, products: 40, time: 507 milliseconds

For 1-wise testing, this does not reduce the effort because there were three important products, but only two were required to achieve 1-wise coverage. For 2-wise testing, the new covering array has 13, only one more than a fresh 2-wise covering array. For 3-wise testing, the new covering array has 40 products, also only one more than the 39 required when building a fresh covering array.

We can also extend covering arrays based on weight. If we start with the three important products again, let us see what is needed to extend with one more product. We could also extend based on percentages, but this opportunity is used to demo the size limits.

First of all, the weight coverages of the 3 important products are 98.68%, 97.05% and 95.08%, respectively. Run the following command to extend the set of products with a maximum of two new product based on maximizing the weight coverage starting from three important products for $1 \leq t \leq 3$: `-t t_wise_weighted -a ChvatalWeighted -fm Eclipse.m -s <t> -weights eclipse-red.m.actual-weighted.csv -startfrom eclipse-red.m.important.csv -sizelimit 5` (v0.4). The new set of products of sizes 4, 5 and 5 has weight coverages of 100%, 99.5% and 98.5%, respectively.

D.3.4 Improving a Partial Covering Array

In settings where a complete test system has been set up; for example, if the products of a hardware product line has been physically set up, it is not good to have drastic changes to the product configurations.

The tool supports suggesting small changes to an existing set of products to improve its weight coverage.

An especially important usage of this is in the following situation: A product line gets a new feature added to it. The company already has a test lab of, say, 12 products set up. The question now is: To which product is it best to add this feature to maximize the coverage with respect to the market situation (the weight coverage)?

First add a new feature to `Eclipse.m` named `X` as an optional feature of `RCP_Platform`, then add a new row to the set of product offered by Eclipse with X set to all excluded ('-'), then finally, add a new row to the weighted sub-product line model with all unassigned ('?'). This row of unassigneds tells the tool, in effect, that we do not know who will be using the new feature, so we had better ensure that it is exercised during testing and that it is tested out together with the other features.

To have the tool figure this out automatically, run `-t improve_weighted -fm Eclipse.m -s 2 -ca eclipse-red.m.actual.csv -weights` `eclipse-red.m.actual-weighted.csv -search 1 > list.txt` (v0.4). Here, we assume that the offered products are also the products that the Eclipse Projects tests, a realistic assumption. We want to maximize our 2-wise interaction coverage, which means that we want our new feature to be exercised against any other feature. Finally, we want to maximize the weight coverage to ensure market relevance. Because we only want to activate a single feature, we only search for single changes, `-search 1`. Running the task results in the following results:

- Original weight coverage: 96.3%
- Suggested change: Set feature X to included for the product named "1" (the second product).
- New weight coverage: 99.3%.
- Time taken: about 2 seconds.

D.3.5 Automatic CIT: Eclipse-Based Product Lines

This part of the manual explains how to apply Automatic CIT to Eclipse-based product lines. The first thing that is needed is to set up the directory structure where all files will go. First, decide on a root directory in which all files related to the testing will be stored. In this directory, create these directories:

- `models` — contains all input to the testing tools.
- `packages` — contains all packages to be used to construct the software to be tested.
- `products` — will store all built and tested products and their associated workspaces.
- `results` — will store all generated results.
- `scripts` — contains all scripts used for testing.
- `tools` — contains all tools used for testing.
- `workspace` — workspace for the Eclipse tool used to build the package repository.

The next step is to acquire the required tools. Links are available from the ICTSS 2012-paper resource page[5]. Note that all downloads mentioned in this part of the manual is found on that resource page. Download and extract the following free tools in the "tools" directory.

- 7-Zip Command Line Version (9.20)
- Eclipse Platform (3.7.0)
- UnxUtils (of 2007-03-01), ports of the most important GNU utilities to Windows
- An implementation of Algorithm 3 from the paper

The next step is to build the package repository. Download the mirroring-script, and place it in the scripts directory. Download and edit the environment set up script in which the path to the java.exe, cmd.exe and the root directory must be specified. Executing the mirroring script will construct mirrors for the Eclipse Indigo repositories in "%basedir%/packages/repos/". This will take a while and might require rerunning the script several times to ensure that all packages

[5]Appendix A or `http://heim.ifi.uio.no/martifag/ictss2012/`

were downloaded correctly. The size of these repositories during the experiments presented in the paper was 3.6 GiB.

Place a copy of the package containing the Eclipse Platform in "packages". Download the Eclipse Automated Tests for Eclipse 3.7.0. Inside it there is a zip-file called `eclipse-testing/eclipse-junit-tests-I20110613-1736.zip`. Extract it in the "packages" directory.

The next step is setting up the required models. The technique requires three files to be built: (1) The feature model, for example such as the one for the Eclipse IDEs. (2) The mapping between features and code artifacts, as for example the one used in the experiment in the paper. (3) The mapping between features and tests, as for example the one used in the experiment in the paper. Examples of these files for the Eclipse 3.7.0 system are available on the resource page.

The next step is execution. The following command will execute Automatic CIT: `java no.sintef.ict.splcatool.lasplt.LTSPLT <basedir> <t> <timeout>`.

`basedir` is the root directory of the testing system, t is the strength of the combinatorial interaction testing and timeout is the time in seconds after which a test suite execution will be aborted.

When the feature model, the features themselves or the test suites are changed, there is no additionally required manual work to redo the testing other than re-executing the above command.

Output such as the following will be produced on the command line:

```
Converting Package Mapping File to CSV [done]
Converting Feature-Test Mapping File to CSV [done]
Loading Mapping Files [done]
Generating 2-wise Covering Array [done]
Loading Covering Array [done] Products: 14
Test Product 0:
  Installing base product: [done]
  Installing features:
    Installing feature EclipseIDE [nothing to install]
    Installing feature RCP_Platform [nothing to install]
  Installing tests:
    Installing tests for EclipseIDE
    Installing tests for RCP_Platform
      Installing test org.eclipse.test.feature.group [done]
      ...
      Installing test org.eclipse.search.tests [done]
  Running tests:
    Running tests for EclipseIDE
    Running tests for RCP_Platform
      Running test org.eclipse.ant.tests.core.AutomatedSuite [done] 100%
          (85/85)
      Running test org.eclipse.compare.tests.AllTests [done] 89% (101/113)
      Running test org.eclipse.core.internal.expressions.tests.AllTests [
          done] 100% (108/108)
      ...
      Running test org.eclipse.search.tests.AllSearchTests [already exists]
           78% (29/37)
```

```
Test Product 1:
  Installing base product: [already exists]
  Installing features:
    Installing feature WindowBuilder [done]
    Installing feature SVN15 [done]
    Installing feature EclipseIDE [nothing to install]
    ...
    Installing feature RCP_Platform [nothing to install]
    Installing feature Datatools [done]
    Installing feature SVN [nothing to install]
...
Test Product 13:
...
```

As a final stage, it is possible to generate a web-view for the results. This is good for using Automatic CIT in conjunction with continuous integration system such as Hudson[6] or Jenkins[7].

The command in the previous step will produce logs containing the contents of *stdout* and *stderr* during test execution called `failure-<product nr>-<test class name>.log` and test result details in files called `product<product nr>-<test class name> -test-result.xml`. The results produced by the experiment reported in the paper are available on the resource page of Paper 5.

These results can be compiled into a report by executing the following command: `java no.sintef.ict.splcatool.lasplt.GenerateWebView <basedir>` This will produce an HTML-document. An example for the Eclipse IDE experiment is found on the resource page of Paper 5.

D.3.6 Automatic CIT: CVL Based Product Lines

Automatic CIT can be applied to CVL-based product lines in a similar way to Eclipse-based product lines. The set up required is found in `CVLLASPLTInput.java` a template found in CVL tool chain available from the resource page website. The guidelines for Eclipse-based product lines can be used to understand how to run Automatic CIT for CVL-based product lines.

[6]`http://hudson-ci.org/`
[7]`http://jenkins-ci.org/`

Appendix E

Details on the Eclipse Case Study

The Eclipse IDEs are used as a case study in the EuroPLoP 2011, MODELS 2012 and ICTSS 2012 papers. All experiments used Version 3.7.0 (Indigo) of Eclipse; it was released in the summer of 2011.

E.1 Technical About Eclipse v3.7.0 (Indigo)

The architecture and design of Eclipse is documented in Rivieres and Beaton 2006 [132]. The development of Eclipse is open, and most development artifacts, choices made and design decisions are recorded online.

Twelve products are offered online by the Eclipse Project. The configurations of these products are shown online as a matrix, shown in Figure E.1.

These products are available for 15 combinations of operating system, windowing systems and hardware architectures. The list of 15 builds available from the Eclipse project is shown in Figure E.2.

Further, the Eclipse Project specifies which versions are supported. These are shown in Figure E.3[1]. According to their documentation, "Eclipse 3.7 is tested and validated on the following reference platforms".

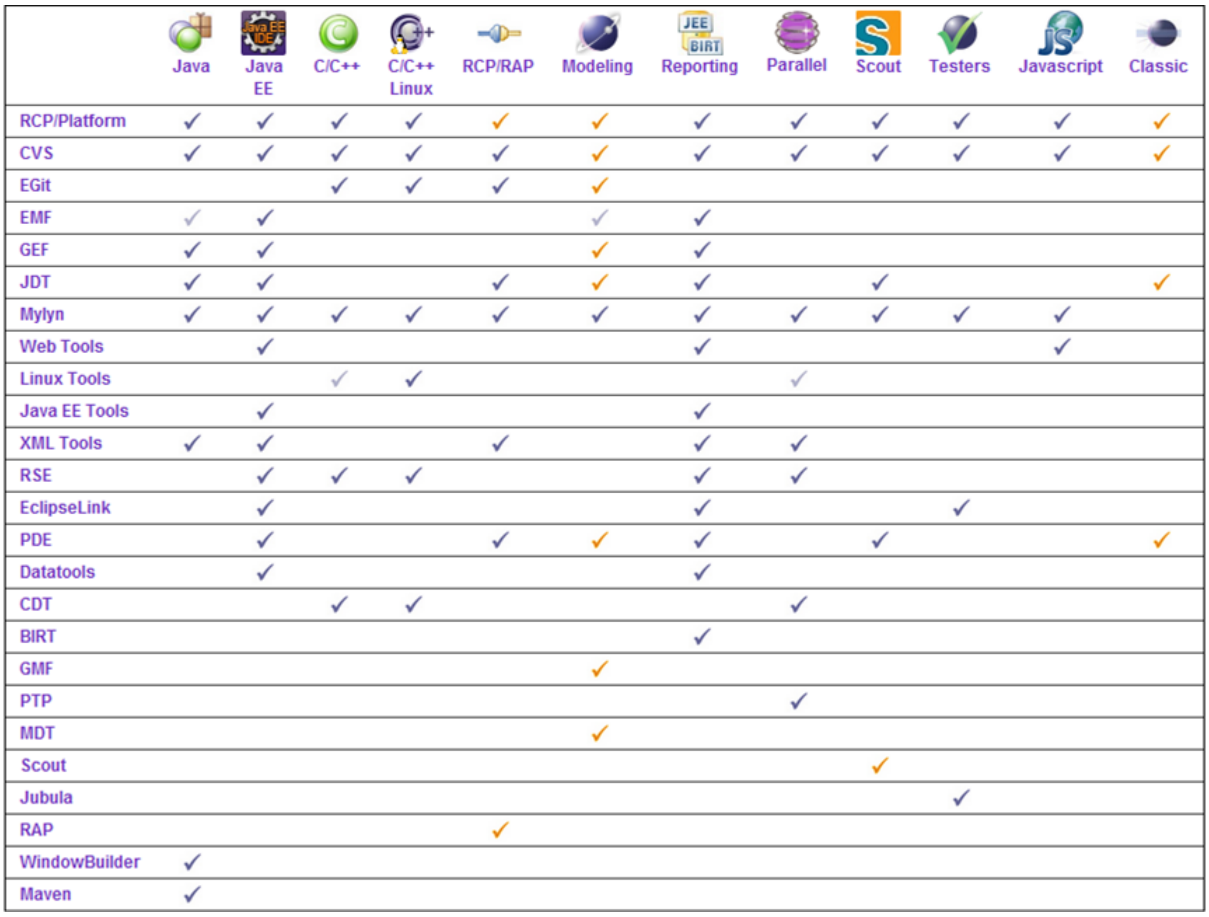
Eclipse IDE products are built from OSGi bundles [63]. These software assets are identified by a pair $(id : ID, version : V)$, where id is an ID string unique for that bundle conforming to a grammar for IDs which is the same as the naming conventions for Java packages, and $version$ is a versioning string valid according to one of several version grammars. The version grammar used by the Eclipse plug-in system is $i(.i(.i(.s)?)?)?$ where i are positive integers including 0 and s is a string. The integers are respectively called 'major', 'minor', then either 'micro' or 'service', and, finally, the 's' is called 'qualifier'.

Bundles are stored in Java archives named $id\_version.jar$. They are published online[2].

The basic Eclipse platform is distributed as a zip-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 to be installed automatically using the following command:

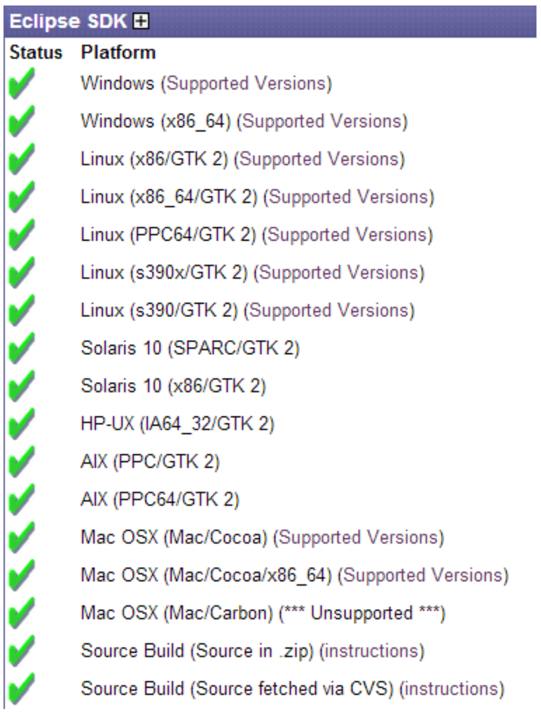
[1]`http://web.archive.org/web/20110608032901/http://www.eclipse.org/projects/project-plan.php?projectid=eclipse`

[2]`http://download.eclipse.org/releases/indigo/`



	Java	Java EE	C/C++	C/C++ Linux	RCP/RAP	Modeling	Reporting	Parallel	Scout	Testers	Javascript	Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EGit			✓	✓	✓	✓						
EMF	✓	✓				✓	✓					
GEF	✓	✓				✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Web Tools		✓					✓				✓	
Linux Tools			✓	✓				✓				
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓				
RSE		✓	✓	✓			✓	✓				
EclipseLink		✓					✓			✓		
PDE		✓			✓	✓	✓		✓			✓
Datatools		✓					✓					
CDT			✓	✓				✓				
BIRT							✓					
GMF						✓						
PTP								✓				
MDT						✓						
Scout									✓			
Jubula										✓		
RAP					✓							
WindowBuilder	✓											
Maven	✓											

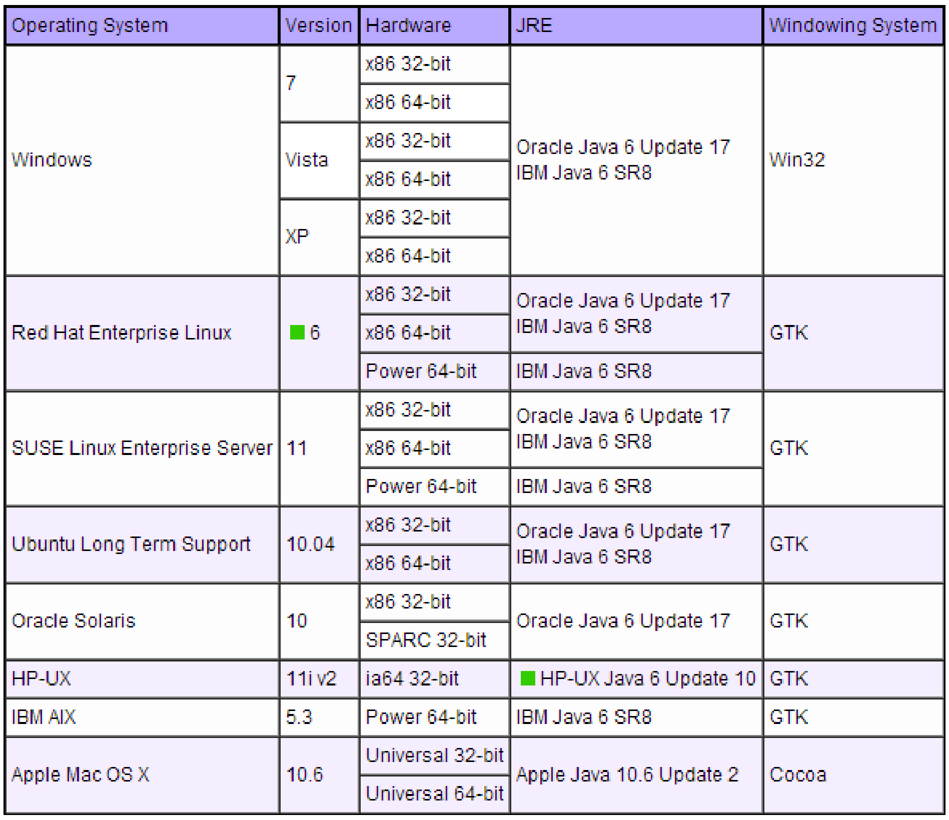
Figure E.1: Product Configuration of the Twelve Eclipse IDE Products Offered by the Eclipse Project



Eclipse SDK ⊞

Status	Platform
✓	Windows (Supported Versions)
✓	Windows (x86_64) (Supported Versions)
✓	Linux (x86/GTK 2) (Supported Versions)
✓	Linux (x86_64/GTK 2) (Supported Versions)
✓	Linux (PPC64/GTK 2) (Supported Versions)
✓	Linux (s390x/GTK 2) (Supported Versions)
✓	Linux (s390/GTK 2) (Supported Versions)
✓	Solaris 10 (SPARC/GTK 2)
✓	Solaris 10 (x86/GTK 2)
✓	HP-UX (IA64_32/GTK 2)
✓	AIX (PPC/GTK 2)
✓	AIX (PPC64/GTK 2)
✓	Mac OSX (Mac/Cocoa) (Supported Versions)
✓	Mac OSX (Mac/Cocoa/x86_64) (Supported Versions)
✓	Mac OSX (Mac/Carbon) (*** Unsupported ***)
✓	Source Build (Source in .zip) (instructions)
✓	Source Build (Source fetched via CVS) (instructions)

Figure E.2: Builds Available from the Eclipse Projects



Operating System	Version	Hardware	JRE	Windowing System
Windows	7	x86 32-bit	Oracle Java 6 Update 17 IBM Java 6 SR8	Win32
		x86 64-bit		
	Vista	x86 32-bit		
		x86 64-bit		
	XP	x86 32-bit		
		x86 64-bit		
Red Hat Enterprise Linux	🟩 6	x86 32-bit	Oracle Java 6 Update 17 IBM Java 6 SR8	GTK
		x86 64-bit		
		Power 64-bit	IBM Java 6 SR8	
SUSE Linux Enterprise Server	11	x86 32-bit	Oracle Java 6 Update 17 IBM Java 6 SR8	GTK
		x86 64-bit		
		Power 64-bit	IBM Java 6 SR8	
Ubuntu Long Term Support	10.04	x86 32-bit	Oracle Java 6 Update 17 IBM Java 6 SR8	GTK
		x86 64-bit		
Oracle Solaris	10	x86 32-bit	Oracle Java 6 Update 17	GTK
		SPARC 32-bit		
HP-UX	11i v2	ia64 32-bit	🟩 HP-UX Java 6 Update 10	GTK
IBM AIX	5.3	Power 64-bit	IBM Java 6 SR8	GTK
Apple Mac OS X	10.6	Universal 32-bit	Apple Java 10.6 Update 2	Cocoa
		Universal 64-bit		

Figure E.3: Supported Versions of Environment Artifacts

```
<eclipse executable> -application org.eclipse.equinox.p2.director -
   repository <repository1,...> -installIU <id>/<version>
```

E.2 Contributed Artifacts

As for all of the other case studies, Eclipse does not employ an explicit product line methodology. As is quite clear, however, their approach is at least very close. It was quite easy to build a feature model by browsing through their bundles.

From any bundle $b \in B$ it is possible to extract a set b_d with the dependencies of bundle b. The dependencies is a set of pairs $(id : ID, vr : VR)$, where id is an ID and vr is a string specifying a range of versions according to grammars of ranges of versions. Figure E.4 shows the complete feature diagram of the Eclipse IDEs. The products shown in E.2 are of course valid (partial) configurations of this feature model.

E.3 Details on Automatic CIT of Eclipse

This section details the large-scale application of the Automatic CIT technique to the entire Eclipse IDE Indigo (v3.7.0) product line supported and maintained by the Eclipse Project.

For the experiment in Paper 5, the part of the Eclipse IDE product line shown in Figure E.5 was selected for testing; three additional features were included. The products offered by

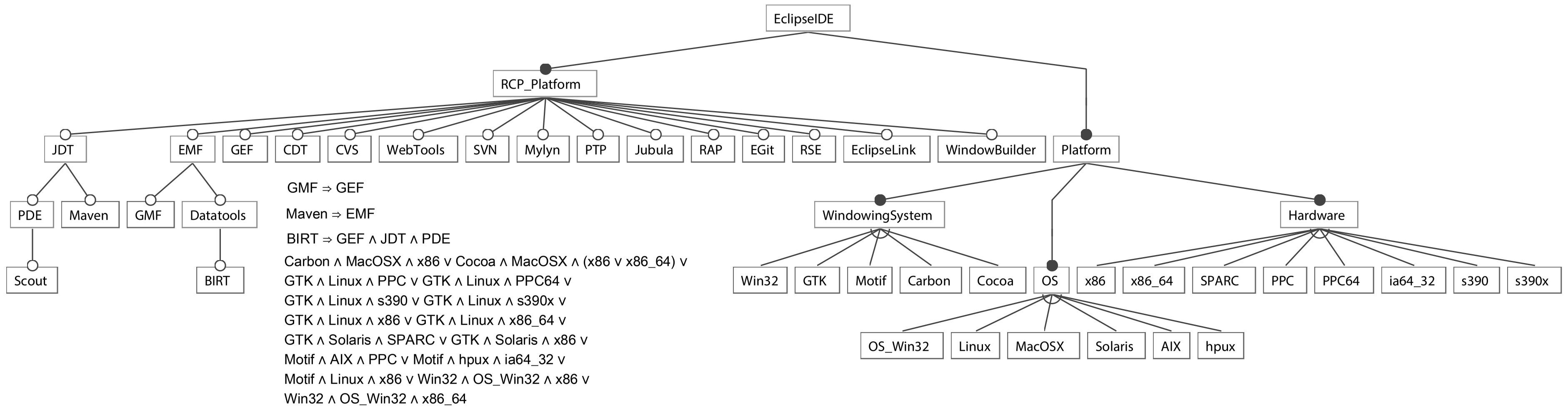

EclipseIDE
RCP_Platform
JDT
EMF
GEF
CDT
CVS
WebTools
SVN
Mylyn
PTP
Jubula
RAP
EGit
RSE
EclipseLink
WindowBuilder
Platform
PDE
Maven
GMF
Datatools
Scout
BIRT
GMF ⇒ GEF
Maven ⇒ EMF
BIRT ⇒ GEF ∧ JDT ∧ PDE
Carbon ∧ MacOSX ∧ x86 ∨ Cocoa ∧ MacOSX ∧ (x86 ∨ x86_64) ∨
GTK ∧ Linux ∧ PPC ∨ GTK ∧ Linux ∧ PPC64 ∨
GTK ∧ Linux ∧ s390 ∨ GTK ∧ Linux ∧ s390x ∨
GTK ∧ Linux ∧ x86 ∨ GTK ∧ Linux ∧ x86_64 ∨
GTK ∧ Solaris ∧ SPARC ∨ GTK ∧ Solaris ∧ x86 ∨
Motif ∧ AIX ∧ PPC ∨ Motif ∧ hpux ∧ ia64_32 ∨
Motif ∧ Linux ∧ x86 ∨ Win32 ∧ OS_Win32 ∧ x86 ∨
Win32 ∧ OS_Win32 ∧ x86_64
WindowingSystem
Hardware
Win32
GTK
Motif
Carbon
Cocoa
OS
x86
x86_64
SPARC
PPC
PPC64
ia64_32
s390
s390x
OS_Win32
Linux
MacOSX
Solaris
AIX
hpux

Figure E.4: Feature Diagram of the part of Eclipse IDE Indigo (v3.7.0) supported by the Eclipse Project

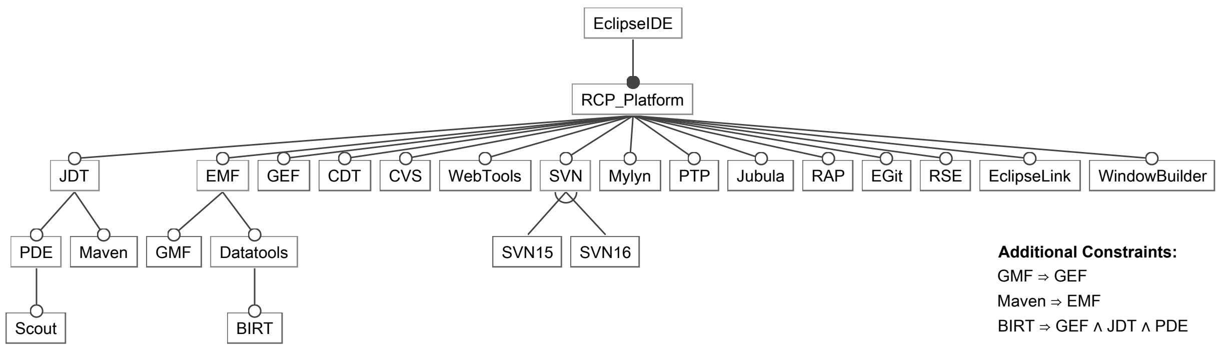


Figure E.5: Feature Model for the Eclipse IDE Product Line

Table E.1: Eclipse IDE Products, Instances of the Feature Model in Figure E.5

(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	-	-

Eclipse and a 2-wise covering array are shown in Table E.1.

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 fresh copy of the bare Eclipse Platform, which is an Eclipse IDE with just the core features. When all features of a product have been installed, we can run the test suite associated with each feature (or a combination of features).

E.3.1 Feature Models and Covering Arrays

There are many tools for doing feature modeling and generating covering arrays. For our implementation we use Feature IDE to model the feature model as shown in Figure E.5, and our SPL Covering Array Tool suite introduced in Paper 4 which is freely available on the paper's resource website. The covering array generated from the running example using this tool is presented in Table E.1b.

E.3.2 Implementation with the Eclipse Platform's Plug-in System

Algorithm 9 shows the algorithm of our testing technique for the Eclipse Platform plug-in system[3]. 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 E.5. The algorithm loops through each configuration in the covering array. In the experiment, it was the one given in Table E.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 be used. It contains the capabilities to allow each feature and test suite to be installed automatically using the following command:

```
<eclipse executable> -application org.eclipse.equinox.p2.director -
   repository <repository1,...> -installIU <id>/<version>
```

Similar commands allow tests to be executed.

A mapping file provides the links between the features and the test suites. This allows Algorithm 9 to select the relevant tests for each product and to run them against the build of the Eclipse IDE. The results are put into their respective entry in the result table. (An example is shown later in Table E.4b.)

[3]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 9 Pseudo Code of Eclipse-based version of Automatic CIT

```
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
```

Table E.2: Part of the Relation between Feature Names and Eclipse Bundle ID, Version and Repository

Feature	Bundle	Version	Repository
CVS	org.eclipse.cvs	1.3.100...	RepoA
EGit	org.eclipse.egit	1.0.0...	RepoA
EMF	org.eclipse.emf	2.7.0...	RepoA
GEF	org.eclipse.gef	3.7.0...	RepoA
JDT	org.eclipse.jdt	3.7.0...	RepoA
PDE	org.eclipse.pde	3.7.0...	RepoA
CDT	org.eclipse.cdt	8.0.0...	RepoA
BIRT	org.eclipse.birt	3.7.0...	RepoA
GMF	org.eclipse.gmf	1.5.0...	RepoA
SVN16	org.polarion.eclipse.team.svn.connector.svnkit16	2.2.2...	RepoA, RepoB
SVN15	org.polarion.eclipse.team.svn.connector.svnkit15	2.2.2...	RepoA, RepoB
...	...	...	...

E.3.3 Bindings Between Feature Names and System Artifacts

We require bindings from the feature names to the system artifacts. Feature names are found in the feature model and in the covering array. For Eclipse-based product lines, system artifacts are identified as with a bundle ID and a version. We also need to know where we can find the system artifacts. Thus, we also need the URL(s) of the artifact and its dependencies. This information can be easily entered in a spreadsheet and exported as a Comma Separated Values (CSV) file. The testing system takes this file as input along with the feature model.

For the running example, an excerpt from such a relation is shown in Table E.2[4]. Note that the table has been reduced for presentation in this document. The bundles all really end with `.feature.group`, some versions end with a hash and the repositories have been replaced with a symbolic name. `RepoA` is the Eclipse update site, and `RepoB` is the Polarion update site for subversive.

E.3.4 Automatic Building

We are now in a position to automatically build any Eclipse product. Start with a bundle of the Eclipse platform, and, for each product, run the following command for each feature:

```
<eclipse executable> -application org.eclipse.equinox.p2.director -
    repository <repository1,...> -installIU <bundle>/<version>
```

Since the feature name is bound to the bundle ID, bundle version and repositories, we can simply enter that information to the above boilerplate. After this is done all the products are set up.

For the running example, setting up the 13 products of the covering array took about 1 hour, and it resulted in about 2.9 GB of Eclipse IDE products. The repositories were locally mirrored[5]. For regular usage of this technique, it would be impractical to download the same files again and again. Therefore, executing the technique with the local mirror is realistic.

[4]The full tables from the experiment as well as the implementation is available online at `http://heim.ifi.uio.no/martifag/ictss2012/`. On this website, everything to reproduce out experiment is available for free.

[5]The local cashes of the update sites takes 3.09 GB and took about 6–7 hours to download at SINTEF.

Table E.3: Part of the Relation between Feature Names, Test Bundles, Versions and Repositories, Test Classes and Test Applications.

Feature	TestApp	Bundle	Version	Class	Repository
RCP_Platform		[oe].test.feature.group	3.5.0...		RepoC
RCP_Platform		[oe].ant.core	3.2.300...		RepoC
RCP_Platform		[oe].ui.tests	3.6.0...		RepoC, RepoA
CVS	UITA	[oe].team.tests.core	3.7.0...	[oe].team.tests.core.AllTeamTests	RepoC
CVS	UITA	[oe].team.tests.core	3.7.0...	[oe].team.tests.core.AllTeamUITests	RepoC
PDE	CoreTA	[oe].pde.api.tools.tests	1.1.100...	[oe].pde.api.tools.tests.ApiToolsPluginTestSuite	RepoC
PDE	CoreTA	[oe].pde.ds.tests	1.0.0...	[oe].pde.internal.ds.tests.AllDSModelTests	RepoC
JDT	CoreTA	[oe].jdt.compiler.tool.tests	1.0.100...	[oe].jdt.compiler.tool.tests.AllTests	RepoC
JDT	CoreTA	[oe].jdt.core.tests.builder	3.4.0...	[oe].jdt.core.tests.builder.BuilderTests	RepoC
JDT	CoreTA	[oe].jdt.core.tests.compiler	3.4.0...	[oe].jdt.core.tests.compiler.parser.TestAll	RepoC
JDT	UITA	[oe].jdt.apt.tests	3.3.400...	[oe]..jdt.apt.tests.TestAll	RepoC
JDT	UITA	[oe].jdt.ui.tests	3.7.0...	[oe].jdt.ui.tests.LeakTestSuite	RepoC
...	...	...	...	...	...
RCP_Platform	CoreTA	[oe].core.filebuffers.tests	3.5.100...	[oe].core.filebuffers.tests.FileBuffersTestSuite	RepoC
RCP_Platform	CoreTA	[oe].core.tests.net	1.2.100...	[oe].core.tests.net.AllNetTests	RepoC
RCP_Platform	CoreTA	[oe].core.tests.runtime	3.7.0...	[oe].core.tests.runtime.AutomatedTests	RepoC
RCP_Platform	CoreTA	[oe].equinox.security.tests	1.0.100...	[oe].equinox.security.tests.AllSecurityTests	RepoC
RCP_Platform	UITA	[oe].search.tests	3.6.0...	[oe].search.tests.AllSearchTests	RepoC
RCP_Platform	UITA	[oe].text.tests	3.6.0...	[oe].text.tests.EclipseTextTestSuite	RepoC
RCP_Platform	UITA	[oe].ui.editors.tests	3.4.100...	[oe].ui.editors.tests.EditorsTestSuite	RepoC
...	...	...	...	...	...

E.3.5 Bindings Between Feature Names and Test Suites

In order to test these products we need some test suites. Test suites are also bundled up as Eclipse plug-ins. Thus, we can use the same strategy as with the feature bundles.

For each (1) feature name we need (2) a bundle name, (3) a bundle version and (4) one or more repositories where the bundle and its dependencies can be found. In order to run the tests, we need (5) the name of the test application that will run the tests and (6) the name of the class with the test suite. These six things can be entered in a spreadsheet and exported as a CSV-file for the tool to use.

For each feature included in a product, the same command as for installing bundles can be used.

An example of this for the Eclipse IDE is shown in Table E.3. In the experiment, 37 test suites were used. In the table, only a part of this is shown due to space constraints. The table has been shortened compared to the real one. The repeating "org.eclipse" has been shortened to [oe]. CoreTA and UITA are the "core test application" and the "UI test application" bundle names, respectively. The first three bundles are bundles required to run tests for Eclipse-based product lines.

Running the Tests To run the tests, the following command is used for each test suite of each feature included in the product:

```
<eclipse executable> -application <TestApp> -os <os> -ws <ws> -arch <arch>
    -dev bin -testpluginname <bundle> -classname <testClass> formatter=org.
    apache.tools.ant.taskdefs.optional.junit.XMLJUnitResultFormatter,<xml-
    output-file>
```

We have all the information needed to fill in this boilerplate in the bindings file. The output file records the test results. The information can be extracted and put into a report.

Table E.4: Tests and Results for Testing the Eclipse IDE Product Line, Figure E.5, Using the 2-wise Covering Array of Table E.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							-	-		-		-		

Results We implemented Automatic CIT for the Eclipse Platform plug-in system, and we created a feature mapping for 37 test suites. The result of this execution is shown in Table E.4b. This experiment[6] 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 E.4b, 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 that there were no tests for this feature in the test setup. The cells with numbers show the number of errors produced by running the tests available for that feature.

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

We executed the whole thing several times to ensure that the results were not coincidental, and we looked at the execution log to make sure that the problems were not caused by the experimental set up such as lacking 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 unavailable to us during our research.

[6]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.